Learning AWS

Second Edition

Design, build, and deploy responsive applications using AWS Cloud components



Packt>

www.packt.com

By Aurobindo Sarkar and Amit Shah

Learning AWS
Second Edition

Design, build, and deploy responsive applications using AWS Cloud components

Aurobindo Sarkar
Amit Shah

Packt>

BIRMINGHAM - MUMBAI

Learning AWS
Second Edition

Copyright © 2018 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Commissioning Editor: Vijin Boricha
Acquisition Editors: Prateek Bharadwaj, Shrilekha Inani
Content Development Editor: Sharon Raj
Technical Editor: Mohit Hassija
Copy Editors: Safis Editing, Dipti Mankame, Laxmi Subramanian
Project Coordinator: Virginia Dias
Proofreader: Safis Editing
Indexer: Francy Puthiry
Production Coordinator: Deepika Naik

First published: July 2015
Second edition: January 2018

Production reference: 1310118

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78728-106-6

www.packtpub.com



mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

PacktPub.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the authors

Aurobindo Sarkar is currently the country head (India Engineering Center) for ZineOne Inc. With a career spanning over 25 years, he has consulted at some of the leading organizations in India, the US, the UK, and Canada. He specializes in real-time architectures, machine learning, cloud engineering, and big data analytics. Aurobindo has been actively working as a CTO in technology startups for over 8 years now. He also teaches machine learning courses at business schools and corporates.

I would like to thank the editors and publishing staff at Packt for the opportunity to write the second edition of this book. I would especially like to thank Sharon Raj, Content Development Editor, who worked with me patiently.
Most of all, I am thankful to my wife, Nitya, and kids, Somnath, Ravishankar, and Nandini.

Amit Shah has a bachelor's degree in electronics. He is a director at Global Eagle. He has been programming since the early '80s, the first wave of personal computing—initially as a hobbyist and then as a professional. His areas of interest include embedded systems, IoT, analog, and digital hardware design, systems programming, cloud computing, and enterprise architecture. He has been working extensively in the fields of cloud computing and enterprise architecture for the past 7 years.

About the reviewer

Rishabh Sharma has around 7+ years of cloud operations and architectural experience in the Fortune 500, MNCs, and start-ups. Currently, he is working as a deputy system manager in a reputed IT company in Hong Kong. He has authored many research papers in international journals and IEEE on a variety of issues related to cloud computing and has authored six technical books to date including hands on guides. He recently published four books internationally: *Cloud Computing: Fundamentals, Industry Approach and Trends*, *Learning OpenStack High Availability*, and *Learning Chef*.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface 1

Chapter 1: Cloud 101 – Understanding the Basics 6

- **Defining cloud computing** 7
- **Introducing public, private, and hybrid clouds** 8
- **Introducing cloud service models – IaaS, PaaS, and SaaS** 9
- **Introducing multi-tenancy models** 10
- **Understanding cloud-based workloads** 13
  - Migrating on-premise applications to the cloud 13
  - Building cloud-native applications 15
- **Setting up your AWS account** 15
  - Creating a new AWS account 16
  - Exploring the AWS management console 21
- **Summary** 26

Chapter 2: Designing Cloud Applications 27

- **Introducing cloud-based multitier architecture** 28
- **Designing for multi-tenancy** 30
  - Addressing data-at-rest security requirements 32
  - Addressing data extensibility requirements 34
- **Understanding cloud applications design principles** 39
  - Designing for scale 39
  - Automating cloud infrastructure 42
  - Designing for failure 43
  - Designing for parallel processing 44
  - Designing for performance 45
  - Designing for eventual consistency 46
- **Understanding emerging cloud-based application architectures** 47
  - Understanding polyglot persistence 49
  - Understanding Lambda architecture 50
  - Understanding Kappa architecture 51
  - Deploying cloud-based machine learning pipelines 51
  - Deploying cloud-based machine learning models 53
- **Estimating your cloud computing costs** 53
- **A typical e-commerce web application** 56

Setting up your development environment 58
Running the application 61
Building a war file for deployment 63
Application structure 64
Summary 66
Chapter 3: Introducing AWS Components 67
AWS components 68
Amazon compute-related services 68
Amazon EC2 68
Amazon EC2 container service 69
AWS Lambda 69
Amazon storage-related services 69
Amazon S3 69
Amazon EBS 70
Amazon Glacier 70
Amazon database-related services 70
Amazon Relational Database Service (RDS) 70
Amazon DynamoDB 71
Amazon Redshift 71
Amazon ElastiCache 72
Amazon messaging-related services 72
Amazon SQS 72
Amazon SNS 72
Amazon SES 72
Amazon Pinpoint 73
Amazon networking and content delivery services 73
Amazon VPC (Virtual Private Cloud) 73
Amazon Elastic Load Balancing 73
Amazon Route 53 74
Amazon CloudFront 74
AWS Direct Connect 74
Amazon management tools 74
AWS CloudFormation 74
Amazon CloudWatch 74
AWS CloudTrail 75
Amazon security, identity, and compliance services 75
AWS Identity and Access Management (IAM) 75
AWS Directory Service 76
Amazon Certificate Manager 76
AWS Key Management Service 76
AWS WAF 76
Amazon analytics-related services 76
Amazon EMR 77
Amazon Kinesis 77

Amazon machine learning/AI-related services 77
Amazon Machine Learning 77
Other Amazon AI-related services 78
Other Amazon services 78
Managing costs on AWS cloud 79
Setting costs-related objectives 79
Optimizing costs on the cloud 79
Strategies to lower AWS costs 81
Monitoring and analyzing costs 81
Choosing the right EC2 Instance 82
Turn-off unused instances 84
Using Auto Scaling 84
Using reserved instances 86
Using spot instances 87
Using Amazon S3 storage 88
Optimizing database utilization and costs 89
Using AWS services 90
Using queues 91
Application development environments 91
Development environment 91
QA/test environment 92
Staging environment 92
Production environment 92
Setting up the AWS infrastructure 92
AWS Cloud deployment architecture 93
AWS cloud construction 95
Creating security groups 95
Creating EC2 instance key pairs 97
Creating roles 98
Creating an EC2 instance 101
Creating and associating Elastic IPs (EIP) 109
Configuring the Amazon Relational Database Service (RDS) 112
Installing and verifying the software stack 126
Summary 129
Chapter 4: Designing for and Implementing Scalability 130
Defining scalability objectives 130
Designing scalable application architectures 131
Using AWS services for out-of-the-box scalability 132
Using a scale-out approach 132
Implementing loosely-coupled components 132
Implementing asynchronous processing 133
Leveraging AWS infrastructure services for scalability 133

Using AWS CloudFront to distribute content 134
Using AWS ELB to scale without service interruptions 134
Using Amazon CloudWatch for Auto Scaling 134
Scaling data services 135
Scaling proactively 136
Using the EC2 container service 136
Evolving architecture against increasing loads 137
Scaling from one to half a million users 137
Scaling from half a million to a million users 139
Scaling from a million to ten million users 141
Event handling at scale 141
Implementing a large-scale API-based architecture with AWS services 142
Using Amazon API Gateway 142
Using AWS Lambda 142
Using Kinesis Streams 143
Using Elasticsearch 143
Analyzing streaming data in real time with Amazon Kinesis Analytics 144
Using Amazon Kinesis Firehose 145
Using Amazon Kinesis Analytics 145
Building real-time applications with Amazon Kinesis Analytics 145
Setting up Auto Scaling 146
AWS Auto Scaling construction 146
Creating an AMI 146
Creating the Elastic Load Balancer 149
Creating launch configuration 158
Creating an Auto Scaling group 164
Testing Auto Scaling groups 175
Summary 176
Chapter 5: Designing for and Implementing High Availability 177
Defining availability objectives 178
Nature of failures 179
Setting up VPC for high availability 179
Using ELB and Route 53 for high availability 180
Instance availability 180
Auto Scaling for increased availability and reliability 181
Zonal Availability or Availability Zone Redundancy 182
Region availability or regional redundancy 182
Setting up high availability for application and data layers 183
Implementing high availability in the application 185
Using AWS for disaster recovery 186
Using a backup and restore DR strategy 187
Using a Pilot Light architecture for DR 187

Using a warm standby architecture for DR 187
Using a Multi-Site architecture for DR 188
Testing disaster recovery strategy 188
Setting up high availability 189
AWS high availability architecture 189
HA support for Elastic Load Balancer 194
HA support for the Relational Database Service 197
Summary 202
Chapter 6: Designing for and Implementing Security 203
Defining security objectives 204
Understanding the security responsibilities 205
Best practices in implementing AWS security 206
Security considerations while using CloudFront 208
CloudFront and ACM integration 209
Understanding access control options 210
Web Application Firewall 210
Securing the application 211
Implementing Identity Lifecycle Management 211
Tracking AWS API activity using CloudTrail 212
Logging for security analysis 212
Using third-party security solutions 212
Reviewing and auditing security configuration 213
Setting up security 214
Using AWS IAM to secure an infrastructure 214
Understanding IAM roles 215
Using the AWS Key Management Service 217
Creating KMS keys 218
Using the KMS key 222
Application security 223
Implementing transport security 224
Generating self-signed certificates 224
Configuring ELB for SSL 225
Securing data at rest 230
Securing data on S3 230
Using the S3 console for server-side encryption 230
Securing data on RDS 235
Summary 235
Chapter 7: Deploying to Production and Going Live 236
Managing infrastructure, deployments, and support at scale 237
Creating and managing AWS environments using CloudFormation 238
Creating CloudFormation templates 240

Building a DevOps pipeline with CloudFormation 241
Updating stacks 242
Extending CloudFormation 246
Using CloudWatch for monitoring 246
Using AWS solutions for backup and archiving 247
Planning for production go-live activities 249
Setting up for production 250
AWS production deployment architecture 250
VPC subnets 252
Private subnet 252
Bastion subnet 258
Bastion host 258
Security groups 259
Infrastructure as Code 261
Setting up CloudFormation 261
Centralized logging 268
Setting up CloudWatch 269
Summary 270
Chapter 8: Designing a Big Data Application 271
Introducing big data applications 272
AWS components used in big data applications 274
Analyzing streaming data with Amazon Kinesis 274
Best practices for serverless big data applications 275
Best practices for using Amazon EMR 276
Understanding common EMR use cases 277
Lowering EMR costs 278
Using Amazon EC2 Spot and Auto Scaling 279
Best practices for distributed machine learning and predictive analytics 280
Using Amazon SageMaker for machine learning 282
Understanding Amazon SageMaker algorithms and features 283
Security overview for big data applications 284
Securing the EMR cluster 284
Encryption 284
Authentication 285
Authorization 285
Securing serverless applications 286
Understanding serverless application authentication and authorization 287
Configuring and using EMR-Spark clusters 290
Summary 308

Chapter 9: Implementing a Big Data Application 309
Setting up an Amazon Kinesis Stream 310
Creating an AWS Lambda function 314
Using Amazon Kinesis Firehose 320
Using AWS Glue and Amazon Athena 325
Using Amazon SageMaker 341
Summary 348
Chapter 10: Deploying a Big Data System 349
Using CloudFormation templates 350
Creating a data lake using a CloudFormation template 350
Authoring and deploying serverless applications 360
Understanding AWS SAM 372
Understanding the SAM template 372
Introducing SAM Local 373
Developing serverless applications using AWS Cloud9 374
Automating serverless application deployments 374
Using AWS Serverless Application Repository 376
Summary 377
Appendix A: Other Books You May Enjoy 378
Leave a review - let other readers know what you think 380
Index 381

Preface

The main focus of this book is to cover cloud concepts followed by design, development, and deployment of scalable, available, and secure applications on AWS. We will introduce you to the fundamental AWS building blocks such as compute instances, storage, security, and networking. We will start by helping you to set up your AWS account and then explain the wide variety of AWS service offerings, cloud environments, and costing models.

This book will not only guide you through various design decision trade-offs and ideas but will also illustrate the implementation of popular use cases, frameworks, and application architectures to get the most out of AWS services. You will also understand the guiding principles and best practices for using AWS services to implement cost-efficient application architectures.

We will explain high-availability, scalability and auto-scaling, security, serverless computing, and Infrastructure as Code concepts on AWS. We will also cover disaster recovery, production deployments and application monitoring techniques in real-world AWS applications. Finally, we will cover the design, implementation, and deployment of emerging applications such as big data analytics, real-time streaming and machine learning applications on AWS.

By the end of this book, you will be well versed with various services that AWS provides and you will learn to design and build scalable, highly available, and secure AWS applications. More specifically, we will cover key architectural components and patterns in large-scale AWS applications that architects and designers will find useful as building blocks for their own specific use cases.

Who this book is for

If you are a developer, engineer, or an architect new to AWS environment, then this book is for you. You will also find this book useful, if you have some prior experience designing and building on-premise applications and are now considering migrating or refactoring your applications to run on AWS cloud. Additionally, if you are keen to explore newer AWS services to build big data, real-time streaming, and machine learning applications on AWS, then you will find the last three chapters specifically covering these topics in detail. Some previous to Java and/or Python programming is all you need to get started with this book.

What this book covers

`Chapter 1`, *Cloud 101 – Understanding the Basics*, describes basic cloud concepts, including the public, private, and hybrid cloud models. It explains and compares the **Infrastructure as a Service** (**IaaS**), **Platform as a Service** (**PaaS**), and **Software as a Service** (**SaaS**) cloud service delivery models. Finally, the next generation of applications being deployed on the cloud, including streaming applications, machine learning pipelines, and deep learning applications are discussed.

`Chapter 2`, *Designing Cloud Applications*, describes familiar and not-so-familiar architectural best practices in the cloud context, including multi-tier architecture, multi-tenancy, scalability, and availability. This chapter introduces design considerations for cloud-based big data applications, batch and streaming architectures, and ML pipelines. It also provides guidelines for estimating cloud-computing costs.

`Chapter 3`, *Introducing AWS Components*, introduces AWS components such as EC2, S3, RDS, DynamoDB, SQS Queues, and SNS. It discusses strategies to lower AWS infrastructure costs and their implications on architectural decisions. It explains the typical characteristics of the development, QA, staging, and production environments on the AWS Cloud.

`Chapter 4`, *Designing for and Implementing Scalability*, provides guidance on how to define your scalability objectives, and then discusses the design and implementation of specific strategies to achieve scalability.

`Chapter 5`, *Designing for and Implementing High Availability*, provides guidance on how to define availability objectives, discusses the nature of failures, and explains design and implementation of specific strategies to achieve high availability. In addition, this chapter describes the approaches that leverage the AWS features and services for disaster-recovery planning.

`Chapter 6`, *Designing for and Implementing Security*, provides guidance on how to define security objectives, explains security responsibilities, and then discusses the implementations of specific best practices for application security.

`Chapter 7`, *Deploying to Production and Going Live*, provides guidance on managing infrastructure, deployments, support, and operations for your cloud application. In addition, it offers some tips on planning production go-live activities.

`Chapter 8`, *Designing a Big Data Application*, introduces the design principles and best practices for using services such as Kinesis, EMR, and Lambda to design AWS-based big data applications.

`Chapter 9`, *Implementing a Big Data Application*, implements several popular AWS-based big data use cases using AWS services such as Kinesis, EMR, Lambda, Glue, and Spark.

`Chapter 10`, *Deploying a Big Data Application*, introduces CloudFormation templates to deploy several popular big data stacks on AWS, including streaming and machine learning applications.

To get the most out of this book

This book primarily requires an AWS account for the hands-on sessions contained in each chapter. For the sample applications, we require Eclipse Java IDE (latest version) and Python 2.7 or 3.6. Maven builds takes care of all other dependencies.

Hardware and OS specifications includes laptop or desktop with an internet connection, and Windows, Linux, or macOS X (preferably the latest versions).

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://www.packtpub.com/sites/default/files/downloads/LearningAWSSecondEdition_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system."

A block of code is set as follows:

```
public class KMSClient{
    private String keyId = "arn:aws:kms:us-
west-2:450394462648:key/1cd0e2d5-61e1-4a71-a6b2-b9db825c9fce";
    private AWSCredentials credentials;
    private AWSKMSClient kms;

    public KMSClient(){
        credentials = new BasicAWSCredentials(accessKey, secretKey);
    kms = new AWSKMSClient(credentials);
    kms.setEndpoint("kms.us-west-2.amazonaws.com");
 }
```

Any command-line input or output is written as follows:

```
mkdir a1electronics
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "From the EC2 navigation pane, click on **Instances** to view all your EC2 instances."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com` and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packtpub.com`.

1

Cloud 101 – Understanding the Basics

In this chapter, we will introduce you to cloud computing and the key terminologies used commonly by cloud practitioners. We will briefly describe what public, private and hybrid clouds are, followed by a description of various cloud service models (offered by the service providers) including the features of **Infrastructure as a Service** (**IaaS**), **Platform as a Service** (**PaaS**) and **Software as a Service** (**SaaS**).

One of the main cloud-based product design elements is multi-tenancy; often considered critical from a profitability and ROI perspective. So, we will spend some time discussing, at a high level, models of multi-tenancy and their implications on design and operations.

We will also discuss some of the traditional workloads being shifted to the cloud and others being developed from the ground up, to leverage cloud services, extensively. These include shifting in-premise systems to the cloud, replacing in-premise product offerings such as ERP and CRM applications with cloud-based versions, and using a mix of in-premise and cloud-based systems. Additionally, we will look at how a large number of modern big data applications, such as recommendation engines, large-scale analytics applications, machine learning pipelines, deep learning workloads, are being targeted for cloud environment deployment only.

To help you get started on AWS, we will end the chapter by walking you through a step-by-step process of creating an AWS account and describing some of the salient features of the AWS dashboard.

This chapter will cover the following points:

- Define cloud computing and describe some of its characteristics
- Describe and compare public, private, and hybrid clouds
- Explain and compare IaaS, PaaS, and SaaS cloud service delivery models
- Explain multi-tenancy models and some challenges they present in design, implementation and operations
- Briefly describe typical cloud-based workloads
- Outline the steps to create an AWS account
- A brief overview of the AWS management console

Defining cloud computing

Wikipedia defines cloud computing as:

> *"…internet-based computing in which large groups of remote servers are networked to allow the centralized data storage, and online access to computer services or resources."*

The National Institute of Standards and Technology (NIST) gives the following definition of cloud computing:

> *"…a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."*

There are several other broadly accepted definitions of cloud computing. Some explicitly emphasize configurability of the resources, while others include the need for rapid on-demand provisioning of resources, still others drop the requirement of access via the internet. We define cloud computing as a model that enables the features listed as follows:

- Users should be able to provision and release resources on-demand
- The resources can be scaled up or scaled down, automatically, depending on the load
- The provisioned resources should be accessible over a network
- Cloud service providers should enable a pay-as-you-go model where customers are charged, based on the type and quantum of resources they consume

Some of the implications of choosing to use the cloud for your computing needs are:

- The illusion of infinite processing and storage resources available on demand reduce the need for detailed advance planning and procurement processes.
- The model promotes the use of resources as per customer needs, for example, starting small, and then increasing resources based on an increase in need.
- Provisioning development and test environments on a smaller scale, and enabling them only during working hours to reduce the cost of development.
- The staging environment can be provisioned for a short duration to be a replica of the production environment. This enables testing using production configuration (and scale) and for improved production defect resolution.
- The ability to auto scale in order to better manage spikes in demand and variations due to business cycles or time-of-day reasons, and so on.
- It encourages experimentation by trying out new ideas and software by quickly provisioning resources rather than requisition for resources through time consuming and cumbersome processes.

These and other implications of using cloud services to design scalable, highly available and secure applications are discussed in depth in the subsequent chapters.

Introducing public, private, and hybrid clouds

Basically, there are three types of clouds in cloud computing, they are public, private and hybrid clouds.

In a **public cloud**, third-party service providers make resources and services available to their customers via the internet. The customers' applications and data are deployed on infrastructure that is owned and secured by the service provider.

A **private cloud** provides many of the same benefits of a public cloud but the services and data are managed by the organization, or a third-party, solely, for the customer's organization. Usually, a private cloud places increased administrative overheads on the customer but gives greater control over the infrastructure and reduces security-related concerns. The infrastructure may be located on or off the organization’s premises.

A **hybrid cloud** is a combination of both a private and a public cloud. The decision on what runs on the private versus the public cloud is usually based on business criticality of the application and sensitivity of the data. But in some cases, spikes in demand for resources, or spillovers, in the private cloud are also handled in the public cloud.

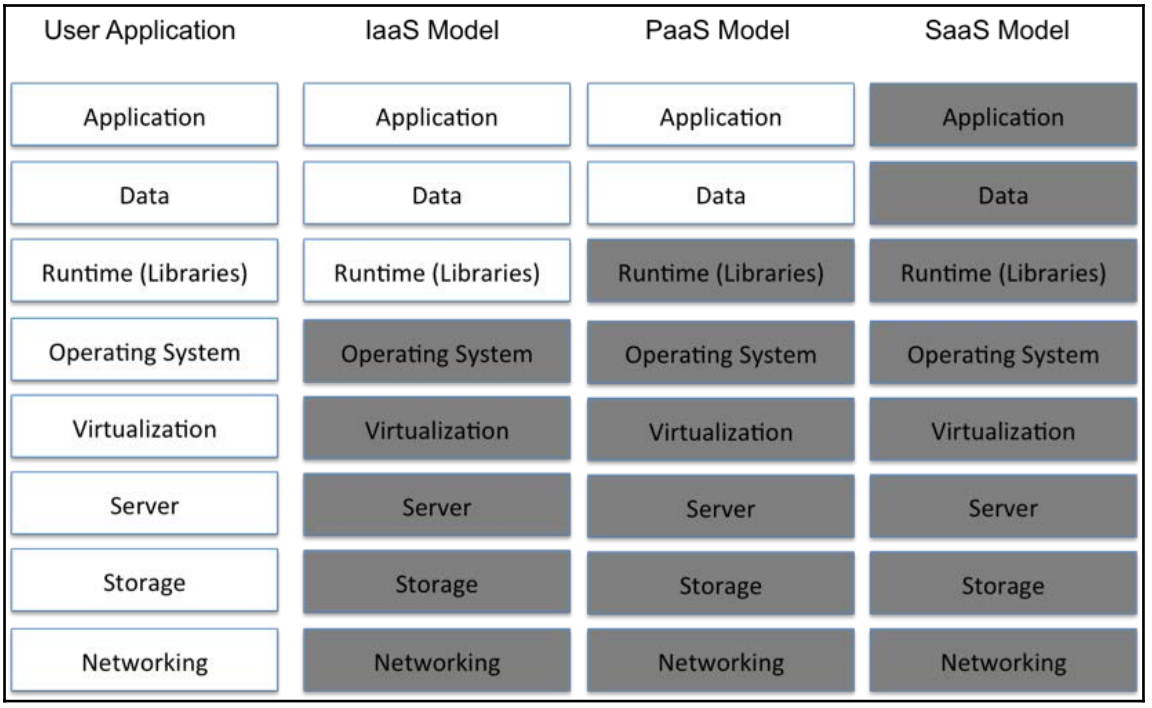
Introducing cloud service models – IaaS, PaaS, and SaaS

There are three cloud-based service models. The main features of each of these are listed as follows:

- IaaS provides a capability for users to provision processing, storage, and network resources on demand. The customers deploy and run their own applications on these resources. Using this service model is closest to the traditional in-premise models but without the lengthy procurement processes. The onus of administering these resources rests, largely, with the customer.
- In PaaS, the service provider makes certain core components such as databases, queues, workflow engines, email, and so on, available as services to the customer. The customer then leverages these components for building their own applications. The service provider ensures high service levels, and is responsible for scalability, high availability, and so on, for these components. This allows customers to focus a lot more on their application functionality. However, this model also leads to application-level dependency on the providers' services.
- In the SaaS model, typically, third-party providers using a subscription model provide end user applications to their customers. The customers may have some administrative capability at the application level, for example, to create and manage their users. Such applications also provide some degree of customizability, for example, the customers can use their own corporate logos, colors, and so on. Applications that have a very wide user base most often operate in a self-service mode. In contrast, the provider provisions the infrastructure and the application for the customer for more specialized applications. The provider also hands over the management of the application to the customer's application administrator (in most cases this is limited to user management tasks).

From an infrastructure perspective, the customer does not manage or control the underlying cloud infrastructure in all three service models.

The following figure illustrates who is responsible for managing the various components of a typical user application across IaaS, PaaS, and SaaS cloud service models. The shaded boxes represent the service-providers' responsibilities, while the other boxes represent the users' or end customers' responsibilities.



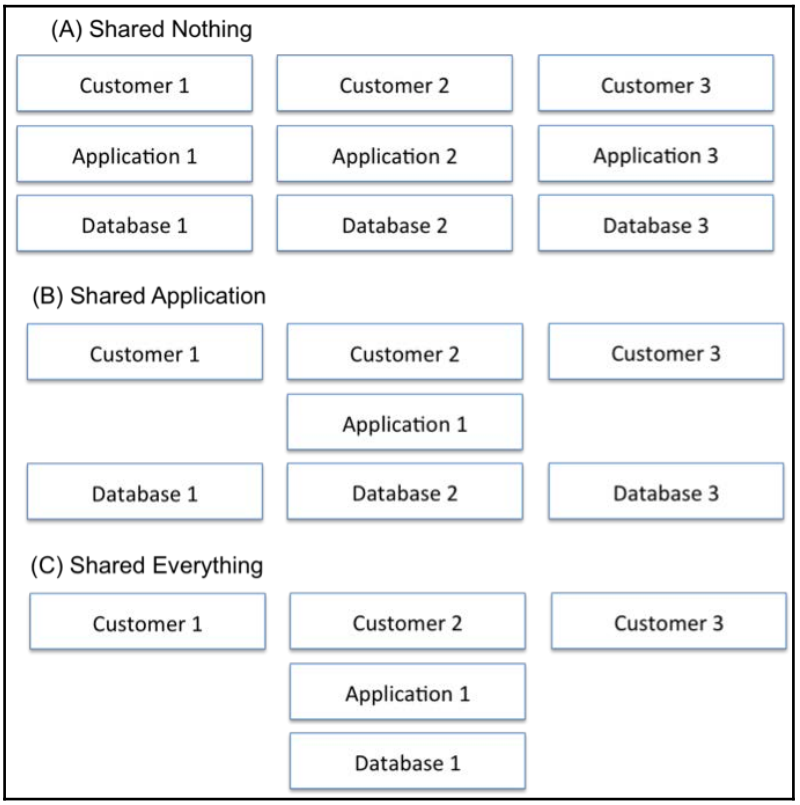
The level of control the user has over operating systems, storage, applications, and certain network components (for example, load balancers) is the highest in the IaaS model while the least (or none) in the SaaS model.

Introducing multi-tenancy models

Multi-tenancy and multi-tenant architecture come up, repeatedly, especially in the context of cloud product design and architecture discussions. Given that most SaaS products are offered as a subscription, it is vital to understand this concept clearly as it can often be the difference between having a highly profitable and an easy-to-manage product, and a failed product or venture.

Multi-tenancy is a principle in software architecture where a single instance of the software serves multiple tenants or customers. The realization of this concept in software design is probably one of more complex tasks in implementing and operating a cloud-based product.

Let's start by examining three basic models of a multi-tenant application. The following figure illustrates these models:



In the first model, **(A) Shared Nothing** architecture, each customer has a separate copy of the application and the database for their exclusive use. In some cases, the hardware infrastructure is also separated out for each customer. Large enterprises may insist on this separation mainly due to security concerns but also due to service-level concerns associated with resource sharing. This is essentially application hosting rather than application multi-tenancy.

The second model, **(B) Shared Application** architecture, shares the same application instance but the data is separated for each customer. And in the third model, **(C) Shared Everything** architecture, both the application and the database instances are shared resources among all the customers.

In real life, it is fairly common for customers to request a dedicated application and infrastructure stack. Most smaller companies and start-ups give in under pressure, especially if it is a major customer. However, this can be a very expensive option to sustain over a period of time. Before you know it, you are maintaining several different versions of the application and technology stacks, multiple database schema, operational and support-related overheads across a set of customers. This makes application maintenance, QA/testing effort, upgrades/releases, and customer support impossibly complex or expensive or both. You are no longer an SaaS application provider!

It is very common and reasonable to expect a majority of customers to request that their data be kept separated from other customers. It is also common for smaller businesses and price-sensitive customers to not care how and where you store their data, as long as they are satisfied by the security measures you have implemented. In all cases, it is imperative to use encryption for the securing of sensitive data-at-rest.

Hence, in reality you end up with a mix of models where the concern is more focused on data security rather than shared infrastructure or application code (as long as you can provide a certain level of application customizability per customer and meet their service-level requirements).

In this scenario, apart from security considerations (discussed in a later chapter) some of the challenges that arise are listed as follows:

- Increased costs of development: Isolating each customer's data in a separate database is easier and faster to build, while using a shared approach requires a larger development effort and initial costs, but can result in being able to serve more tenants per server at a lower overall operational cost.
- Isolating each tenant's database can give you more flexibility in handling each customer's individual requirements (but there could be severe complications when you release product upgrades, especially if your database schema is not designed to handle such changes).
- Regulatory considerations can directly impact your design and leave you with no choice in terms of using a shared approach.

- Backups and restores are simpler operations in the case of isolated databases. In the shared database approach, the impact of servicing a particular tenant's backup or restore request will impact other active customers. These kinds of issues can lead to SLA-related issues and other management overheads related to explaining and issuing appropriate communications to all other customers, and so on.

Normally, businesses charge their customers differently based on whether a customer requests a separate database instance, or a fully segregated infrastructure for their exclusive use only. Suppose you have grouped your customers into three categories such as Platinum, Gold, and Silver. Your Platinum class customers are your biggest and the best. They are willing to pay a premium for additional features and/or better service levels. Let's say, you decide to provision separate infrastructure and database instances, for such customers. Simultaneously, you decide to share the infrastructure and use a single database instance for all your regular, or Silver class, customers. Imagine the operational complexity of a situation where one of your Platinum class customers (**Shared Nothing**) wants to downgrade their subscription level to the Silver class (**Shared Everything**) the following year!

Understanding cloud-based workloads

In this section, we will discuss various workloads being deployed on the cloud. These could be in-premise systems moved to the cloud, on-premise product versions replaced by cloud-based offerings, and new applications being developed for cloud-only environments.

Migrating on-premise applications to the cloud

There are several reasons for organizations wanting to migrate their applications to the cloud. These reasons typically include driving cost efficiency, improving productivity, supporting faster go-to-market strategies, achieving better operational efficiency, and others. Additionally, there are also several different strategies employed to move a portfolio of applications to the cloud.

One of the most commonly used approaches is the lift-and-shift, or rehosting, existing applications in the cloud. This approach can lead to some cost savings, especially if the infrastructure is right-sized and expensive commercial licenses of proprietary products replaced with cloud-based services (from the cloud service provider or third-party service providers) or using equivalent open-source products.

This approach is very popular compared to other approaches as it can be quicker to implement, and some benefits may be realized right away. However, design limitations and application inefficiencies in the existing in-premise application also get migrated to the cloud along with the application. Typically, steady-state applications that are service-oriented, loosely coupled, and with minimal inter dependencies with other applications are the best candidates for using this approach.

A rehosting strategy can lead to disappointments when a changeover to a cloud environment does not yield the expected levels of cost savings or a simpler operating environment. This may be because the full benefits of the cloud are fully realized only when cloud-native designs are implemented for various parts of the architecture. However, resizing infrastructure as per application requirements or replatforming the application to use cloud services or open-source products will definitely lead to increased cost advantages but also take longer to implement.

Most times, subscribing to a product's cloud-based offering or shifting to another equivalent or better cloud product can prove to be an advantageous strategy. For example, shifting to cloud-based offerings of SAP or shifting over to Salesforce for CRM functionality is increasingly becoming a favored strategy in many organizations. Finally, for some systems, it is best to re-factor and/or re-architect the application for deriving the maximum benefits of a migration to the cloud. Whatever the reasons and the strategy for migrating systems to the cloud, it needs to be a well-planned exercise that includes infrastructure, application, and data migration, with significant verification and validation effort at each step in the process.

Typically, migration projects start with an analysis of the existing portfolio of applications to figure out the sequence and strategy for each system to be migrated. Additionally, the speed of such projects picks up as a result of increased exposure to the cloud environment based on the initial set of migrations. The overall strategy in many cases is a mass lift-and-shift followed by iterative improvements introduced in the application architecture over a period of time. Sometimes these migrations are timed to avoid expensive lease and license renewals, and/or hardware refreshes. The portfolio analysis exercise often consolidates and/or rationalizes the hardware and software stacks used in an organization, identifies applications that can be retired at specific points along the journey, and other applications that will never be migrated due to regulatory or other concerns.

Building cloud-native applications

Cloud-native applications are specifically designed and implemented to operate in cloud-only environments. The nature of the application, infrastructure requirements, and data volumes can significantly influence the decision to use the cloud. Smaller organizations and startups often use the cloud for all their infrastructure and software/applications needs. Many such organizations also offer their products on a subscription-based licensing model to their customers (SaaS model).

Applications having wide variability in their usage patterns are great candidates for the cloud. The infrastructure costs in such cases can be reduced significantly by scaling up resources to match the increased demand, and scaling down subsequently to serve lighter loads. Similarly, it is common to scale up for a specific task, such as training a machine learning model (at certain intervals) instead of maintaining high capacity infrastructure, continuously. Specialized workloads requiring high memory, short bursts of high-compute server usage, GPUs, and so on, can leverage the ability to provision resources on demand (as per the requirements). For example, running large-scale deep learning workloads typically require GPU-based instances for quicker turnaround times. These server instances can be spun up and used, only when they are actually required.

Both **streaming applications** with incoming data at very high velocities and batch systems with very high data volumes, can benefit from easy availability and scalability of cloud resources. Additionally, applications using unstructured data such as vast document corpuses, image repositories, and audio and video libraries, can leverage the storage and processing power available on-tap in the cloud. The variety and number of ready-to-use cloud services that are available (via simple APIs) to developers, allows them to build applications without having to worry about the complexities of the underlying service.

We would like to conclude our introduction to cloud computing by getting you started on AWS, right away. The next section will help you set up your AWS account and familiarize you with the AWS management console.

Setting up your AWS account

You will need to create an account on Amazon before you can use the **Amazon Web Services** (**AWS**). Amazon provides a 12-month limited fully functional free account which can be used to learn the different components of AWS. With this account, you get access to many services provided by AWS but there are some limitations based on resources consumed. The list of AWS services is available at `http://aws.amazon.com/free`.

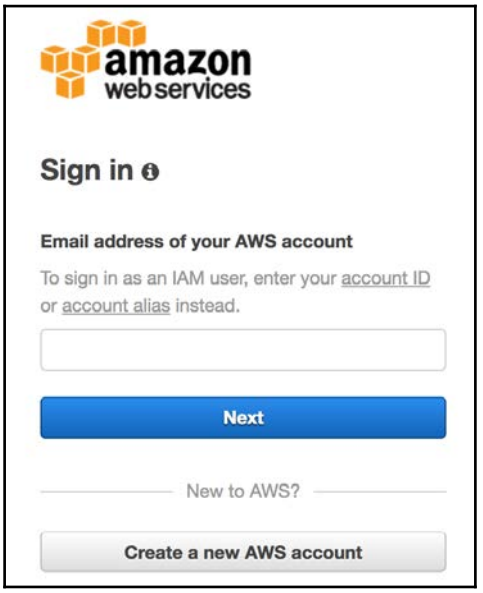
Creating a new AWS account

We are assuming that you do not have a pre-existing AWS account with Amazon (in case you do then please feel free to skip this section).

1. Point your browser to `http://aws.amazon.com/free` and click on **Create a Free Account**.



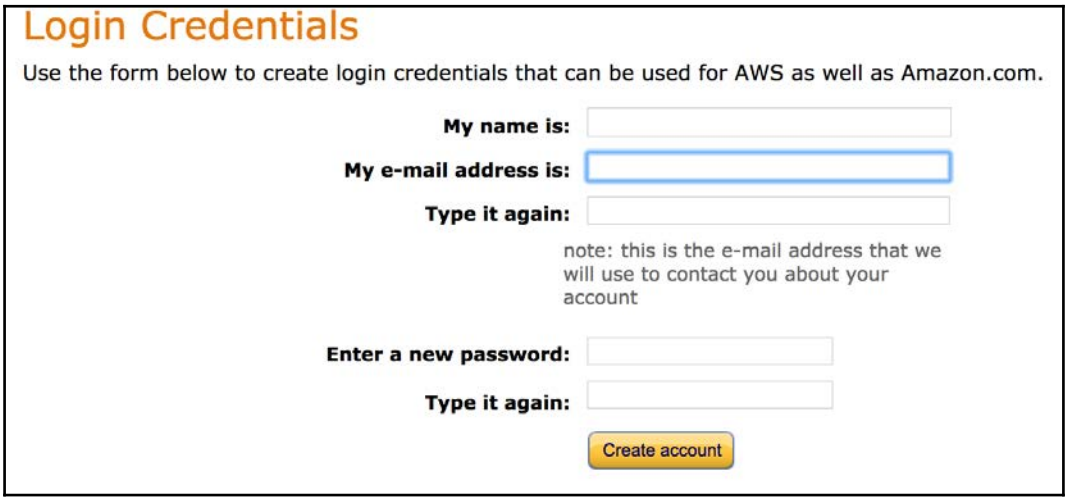
2. Click on **Create a new AWS account**, as shown:



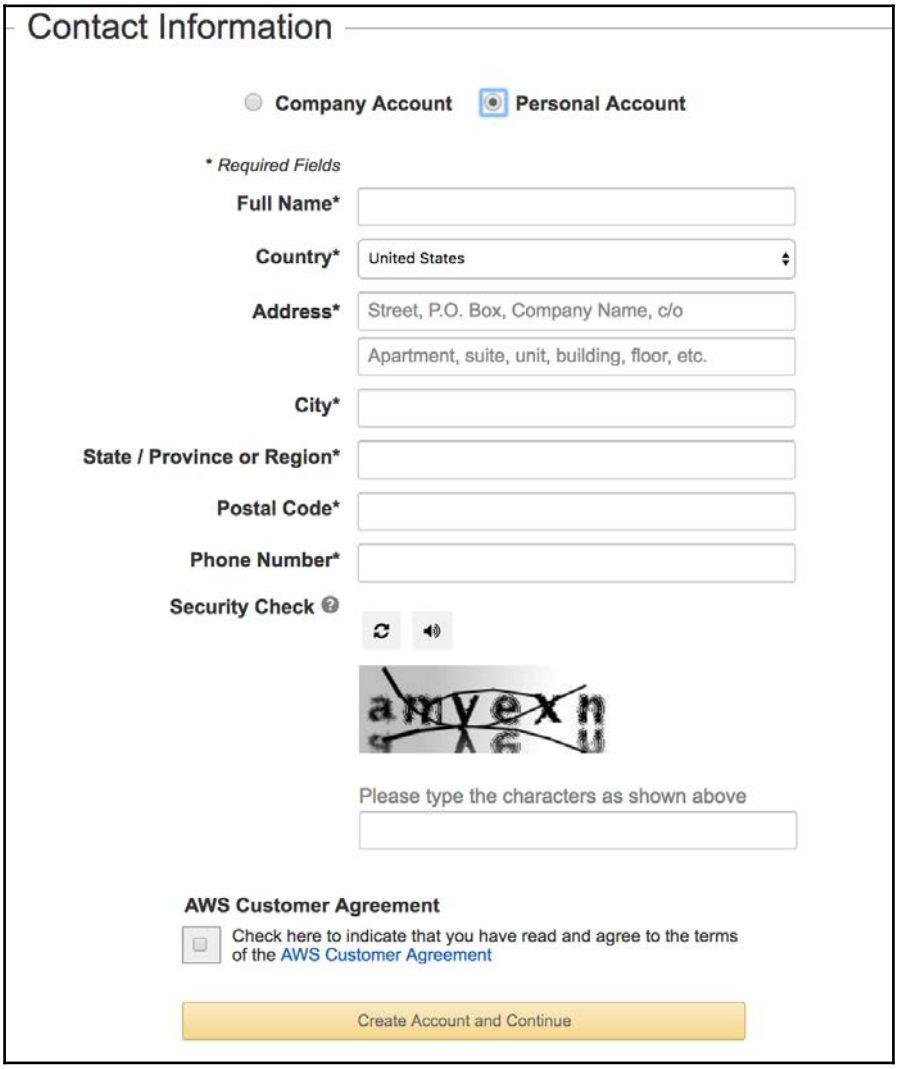
3. Enter your email address, select **I am a new user** option, and then click on **Sign-in using our secure server** button:



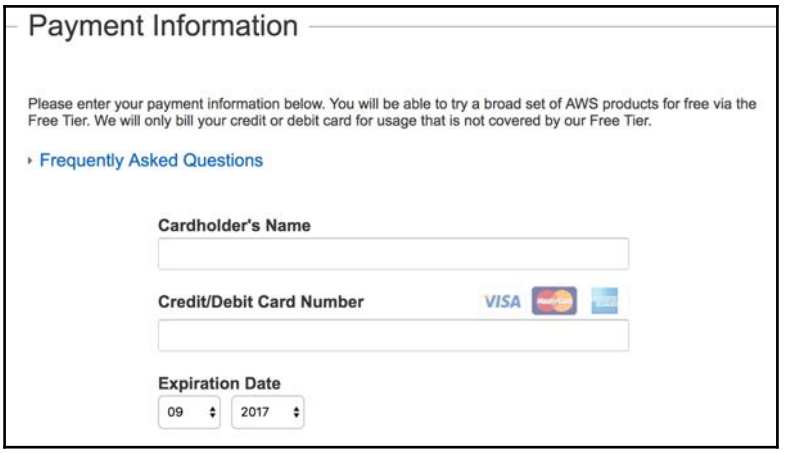
4. A series of intuitive screens shown, as follows, will guide you easily through the process of creating an AWS account. Provide your name, email address, and a password in the form. Click on the **Create account** button to proceed to the next step:



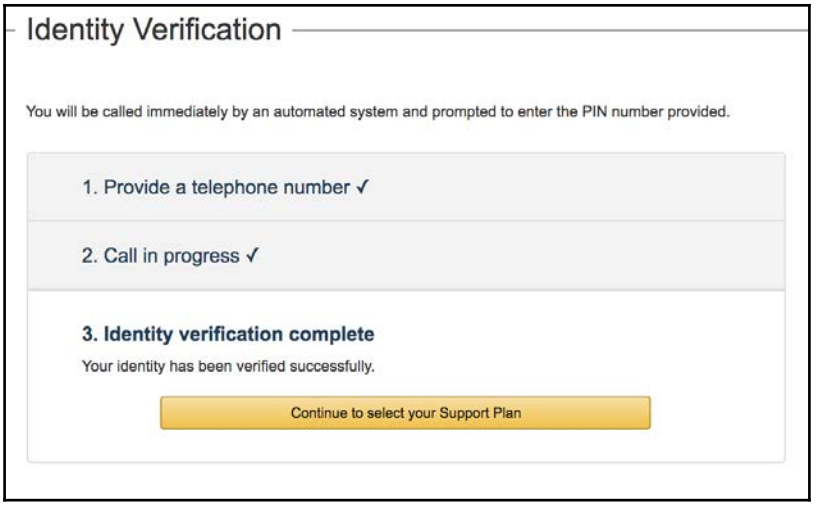
5. Provide the **Contact Information** details as requested in the form shown here. Select **Personal Account** for your initial learning purposes. Amazon uses this information for billing and invoicing:



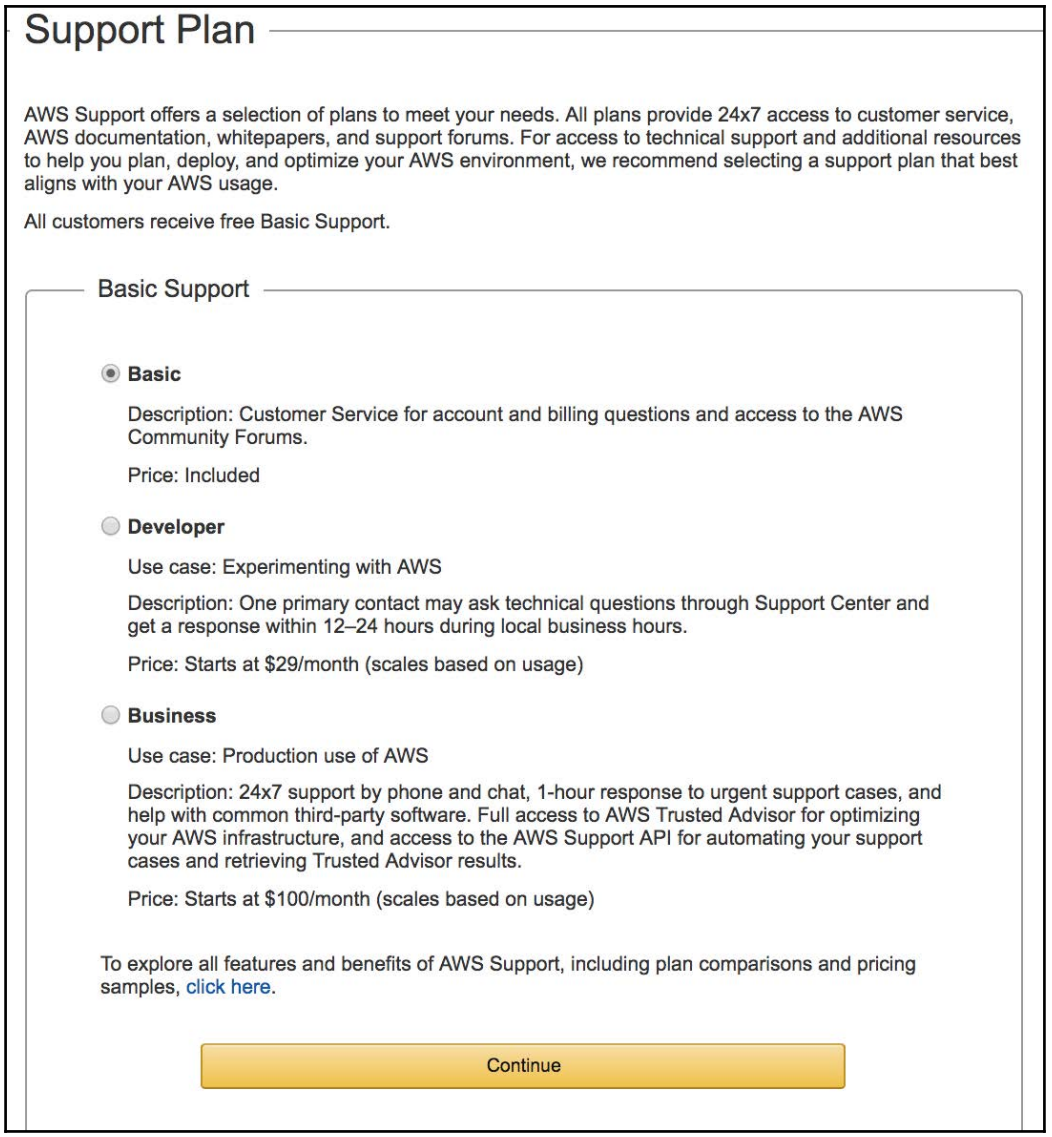
6. Provide **Payment Information** in the form as shown in the following screenshot. When you create an AWS account and sign up for services, you are required to enter the payment information. Amazon will execute a $1 transaction against the card to confirm its validity:



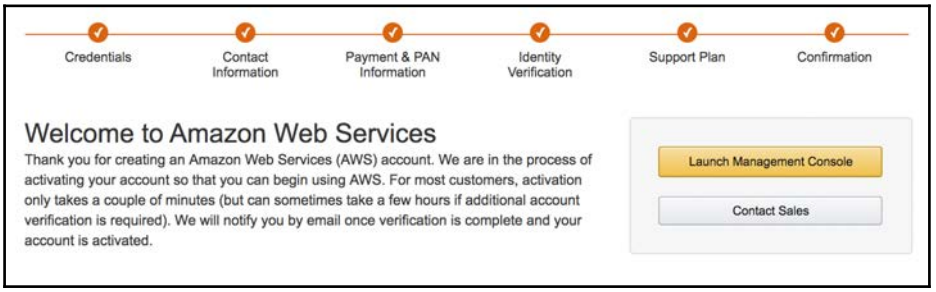
7. Next, Amazon executes an identity verification step. It includes a call back via an automated system to verify your telephone number. You will also need to enter a four digit PIN (displayed on your screen) when prompted. After the verification process is completed, click on **Continue to select your Support Plan** button:



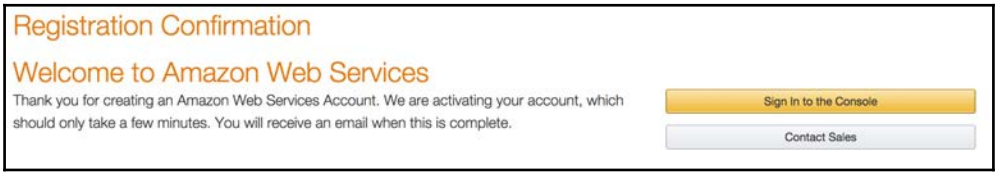
8. Select your **Support Plan**: You can subscribe to one of: **Basic**, **Developer**, **Business**, or **Enterprise** plans. We recommend subscribing to the **Basic** plan at this stage. The **Basic** plan costs nothing but is also limited. You should one of the other plans for production accounts. However, the **Basic** plan is an acceptable option for learning purposes. Click on **Continue** to proceed to the next step:



9. At **Confirmation** stage, you have completed all the steps requiring your input for setting up an AWS account (see all the steps checked at the top of your screen as shown). Click on **Launch Management Console**:



10. At this stage, you have successfully created an AWS account, and you are ready to start using the services offered by Amazon Web Services. On clicking **Sign-in to the Console** button, you will be requested to log in:



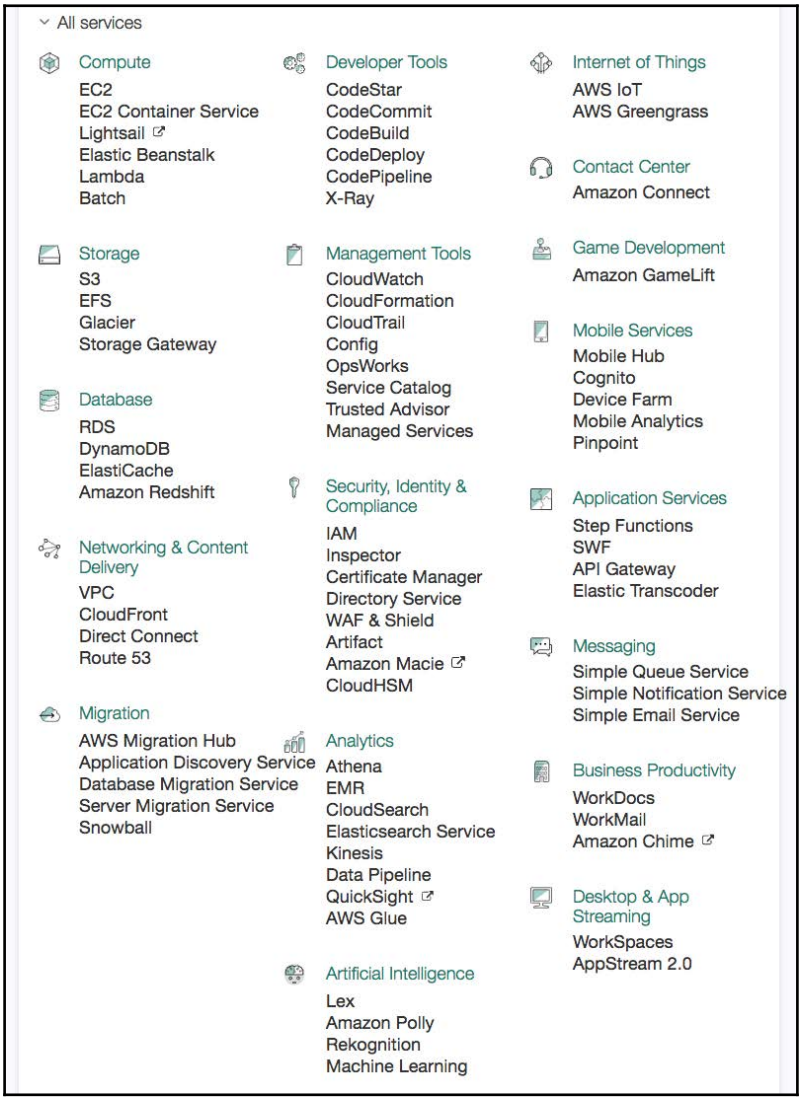
Exploring the AWS management console

The AWS management console is the central location from where you can access all Amazon services. The management console has links to the following:

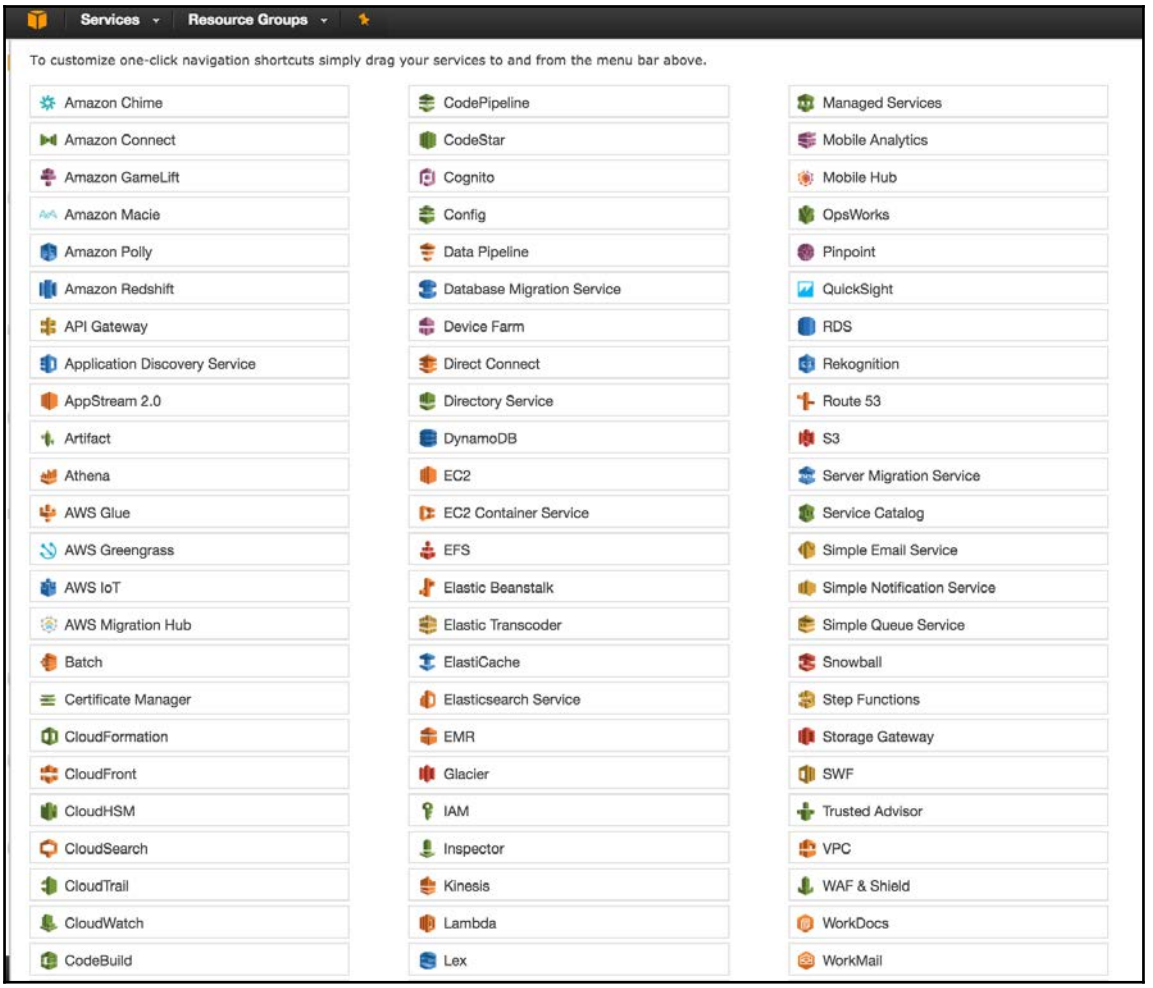
- The home screen of the console is shown as follows:



- Click on **All services** to expand the display. This view lists all the AWS services available in a specific Amazon region. Clicking on any one of these launches the dashboard for the selected service:



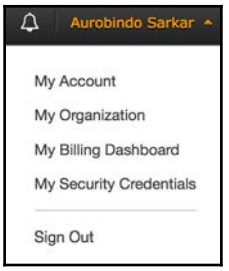
- **Shortcuts for Amazon Web Services**: On the console management screen you can create shortcuts of frequently accessed services by clicking on the pin (located after the **Services** and the **Resource Groups** links) in the title bar. We can drag and drop the services from the list to the title bar to add it:



- The modified title bar after adding **EC2**, **S3**, **RDS** and **VPC** components to it is as shown:



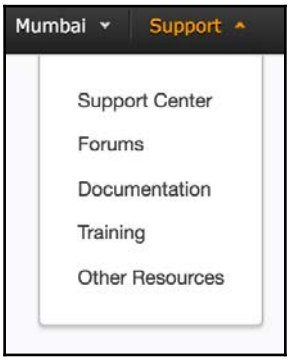
- **Account-related information**: This allows you to access your account-related data. It includes security credentials needed to access the AWS resources, and the **My Billing Dashboard** option gives you real-time information on your current month's billing:



- **Amazon regions**: This option allows you to access the Amazon Web Services in a specific region. For example, the list of Amazon Web Services, shown earlier, were for the **Asia Pacific (Mumbai)** region:



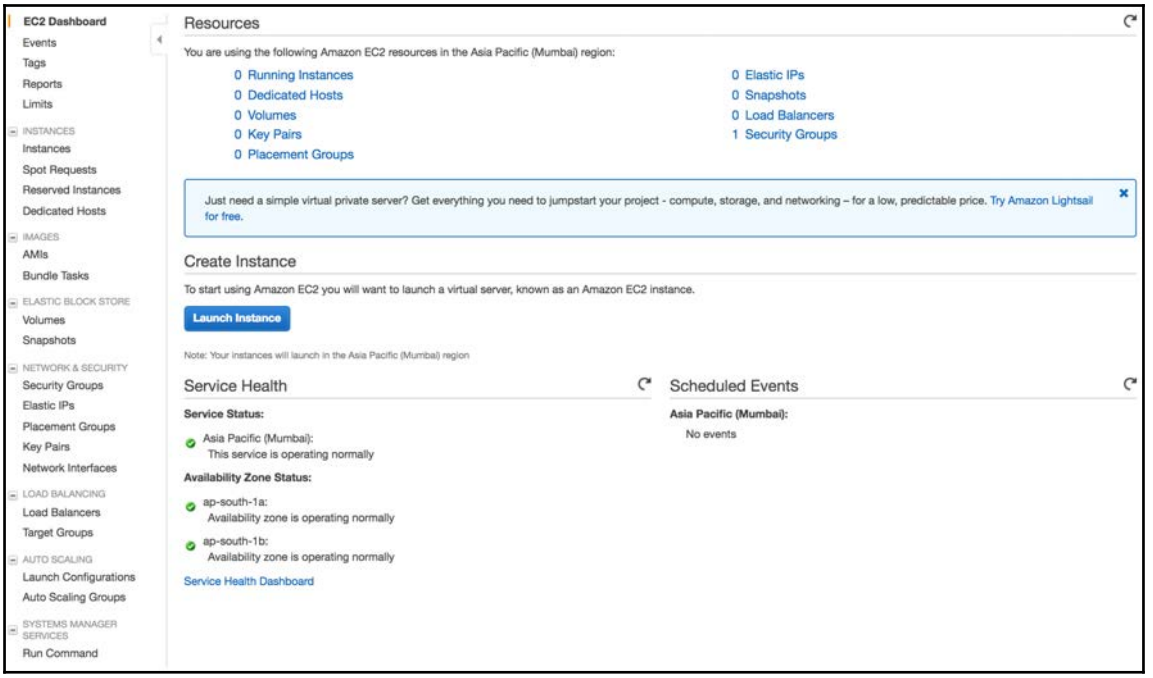
- **Help**: Click on the **Support** menu (located on the title bar) to access help-related items. You can navigate to help, forums and support pages:



- **Services**: Click on the **Services** menu (located on the title bar) to access specific dashboards. For example, click the **EC2** menu item to open the **EC2 Dashboard**:



- **EC2 Dashboard**: Shows the summary of EC2 resources and service health information for the region and associated Availability Zones. You can also launch a new EC2 instance from here:



Summary

In this chapter, we introduced you to a few cloud computing concepts and terminologies. We described the basic features of public, private, and hybrid clouds. We introduced the main cloud delivery models, namely, IaaS, PaaS, and SaaS. We described various multi-tenancy models, and some of their main implications and challenges. We also described typical cloud-based workloads including both traditional in-premise systems and new generation applications built for cloud-only environments. Finally, we listed the steps for creating your AWS account and described the salient features of the AWS management console.

With the basics out of the way, in *Chapter 2, Designing Cloud Applications – an Architect's Perspective*, we will describe some familiar and not-so familiar architectural best practices in the cloud context. We will deep dive into the technical details of how multi-tenanted cloud applications are different from traditional multi-tiered applications. We will also walk you through creating a sample application (using Spring and MySQL) that will be used to illustrate key cloud-application design concepts through the rest of this book.

2
Designing Cloud Applications

As an architect, you should have come across terms such as *loosely coupled*, *multitier*, *service oriented*, *highly scalable*, and so on. These terms are associated with architectural best practices, and you will find them listed in the first couple of pages of any system architecture document. These concepts are generally applicable to all architectures, and the cloud is no exception.

In this chapter, we want to highlight how these are accomplished on the cloud. You will notice that the design principles and best practices for developing application architectures on the cloud, largely remain the same as for on-premise architectures. However, you need to be aware of certain peculiarities specific to the cloud environment, in order to architect scalable, available and secure cloud applications. For example, if you are architecting a web-scale application, you need to take into consideration the ability to scale up and down, automatically, depending on the load. What are the implications of such auto scaling on your design?

One of the major differences in cloud-based SaaS applications and on-premise enterprise applications is multi-tenancy. We will consider key architectural questions, such as: what are some of the design considerations of multi-tenancy? How do you design for UI, services, and data multi-tenancy in a multitier architecture?

We will also introduce architectural patterns being used for machine learning workloads and streaming applications on the cloud.

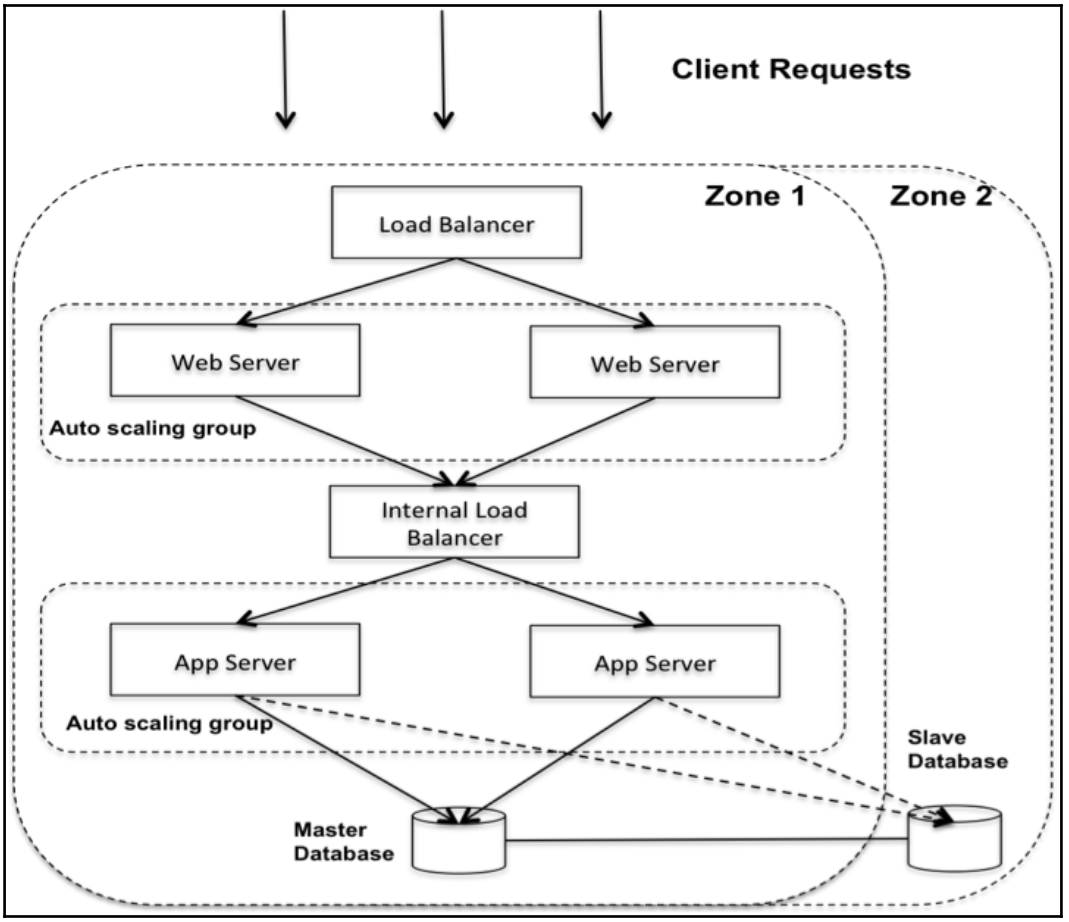
More specifically, we will describe the familiar and not-so familiar architectural best practices in the cloud context by covering the following topics:

- Multitier architecture on the cloud
- Designing for multi-tenancy including data security and extensibility
- Designing for scale
- Automating infrastructure
- Designing for failure
- Parallel processing
- Designing for performance
- Designing for eventual consistency
- Designing for machine learning workloads and streaming applications
- Estimating your cloud computing costs
- Sample application – a typical e-commerce web application

Introducing cloud-based multitier architecture

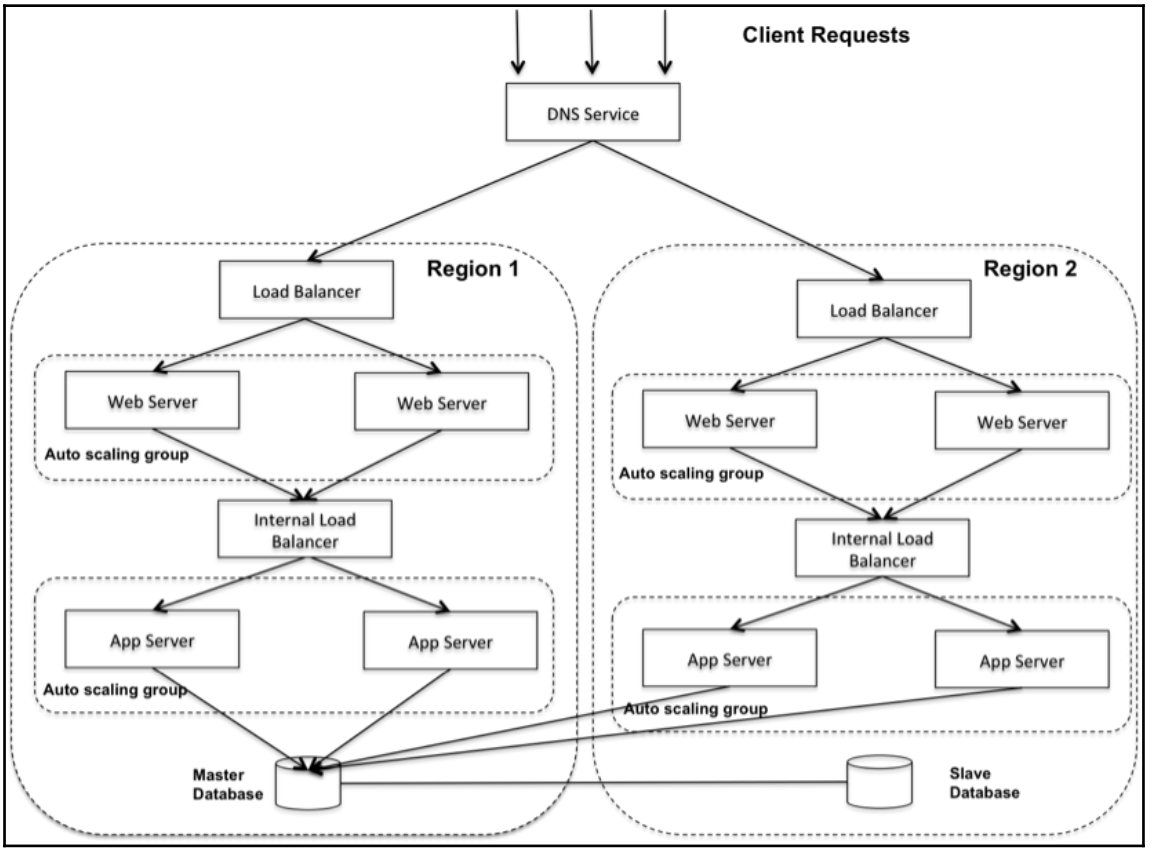
A simple three-tier architecture consists of a web tier, an application or business tier, and the data tier. These tiers are ordinarily implemented using web servers, application servers, and databases, respectively.

The following figure illustrates tiered architecture on the cloud. This architecture supports auto scaling and load balancing of web servers and application servers. Further, it also implements a master-slave database model across two different zones or data centers (connected with high speed links). The master database is synchronously replicated to the slave. Overall, the architecture represents a simple way to achieve a highly scalable and highly available application in a cloud environment.



Cloud applications can be deployed at multiple locations. Typically, using AWS terminology, these locations are regions (that is, separate geographical areas) or zones (that is, distinct locations within a region connected by low-latency networks). It is also possible to separate out the tiers across two different zones regions to provide for a higher level of redundancy including data center wide or zone-level failures or unavailability. We need to consider network traffic flow and data synchronization issues between the regions while designing high-availability architectures across multiple regions. Such issues are discussed in more detail in `Chapter 5`, *Designing for and Implementing High Availability*.

The following architecture diagram illustrates this architecture:



Designing for multi-tenancy

The major benefit of multi-tenancy is cost savings due to shared infrastructure and operational efficiency of managing a single instance of the application across multiple customers or tenants. However, multi-tenancy introduces complexity, and issues can arise when a tenant's action or usage can affect the performance and availability of the application for other tenants on the shared infrastructure. In addition, security, customization, upgrades, recovery, and so on, requirements of one tenant can create issues for other tenants and/or introduce further complexity.

Additionally, customers are concerned about their business-critical data residing on public cloud infrastructure (in the hands of a third-party SaaS application provider). It is of vital importance to ensure that the data architecture is robust and secure to satisfy the security standards, privacy policies, regulatory requirements, etc. in place at large enterprises (as well as smaller businesses). For example, businesses in the financial or healthcare sector are governed by a host of very strict regulatory requirements in terms of privacy, and location and usage of customer data. We need to have strict access control mechanisms to ensure a tenant does not access resources belonging to a different tenant. Simultaneously, we also need to ensure that mechanisms we use are operationally cost-effective and easy to administer.

Customization requirements can easily derail a SaaS product company's business model. Handling differences in data requirements, UI interfaces, business logic, and business workflows requires careful design and implementation across the tiers. In addition, provisioning new tenants, measuring resource usage per tenant, and being able to respond to differences in Quality of Service SLAs per tenant can impose further strain on designers of such applications.

In the multi-tenancy models discussed in `Chapter 1`, *Cloud 101 – Understanding the Basics*, we discussed models that may lie anywhere on the shared-nothing to share-everything continuum. While technical ease may be a key factor from the IT department's perspective, the cloud architect should never lose sight of the business implications and costs of selecting the appropriate approach to implementing multi-tenancy.

Whatever the multi-tenancy model, the data architecture needs to ensure robust security, extensibility, and scalability in the data tier. For example, storing a particular customer's data in a separate database leads to the simplest design and development approach. Having data isolation is also the easiest and the quickest to both understand, and explain to your customers.



It is very tempting to offer tenant-specific customizations when each tenant's data is stored in separate databases. However, this is primarily done to separate data and associated operations, and not to arbitrarily allow dramatic changes to the database schema per tenant.

In this model, suitable metadata is maintained to link each database with the correct tenant. In addition, appropriate database security measures are implemented to prevent tenants from accessing other tenants' data. From an operations perspective, backups and restores are simpler for separate databases as they can be executed without impacting other customers. However, this approach can lead to higher infrastructure and licensing related costs.

Typically, you would offer this approach to your bigger customers who may be more willing to pay a premium to isolate their data. Larger enterprise customers prefer database isolation for higher security, or in some cases to comply with their security policies. On a related note, such customers may also demand more customization.



While architecting multi-tenanted applications, pay particular attention to the expected number of tenants, storage per tenant, expected number of concurrent users, regulatory and policy requirements, and so on. If any of these parameters are heavily skewed in favor of a particular tenant then it may be advisable to isolate their data.

For applications that have a few tables in their database schema, an approach that shares the database server instance across multiple tenants but has a separate database schema for each tenant can be used. This approach is relatively simple to implement and offers flexibility for custom tables to be defined per tenant. However, data restore for a particular tenant can impact other tenants hosted on the same database instance. This approach is often a preferred approach as it can reduce costs while separating out the data of each tenant.

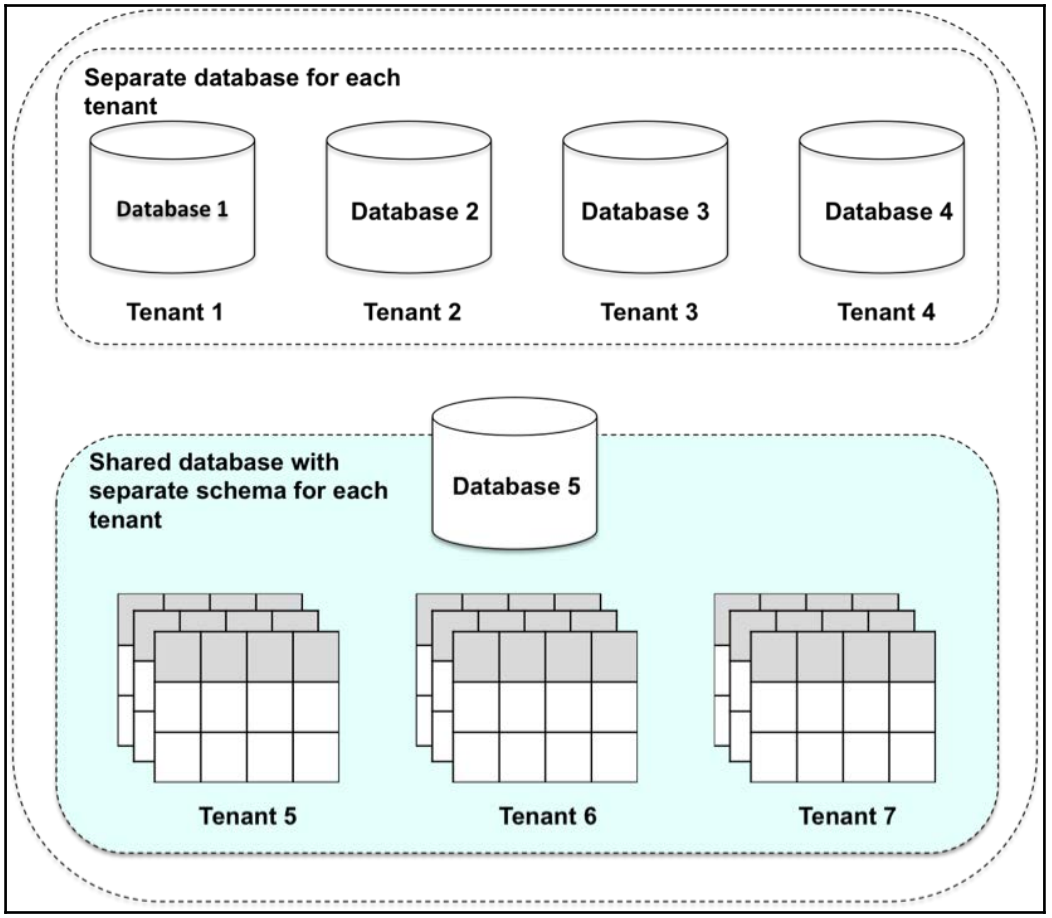
In a shared database, with a shared schema, approach, the costs are minimized but the complexity of the application is much greater. This model works well for cost-conscious customers. However, restoring a customer's data is complicated, as you will be restoring specific rows belonging to a specific tenant while other customers are using the system. Aside from possibly impacting other tenants using the shared database, it can significantly add complexity to scheduling such operations and communicating about them to all other tenants.

In cloud architectures, the main factors to consider while implementing multi-tenancy are data security, extensibility, and scalability.

Addressing data-at-rest security requirements

There are two levels of security to be considered; at the tenant level (typically, an organization) and at the end-user level, of a given tenant. In order to implement a security model you need to create a database access account at the tenant level. This account can specify (using ACLs) the database objects accessible to a specific tenant. Then at the application level, you will need to prevent users from accessing any data they are not entitled to. A security token service can be used to implement the access at the tenant level.

In the approaches that implement multi-tenancy by having either separate databases or separate schema per tenant, you can restrict access tat the database or the schema level for a particular tenant. The following diagram depicts a very common scenario where both these models are present in a single database server instance:



If the database tables are shared across tenants then you need to filter data access by each tenant. This is accomplished by having a column that stores a tenant ID per record (to clearly identify records that belong to a specific tenant). The figure in the next section shows a set of tables with the tenant id column. In such a schema, a typical SQL statement will contain a where-clause based on the tenant id being equal to the security id of the user account, namely an account belonging to the tenant.

Aside from database level security, organizational policies or regulatory requirements can mandate securing your data at rest. The options for implementing encryption to protect your data can range from fully automated solutions to manual ones to be implemented on the client side. There are several solutions available from the cloud service provider and third party vendors to implement these security models. This topic will be discussed in detail in `Chapter 6`, *Designing for and Implementing Security*.

Regardless of the approach, it is a good practice to encrypt sensitive data fields in your cloud database and storage. Encryption ensures that the data remains secure even if a non-authorized user accesses it. This is more critical for shared database/schema models. In many cases, encrypting a database column that is part of an index can lead to full table scans. Hence, try not to encrypt everything in your database as it can lead to poor performance. Therefore, it is important to carefully identify sensitive information fields in your database, and encrypt them more selectively. This will result in the right balance between security and performance.



It is a good idea to store a tenant id for all records in the database and encrypt sensitive data regardless of which approach you take for implementing data multi-tenancy. A customer willing to pay a premium for having a separate database might want to shift to a more economical shared model later. Having a tenant id and encryption already in place can simplify such a migration.

Addressing data extensibility requirements

Having a rigid database schema will not work for you across all your customers. Customers have their specific business rules and supporting data requirements. They will want to introduce their own customization to the database schema. However, ensure that you don't change your schema for a tenant to an extent that your product no longer fits into an SaaS model. But you do want to bake in sufficient flexibility and extensibility to handle the custom data requirements of your customers (without impacting subsequent product upgrades or patch releases).

One approach to achieving extensibility in the database schema is to pre-allocate a bunch of extra fields in your tables, which can then be used by your customers to implement their own business requirements. All these fields can be defined as string or varchar fields. You can also create an additional metadata table to further define a field label, data type, field length, and so on, for each of these fields on a per tenant basis. You can choose to create a metadata table per field or have a single metadata table for all the extra fields in the table. Alternatively, you can introduce an additional column for the table name to have a common table describing all custom fields (for each tenant) across all the tables in the schema.

This approach is depicted in the following figure. Fields 1 to 4 are defined as extra columns in the customer table. Further, the metadata table defines the field labels and data types.

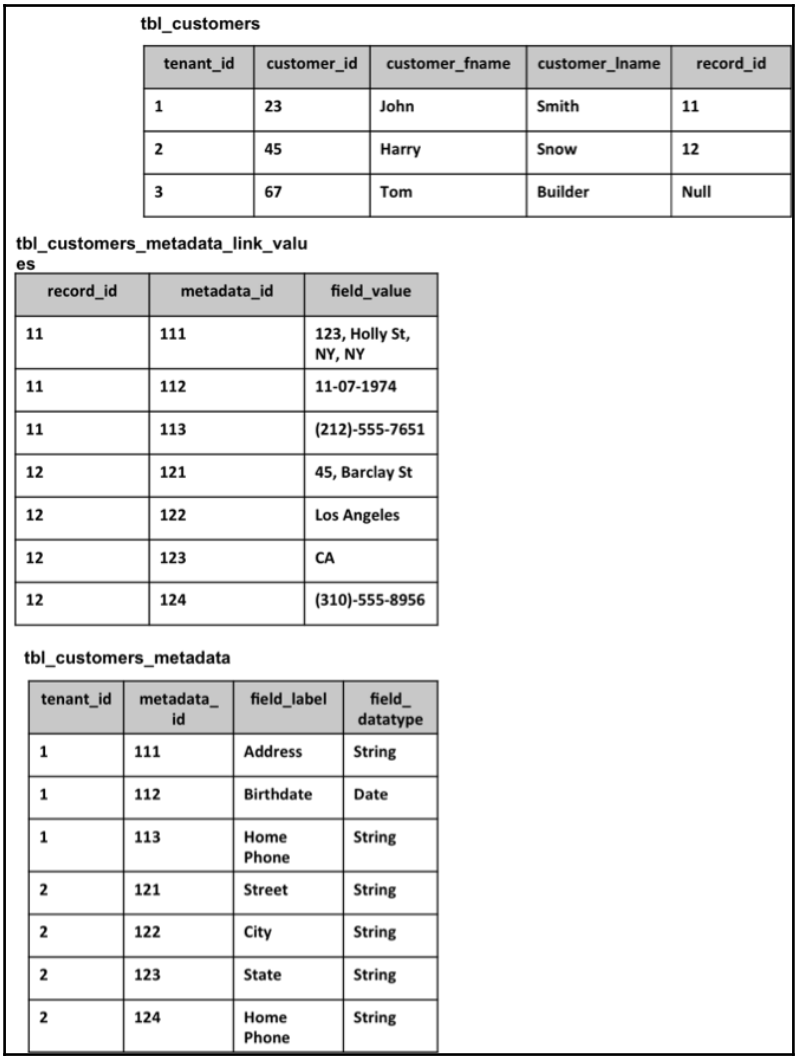
tbl_customers

tenant_id	customer_id	customer_fname	customer_lname	field1	field2	field3	field4
1	23	John	Smith	123, Holly St, NY, NY	11-07-1974	Null	(212)-555-7651
2	45	Harry	Snow	45, Barclay St	Los Angeles	CA	(310)-555-8956

tbl_customers_metadata

tenant_id	field1_label	field1_ datatype	field2_label	field2_ datatype	field3_label	field3_ datatype	field4_label	field4_ datatype
1	Address	String	Birthdate	Date	Null	Null	Home Phone	String
2	Street	String	City	String	State	String	Home Phone	String

A second approach takes a name-value pair approach, where you have a main data table that points to an intermediate table containing the value of the field and a pointer to a meta data table that contains the field label, data type, and so on, information. This approach cuts out potential waste in the first approach but is obviously more complicated to implement.



tbl_customers

tenant_id	customer_id	customer_fname	customer_lname	record_id
1	23	John	Smith	11
2	45	Harry	Snow	12
3	67	Tom	Builder	Null

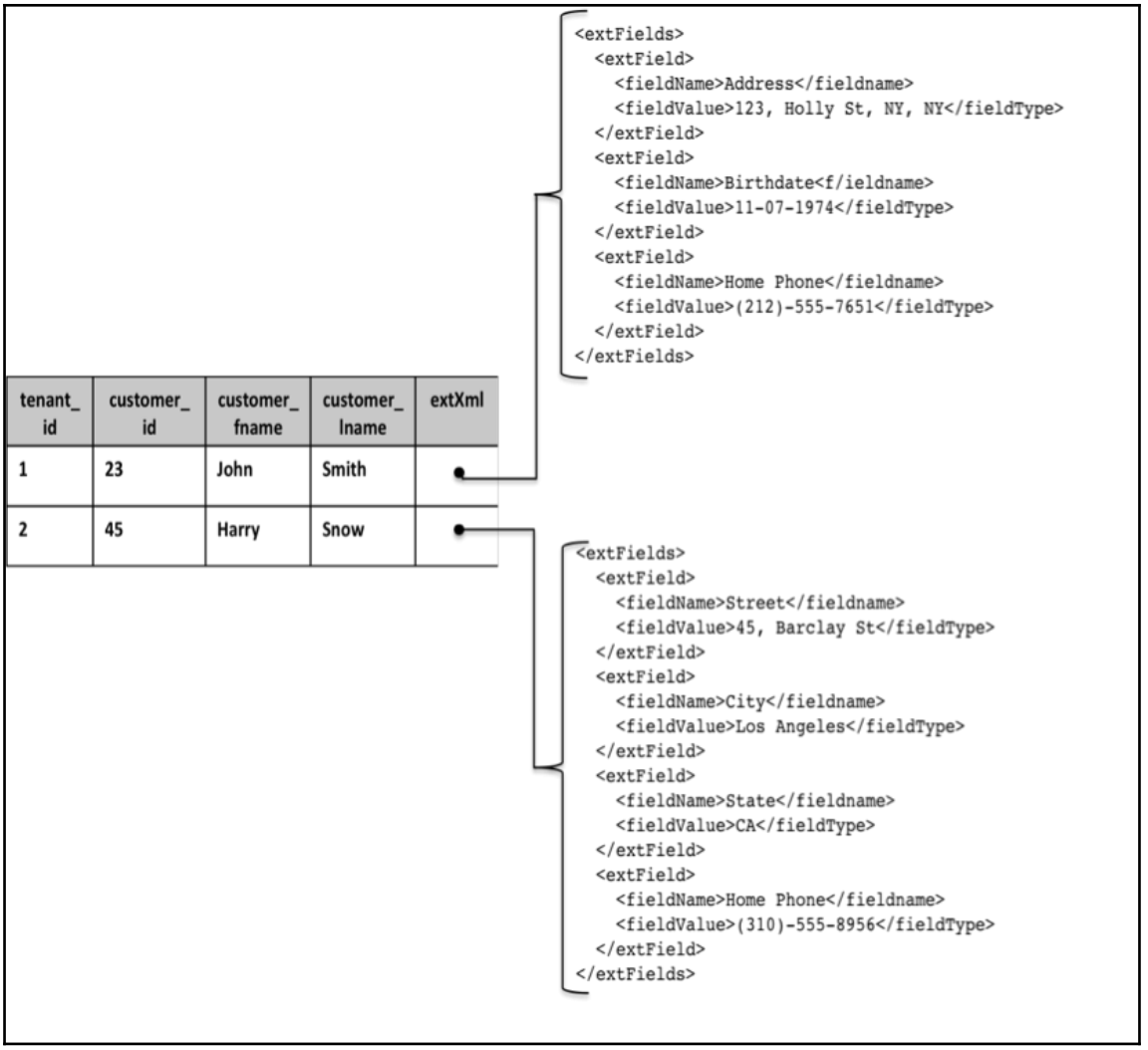
tbl_customers_metadata_link_values

record_id	metadata_id	field_value
11	111	123, Holly St, NY, NY
11	112	11-07-1974
11	113	(212)-555-7651
12	121	45, Barclay St
12	122	Los Angeles
12	123	CA
12	124	(310)-555-8956

tbl_customers_metadata

tenant_id	metadata_id	field_label	field_datatype
1	111	Address	String
1	112	Birthdate	Date
1	113	Home Phone	String
2	121	Street	String
2	122	City	String
2	123	State	String
2	124	Home Phone	String

A variation on these two approaches is to define an extra field per table and store all custom name-value pairs per tenant in an XML or JSON format.



A third approach is to add columns per tenant as required. This approach is more suitable in the separate database or separate schema per tenant models. However, this approach should generally be avoided as it leads to complexity in application code, that is, handling an arbitrary number of columns in a table per tenant. Further, it can lead to operational headaches during upgrades.



You will need to design your database schema carefully for providing custom extensions to your database schema as this can have a ripple effect on the application code and the user interface.

In addition to introducing a tenant id column in the database, if the application has web service interfaces then these services should also include the tenant id parameter in its request and/or response schema. To ensure a smooth transition between shared and isolated application instances, it is important to maintain tenant ids in the application tier. In addition, tenant aware business rules can be encoded in a business rules engine, and tenant specific workflows can be modeled in multi-tenanted workflow engine software using **Business Process Execution Language** (**BPEL**) process templates.

In cases where you end up creating a tenant-specific web service, you will need to design it in a manner that least impacts your other tenants. A mediation proxy service that contains routing rules can help in this case. This service can route the requests from a particular tenant's users (specified by the tenant id in the request) to the appropriate web service implemented for that tenant.

Similarly, the front end or the UI can also be configured for each tenant to provide a more customized look-and-feel (for example, CSS files per tenant), tenant specific logos, and color schemes. For differences in tenant UIs, portal servers can be used to serve up portlets, appropriately.

If different service levels need to be supported across tenants, then an instance of the application can be deployed on separate infrastructure for your higher-end customers. The isolation provided at the application layer (and the underlying infrastructure) helps avoid tenants impacting each other by consuming more CPU or memory resources than originally planned.

Logging also needs to be tenant aware (that is, use tenant id in your log record format). You can also use other resources such as queues, file directories, directory servers, caches, and so on, for each of your tenants. These can be done in a dedicated or separated out application stacks (per tenant). In all cases, make use of the tenant id filter for maximum flexibility.

Understanding cloud applications design principles

In this section, we will cover key guiding principles that are useful while designing cloud-based applications. More specifically, we will introduce designing for scale, automated infrastructure, failures, parallel processing, performance, and eventual consistency.

Designing for scale

Traditionally, designing for scale meant carefully sizing your infrastructure for peak usage and then adding a factor to handle variability in load. At some point, when you reached a certain threshold on CPU, memory, disk (capacity and throughput) or network bandwidth, you would repeat the exercise for handling increased loads and initiate a lengthy procurement and provisioning process. Depending on the application, this could mean a scale up (vertical scaling) with bigger machines or scale out (horizontal scaling) with more machines being deployed. Once deployed, the new capacity would be fixed (and run continuously) whether the additional capacity was being utilized fully or not.

In cloud applications, it is easy to scale—both vertically and horizontally. Additionally, the increase and the decrease in the number of nodes (in horizontal scalability) can be done automatically to improve resource utilization and manage costs better.

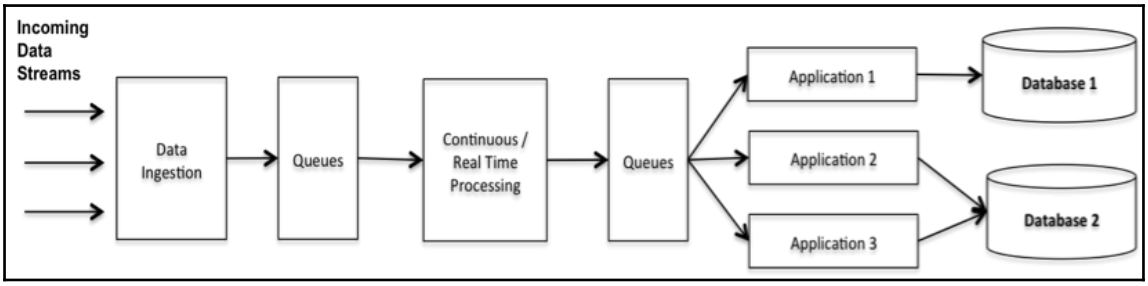
Typically, cloud applications are designed to be horizontally scalable. In most cases, the application services or the business tier is specifically designed to be stateless so that compute nodes can be added or deleted with no impact to the functioning of the application. If the application state is important then it can be stored externally using a caching or storage service. Depending on the application, things like session state can also be provided by the caller in each call, or be rehydrated from a data store.

Horizontal scaling in the data tier is usually achieved through **sharding**. Sharding splits a database across two or more databases to handle higher query or data volumes than what can be effectively handled by a single database node. In traditional application design, you would choose an appropriate sharding strategy and implement all the logic necessary, to route the read/write requests to the right shard. This results in increased code complexity. Instead, if you choose to use a PaaS cloud database service, the responsibility for scalability and availability is largely taken care of by the cloud provider.

An architecture comprising of loosely coupled components is a well-accepted approach and best practice. This is especially true while building highly scalable systems. Loose coupling allows you to distribute your components and scale them, independently.

The most commonly used design approaches to implement loose coupling is to introduce queues between major processing components in your architecture. Most PaaS cloud providers offer a queuing service that can be used to design for high concurrency and unusual spikes in load. In a high velocity data pipeline type application, the buffering capability of queues is leveraged to guard against data loss when a downstream processing component is unavailable, slow or has failed.

The following diagram shows a high capacity data processing pipeline. Notice that queues are placed strategically between various processing components to help match the impedance between the inflows of data versus processing components' speed.



Typically, the web tier writes messages or work requests to a queue. A component from the **services** tier then picks up this request from the queue and processes it. This ensures faster response times for end users as the queue-based asynchronous processing model does not block on responses.

In a traditional architecture, you may have used message queues with simple **enqueue** and **dequeue** operations to add processing requests and remove them for processing from the queues, subsequently. However, implementing queue-based architectures on the cloud is a little different. This is because your queue may be distributed across several nodes, internally, by the cloud service, your messages automatically replicated for you across several nodes, and also because one of these nodes may be unavailable when your request arrives or fails during the processing of your request.

In order to design more effectively, it is important to understand that:

- The message order is not guaranteed to be preserved between the enqueue and dequeue operations. If there is a requirement to strictly preserve this sequence then you need to include sequencing information as a part of the content of each message.

- It may so happen that one of the replicas of the message may not get deleted (due to a hardware failure or the unavailability of the node). Hence, there is a chance that the message or processing request would get processed twice. It is imperative to design your transactions to be idempotent in such circumstances.
- As the queue is distributed across several servers, it is also possible that no messages or not all messages are returned in any given polling request. The cloud queuing service is not guaranteed to check all the servers for messages against each polling request. However, a message not returned in a given polling request will be returned in a subsequent one.
- Due to the variability in the rate of incoming requests, a lot of polling requests (as described previously) need not return any requests for processing. For example, online orders in an online shopping site may show wide variability between daytime and night hours. The empty polling requests are wasteful in terms of resource usage and more importantly, incur unnecessary costs. One solution to reduce these costs is to implement the exponential back-off algorithm (that steadily increases the intervals between empty polling requests). But this approach has the down side of not processing requests soon after their arrival. A more effective approach is to implement long polling. With long polling the queuing service waits for a message to become available, and returns it if the message arrives within a configurable time period. Long polling for a queue can easily be enabled through an API or a UI interface.
- In a cloud queue service, it is important to differentiate between a dequeue and a delete operation. When a message is dequeued, it is not automatically deleted from the queue. This is done to guard against the possibility of failure in the message reaching the processing component (due to a connection or a hardware failure). Therefore, when a message is read off the queue and returned to a component, it is still maintained in the queue. However, it is rendered invisible for a period of time so that other components do not pick it up for processing. As soon as the queue service returns the message, a visibility timeout clock is started. This timeout value is usually configurable. What happens if your processing takes longer than the visibility timeout? In such an eventuality, it is a good practice to extend the time window through your code to avoid the message becoming visible again, and getting processed by another instance of your processing component.
- If your application requirements do not require each message to be processed immediately upon receipt, you can achieve greater efficiency and throughput in your processing by batching a number of requests and processing them together through a batch API.



As charges for using cloud queuing services are usually based on the number of requests, batching requests can reduce your bills as well.

- It is important to design and implement a handling strategy for messages that lead to fatal errors or exceptions in your code. These messages will repeatedly get processed until the default timeout set for how long a message should be retained in the queue. This is wasteful processing and leads to additional charges on your bill. Some queuing services provide a dead letter queue facility to park such messages for further review. However, ensure you place a message in the dead letter queue after a certain number of retries or dequeue the count.



The number of messages in your queue is also a good metric to use for auto scaling your processing tier.

- Depending on the number of different types of messages and their processing duration, it is a good practice to have separate queues for them. In addition, consider having a separate thread to process each queue instead of a single thread processing multiple queues.

Automating cloud infrastructure

During failures or spikes in load you do not want to be provisioning resources, identifying and deploying the right version of the application, configuring parameters (for example, database connection strings), and so on. Hence, you need to invest in creating ready-to-launch machine images, continuously monitor your system metrics to dynamically take action such as auto scaling, develop scripts for automated deployments, centrally store application configuration parameters, and boot new instances quickly by bootstrapping your instances, and so on.

It is possible to automate almost everything on the cloud platform via APIs and scripts, and you should attempt to do so. This includes typical operations, deployments, automatic recovery actions against alerts, scaling, and so on. For example, your cloud service may also provide an auto healing feature. You should leverage this feature to ensure failed/unhealthy instances are replaced and restarted with the original configurations.

Designing for failure

Assume all things will fail. Ensure you carefully review every aspect of your cloud architecture, and design for failure scenarios against each one of them. In particular, assume hardware will fail, cloud data center outages will happen, database failure or performance degradation will occur, expected volumes of transactions will be exceeded, and so on. In addition, in an auto-scaled environment, for example, nodes may be shutdown in response to load getting back to normal levels after a spike. Nodes may also be rebooted by the cloud platform. There can be unexpected application failures. In all these cases, the design goal should be to handle such error conditions gracefully, and minimize any impact to user experience.

There should be a strong preference to minimize human or manual intervention. Hence, it is better to implement strategies using services made available by the cloud platform to reduce the chances of failures or automate recovery from such failures.

Following are a list of key design principles that will help you handle failures in the cloud more effectively:

- Store no application state on your servers because if your server gets killed then you will not lose any application state. Sessions or logging records should never be stored to local filesystem.
- Logging should always be to a centralized location, for example, using a database or a third-party logging service. If you need to store information temporarily for subsequent processing then use the cloud platform's reliable queuing service. This is relevant not only in the case of server failures but also applicable in server scale out situations. During the scaling down process you don’t want to lose information by storing it on the local filesystem.
- Your log records should contain additional cloud-specific information to help the debugging process, for example, instance ID, region, availability zone, tenant ID, and so on. Centralized logging across multiple tenants (in a shared everything configuration) can get voluminous. Therefore, it helps to use tools for viewing, searching, and filtering log records.
- A request passes through numerous components (for example, network components) along its journey to the server side processing components. An error can occur anywhere or anytime during the life of the request. These errors may typically result in a server error (that is, a 5 xx series error). In such cases, it is normal for the application code to implement retry logic. The cloud provider's SDKs usually provide features that make implementing this retry logic simpler.



Remember to log your retry attempts. If you notice a high number of retry attempts then it's a good idea to review the sizing of your infrastructure. You will most likely need to provision additional resources to reduce error or failure rates, and the resultant retry attempts.

- The cloud platform may restrict the number of API requests you can issue in a given time period. Hence, in addition to the total number of retries, you need to ensure you do not exceed the allowed request rates by implementing delays between your retry attempts. This is typically implemented using an exponential back-off algorithm where you progressively introduce longer delays between your retry attempts.
- Avoid single-points-of-failure. Plan to distribute your services across multiple regions and zones (that is, different data centers in the same region). This will minimize the chances of an application outage due to failures in individual instances, an availability zone, or a region.

Sometimes running multiple instances is cost prohibitive for smaller organizations (very common for start-ups new to the cloud). If you want to run a single instance then ensure you still configure it for auto scaling. Set the minimum and maximum number of servers equal to one. This will ensure that in case your instance becomes unhealthy then the cloud service can replace it with a new instance within a few minutes of downtime.

In some cases, for example, highly interactive applications, it is best to just display a simple message to the end user to resubmit the transaction or refresh the screen (the resulting retry will likely succeed).

Designing for parallel processing

It is a lot easier to design for parallelization on the cloud platform. You need to use parallel designs throughout your architecture from data ingestion to its processing. So, use multithreading for parallelizing your cloud service requests, distribute load using load balancing, ensure multiple processing components or service endpoints are available via horizontal scaling, and so on.

Exploit both multithreading and multi-node processing. For example, using multiple concurrent threads for fetching objects from cloud data storage service is a lot faster than fetching them sequentially. In the pre-cloud or non-cloud environments, parallel processing across a large number of nodes was a difficult and expensive problem to solve. However, with the advent of cloud it has become very easy to provision a large number of compute instances within minutes. These instances can be provisioned, used and then released using APIs. In addition, frameworks such as Apache Spark and Hadoop have reduced the earlier complexity and expenses involved in building large-scale distributed applications.

Designing for performance

When an application is deployed to the cloud, latency can become a big issue. There is sufficient evidence that shows that latency leads to loss in business. It can also severely impact user adoption.

You will need to attack the latency issue through approaches that can improve the user experience by reducing the perceived and real latency. For example, some of the techniques you can use include rightsizing your infrastructure, using caching and placing your application and data closer to your end users.

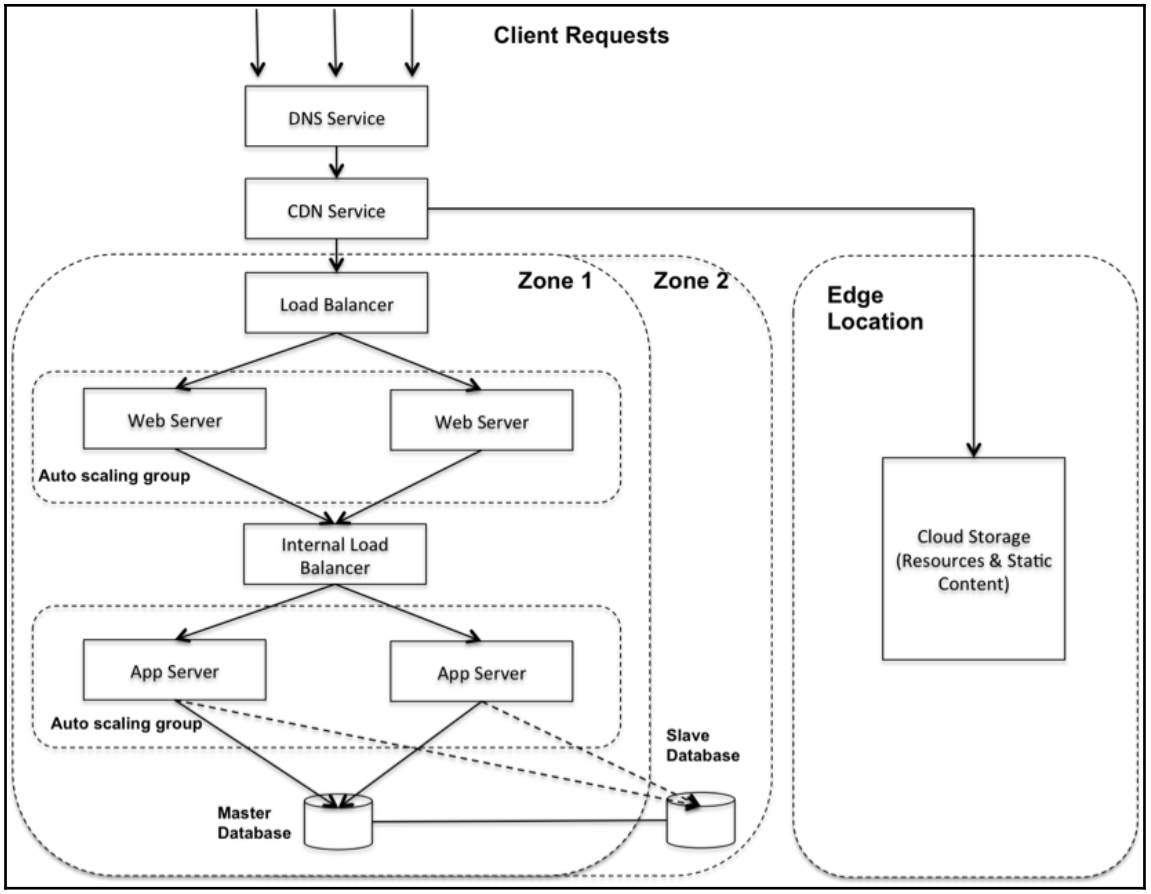
Perceived latency can be reduced by pre-fetching data that is likely to be used by the application, or caching frequently used pages/data. Additionally, you can design your pages in a manner that after they are loaded, the downloaded page doesn't need to traverse the network for most of the subsequent navigation. You can also use AJAX, or similar technology to reduce perceived latency of web pages loading.

Ensure that the data required by your processing components are located as close to each other as possible. Use caching and edge locations to distribute static data as close to your end users as possible. Performance oriented applications use in-memory application caches to improve scalability and performance by caching frequently accessed data. On the cloud, it is easy to create highly available caches and automatically scale them by using the appropriate caching service.

Most cloud providers maintain a distributed set of servers in multiple data centers around the globe. These servers are used to make it easy to use **Content Delivery Network** (**CDN**) to serve content to end users from locations closest to them. This service is made available to you by the cloud service provider through an easy-to-use web service interface. The distributed content could be HTML, CSS, PHP, or image files in regular web applications. CDNs can also be used for rich media and content sites with live streaming video.

The content is distributed to various edge locations, and is served to end users from points closest to them. This reduces latency while simultaneously improving the performance of your web application/site, significantly.

The following figure shows how a typical web application hosted on the cloud can leverage the CDN service to place content closer to the end user. When an end user requests content using the domain name, the CDN service determines the best edge location to serve that content. If the edge location does not have a copy of the content requested, then the CDN service pulls a copy from the origin server (for example, the web servers in **Zone 1**). The content is also cached at the edge location to service any future requests for the same content:



Designing for eventual consistency

Depending on the type of applications you have designed in the past, you may or may not have come across the concept of eventual consistency (unless you have worked extensively on distributed transactions oriented applications). However, it is fairly common in the cloud world. After a data update, if your application can tolerate a few seconds delay before the update is reflected across all replicas of the data then eventual consistency can lead to better scalability and performance.



Cloud platforms typically store multiple replicas of the data to ensure data durability. For example, the replica of a database table could be stored in several geographically distributed locations.

Normally, eventual consistency is the default behavior in a cloud data service. In case the application requires consistent reads at all times then some cloud data services provide the flexibility to specify strongly consistent reads. However, there are several cloud data services that support the eventually consistent option only.

Another approach used to improve scalability and performance beyond the capacity, that is CPU or I/O or both, of a single instance of your database, is to deploy one or more read replicas close to your end users. This is typically used for read-heavy applications. The read traffic can be routed to these replicas for reduced latencies. These replicas can also support resource heavy queries for online report generation or serve read-only requests, while your main database is down for maintenance or operations activities.

Note that changes to the source database are applied to the read replicas continuously, but there is a small lag involved. Hence, read replicas are considered to be eventually consistent.

Understanding emerging cloud-based application architectures

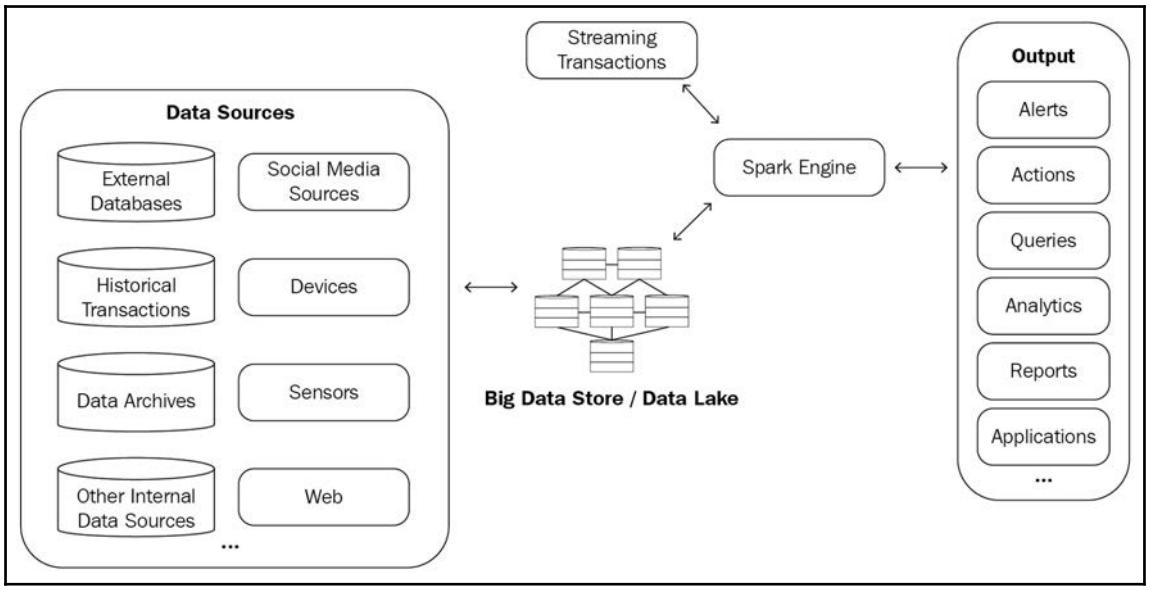
In this section, we will describe common architecture patterns and deployment of some of the main processing models being used for batch processing, streaming applications, and machine learning pipelines. The underlying architecture for these processing models are required to support ingesting very large volumes of various types of data arriving at high velocities at one end, while making the output data available for use by analytical tools, reporting and modeling software, at the other.

The software platforms supporting such applications have the necessary features and support the key mechanisms required to access data across a diverse set of data sources and formats, and prepare it for downstream applications, either as low-latency streaming data or high-throughput historical data stores. For example, **Apache Spark** is an emerging platform that leverages distributed storage and processing frameworks to support querying, reporting, analytics and intelligent applications at scale.



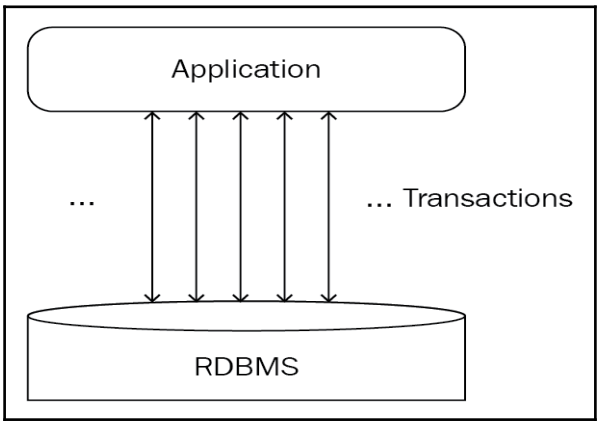
For more details on Apache Spark-based architectures, refer to *Learning Spark SQL, Aurobindo Sarkar, Packt Publishing*.

The following figure shows a high-level architecture that incorporates these requirements in typical Spark-based batch and streaming applications:

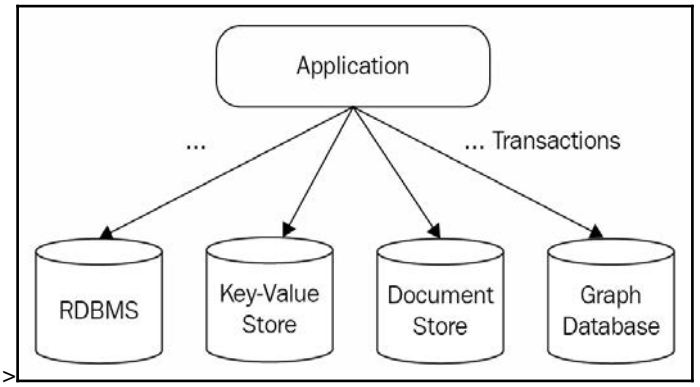


Understanding polyglot persistence

As organizations start employing big data and NoSQL-based solutions across a number of projects, a data layer comprising of RDBMSs alone is no longer the best solution for all the use cases in a modern enterprise application. The following figure illustrates a situation that is rapidly disappearing across the industry:



A more typical scenario comprising of multiple types of data stores is shown in the following figure. Applications today use several types of data stores that represent the best fit for a given set of use cases. Using multiple data storage technologies, chosen based upon the way data is being used by applications, is called polyglot persistence. For example, Apache Spark is an excellent enabler of this and other similar persistence strategies in the cloud or on-premise deployments:

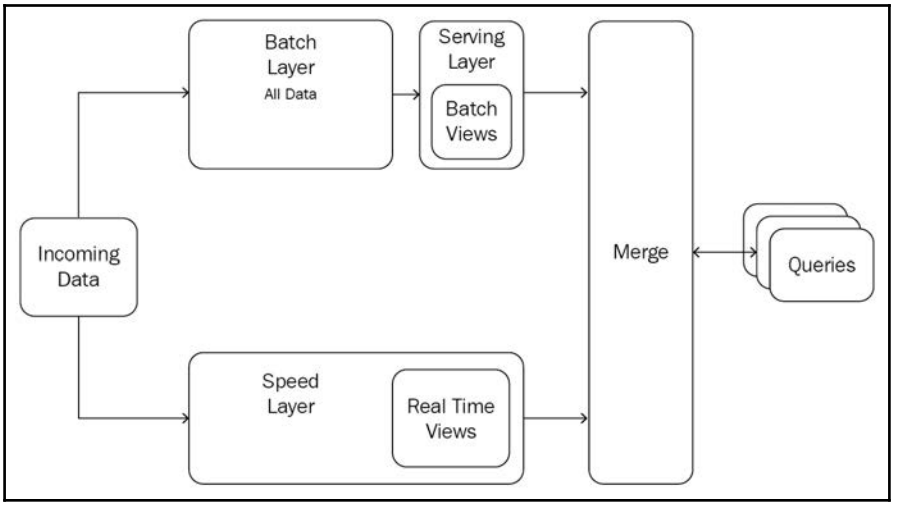


In the next section, we discuss the key concepts of batch and stream processing architectures.

Understanding Lambda architecture

The Lambda architectural pattern attempts to combine the best of both worlds—batch processing and stream processing. This pattern consists of several layers: **Batch Layer** (ingests and processes data on persistent storage such as HDFS and S3), **Speed Layer** (ingests and processes streaming data, that has not been processed by the batch layer yet), and the **Serving Layer** that can combine outputs from the batch and speed layers to present merged results. This is a very popular architecture in Spark-based cloud environments because it can support both batch and speed layer implementations with minimal code differences between the two.

The following figure depicts the Lambda architecture as a combination of the batch processing and stream processing:

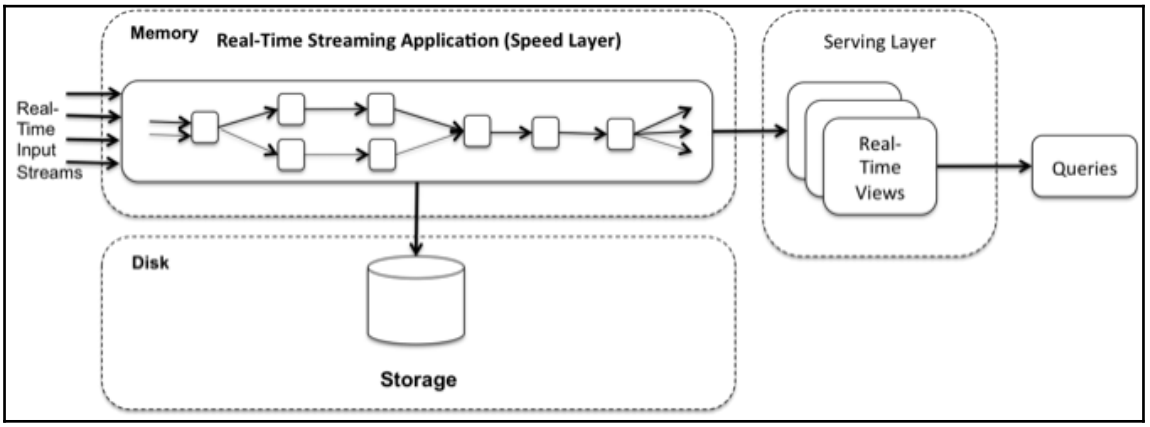


In the next section, we discuss a simpler architecture called Kappa architecture that dispenses with the batch layer entirely and works with stream processing in the speed layer only.

Understanding Kappa architecture

Kappa architecture is simpler than the Lambda pattern as it comprises of the speed and serving layers only. All the computations occur as stream processing and there are no batch recomputations done on the full dataset. Recomputations are only done to support changes and new requirements.

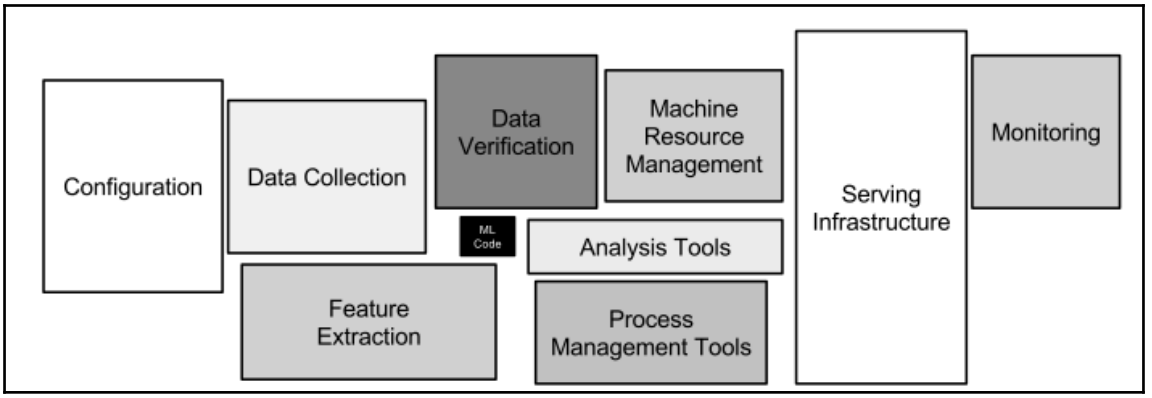
Typically, the incoming real-time data stream is processed in memory and is persisted in a database or HDFS, to support queries, as illustrated in the following figure:



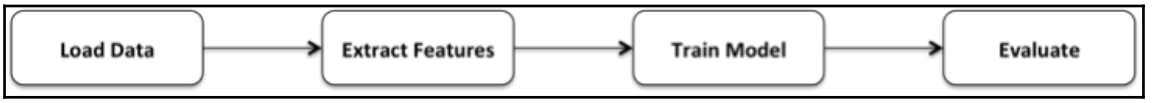
Kappa architecture can be realized using a queueing solution such as Apache Kafka or Kinesis. If the data retention times are bound to several days to weeks then Kafka could also be used to retain the data for the limited period of time.

Deploying cloud-based machine learning pipelines

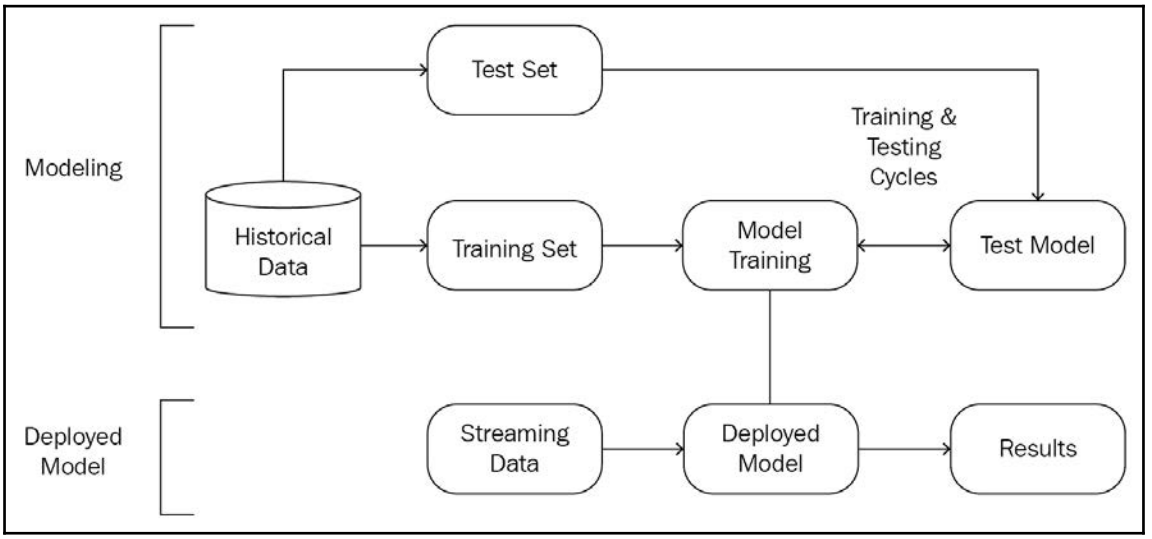
We observe that only a small fraction of real-world ML systems are composed of ML code (the small black box in the figure shown here). However, the infrastructure surrounding this ML code is vast and complex. There are many services offered by cloud providers to provide and manage the infrastructure required for modern machine learning applications:



The following figure illustrates a typical machine learning pipeline at a conceptual level. However, real-life ML pipelines are a lot more complicated with several models being trained, tuned, combined, and so on.



The following figure shows core elements of a typical machine learning application split into two parts—the modeling including model training and deployed model (used on streaming data to output the results):



Typically, the data scientists experiment or do their modeling work in Python and/or R. Their work is then re-implemented in Java/Scala before deployment in a production environment. The enterprise production environments often consist of web servers, application servers, databases, middleware, and so on. The conversion of prototypical models to production-ready models, typically results in additional design and development efforts that lead to delays in rolling out updated models. However, with cloud services and platforms now available, this effort is reduced substantially as we will see in Chapter 8, *Designing a Big Data Application*.

Deploying cloud-based machine learning models

The model scoring environments can be very diverse. For example, models may need to be deployed in web applications, portals, real-time and batch processing systems, as an API or a REST service, embedded in devices or in large legacy environments.

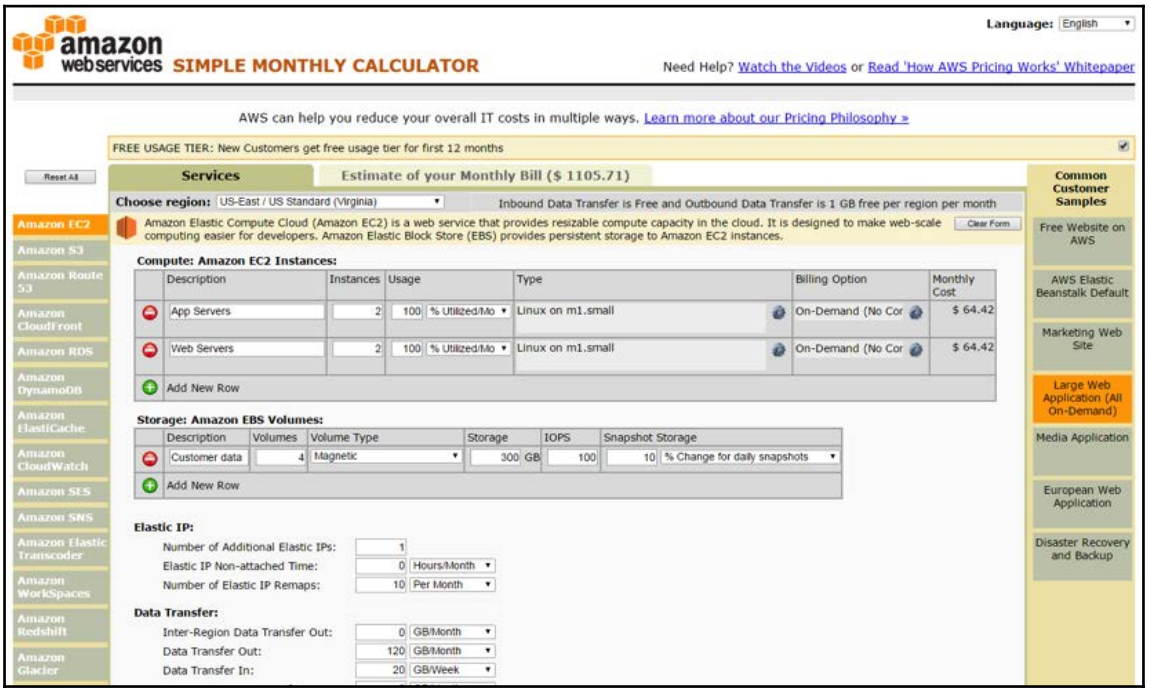
The technology stack can comprise Java Enterprise, C/C++, legacy mainframe environments, relational databases, and so on. Additionally, non-functional requirements and customer SLAs with respect to response times, throughput, availability and uptime can also vary widely. However, in all cases our cloud deployment process will need to support A/B testing, experimentation, model performance evaluation, and be agile and responsive to business needs.

Typically, practitioners use various methods to benchmark and phase-in new or updated models to avoid high-risk big bang production deployments. We will explore more on deploying such applications in *Chapter 10*, *Deploying a Big Data Application*.

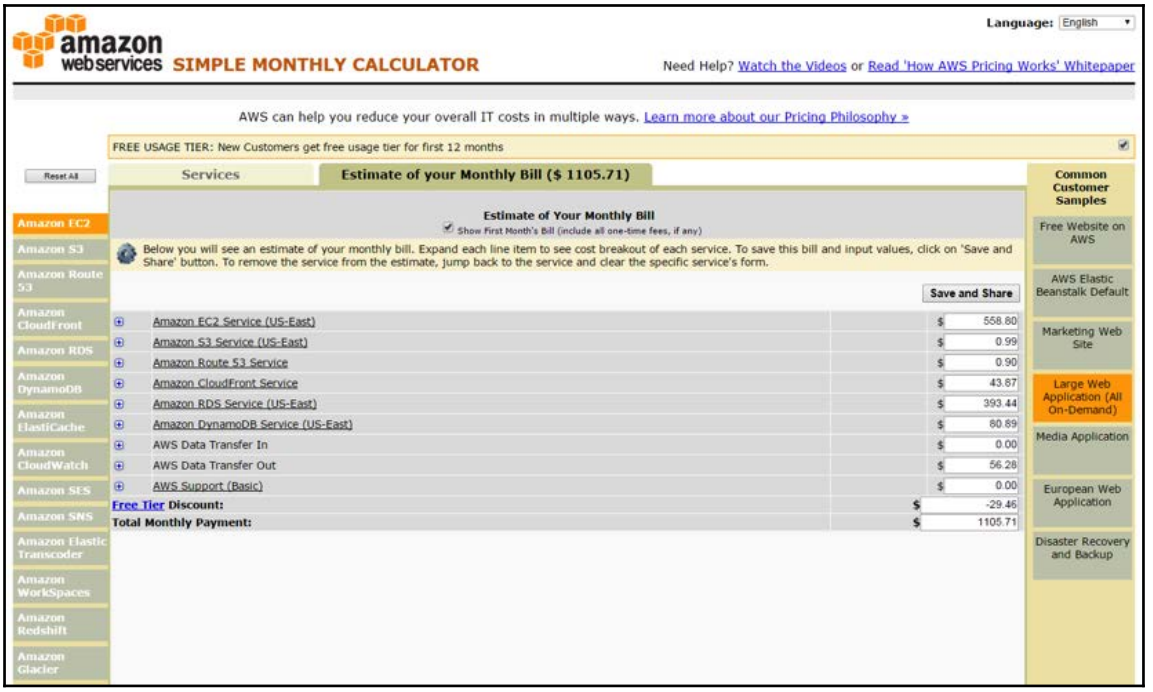
Estimating your cloud computing costs

Costs are central to designing for the cloud. Selecting the most appropriate options from a wide variety of tunable parameters available for each of the services can make this a challenging task. Typically, if you understand the costing for your compute nodes and database services well, then you would have largely accounted for a big chunk of your expected bill. Using an 80:20 principle can help you get to ballpark cost estimates, quickly. Typically, if you understand the costing for your compute nodes and database services well, then you would have largely accounted for a big chunk of your expected bill.

Most cloud service providers make online calculators available to arrive at the ballpark figures for your infrastructure. The following is a sample screenshot for provisioning AWS EC2 instances (compute nodes) in a calculator provided by Amazon. The left margin contains links to costing the various AWS services that you plan to provision for your application:



The following figure is a sample screenshot of the AWS calculator's monthly bill tab. This tab presents the total costs you can expect on a monthly basis. These calculators are typically very easy to use, and there is a lot of guidance and help available to select the appropriate options for each of the services:



The calculations and the totals obtained from these calculators is a good estimate, however it is a snapshot-in-time or a static estimate. You will need to create several of these to accurately reflect your costs through the product development life cycle. For example, you will provision development, QA/test, staging, and production environments at different times, and with different sizing and parameter values. In addition, you may choose to shutdown all development and QA/test environments at the end of each work day, or bring up the Staging environment only for load tests and a week prior to any production migrations.

Cloud-service providers present you with an itemized bill that includes the details of your resource usage. Compare the actual resource usage against your provisioned resources for identifying tuning opportunities. This can help lower your cloud environment costs.

It is very important to understand your cloud resource usage and the associated costs in your itemized bill. Track your bills closely for the first few months and at crucial times in your product development. These include whenever you spin up new environments, do load testing, run environments round-the-clock, provision a new service, or upgrade or increase the number of your compute instances. It is also important to give a heads up to finance or the leadership team when you expect the bills to show a spike or an uptick.

A typical e-commerce web application

In this section, we go through the specifications of a typical e-commerce website which we will develop and deploy on AWS infrastructure. We will leverage the AWS infrastructure to reduce project timeline and also show you the specific AWS code needed to support your non-functional requirements. However, this application is meant for illustrative purposes and is not a production grade application.

The codebase for this application will be in Java and the framework used will be Spring 4.x along with MySQL as the database. We will not delve into the detailed design or the specifications as it is not in the scope of the book, nor shall we develop all the functional use cases defined in the specifications. We will, however, dive deep into the non-functional specifications as they tend to leverage the cloud services more.

Suppose electronics retailer, A1 Sales, has decided to create an e-commerce site to boost their brand and revenues. A1 Sales has identified specific functional and non-functional requirements that are typical of any e-commerce web application. A1 Sales has made the decision not to invest in a data center and instead leverage public cloud infrastructure; hence the e-commerce web application needs to be ready for the cloud from day one.

The top level functional requirements identified are:

- The application shall allow users to browse and display detailed information of the selected products
- The application shall provide a shopping cart during online purchase
- The application shall allow users to add/remove products in the shopping cart before confirming a purchase
- The application shall allow a user to register and create his/her credentials
- The application shall authenticate user credentials before purchase of products
- The application shall enable users to enter the shipping address during payment process
- The application shall enable users to enter payment information during the payment process
- The application shall send an order confirmation to the user through email
- The application shall allow users to cancel an order
- The application shall allow the addition/deletion/updating of a selected product for an admin user

The non-functional requirements identified are:

- **Operational cost**: The architected solution for the e-commerce application should have a low monthly operational cost as nothing is free on the cloud. The solution, at a minimum, shall meet the minimum requirements for scalability, availability, fault tolerance, security, and replication and disaster recovery.
- **Scalability**: The cloud infrastructure shall scale the application up or down by adding/removing application nodes from the network depending on the load on the application.
- **Scalability**: The architected solution shall be designed in a loosely coupled and stateless manner which lends itself to scaling.
- **High availability**: The architected solution will be designed in a manner that avoids a single point failure in order to achieve high availability .
- **Fault tolerant**: The application shall be coded to handle cloud failures to a predefined limit.
- **Application security**: The application shall use encrypted channels for communications. All the confidential data shall be stored in an encrypted format. All the files at rest shall be stored in an encrypted format.
- **Cloud infrastructure security**: The cloud infrastructure shall be configured to close all the unnecessary network ports with the help of a firewall. All the compute instances on the cloud shall be secured with SSH keys.
- **Replication**: All the data should be replicated in real time to a secondary location to reduce the possibility of data loss.
- **Backups**: All the data from the databases shall be backed up on a daily basis.
- **Disaster recovery**: The architected solution shall be designed in a manner that supports recovery from an outage with minimal human intervention using automated scripts.
- **Design for failure**: The architected solution shall be designed for failure, in other words the application shall be designed, implemented and deployed for automated recovery from failure.
- The application shall be coded using open source software and open standards to prevent vendor lock-in and drive costs down.

Setting up your development environment

In this section, we show you how to download the source code from GitHub and run the A1 electronics e-commerce application. It is assumed the user has the following packages installed in their development environment:

- **Eclipse or Spring Tool Suite (STS)**: Download link STS: `http://spring.io/tools/sts` Eclipse: `https://eclipse.org/downloads/`
- **JDK 1.8**: Download link `http://www.oracle.com/technetwork/java/javase/downloads`
- **Maven 3**: Download link `http://maven.apache.org/download.cgi`
- **Git command line tools**: Download link `http://git-scm.com/downloads`
- **Eclipse with Maven plugin (m2e)**: m2e is installed by default when using the STS; you can install last M2Eclipse release by using the following update site from within Eclipse Help | Install New Software (`http://download.eclipse.org/technology/m2e/releases`)

The following instructions are for mac OS X but not limited to, they can also be used for Windows and Linux with minor or no modifications. It is assumed the readers have familiarity with the tools and are able to compile and run code from STS. This application is developed using Spring framework 4.x and Java 1.8.

Let's get started:

1. To begin with, create a folder `a1electronics` in your preferred workspace:

```
mkdir a1electronics
```

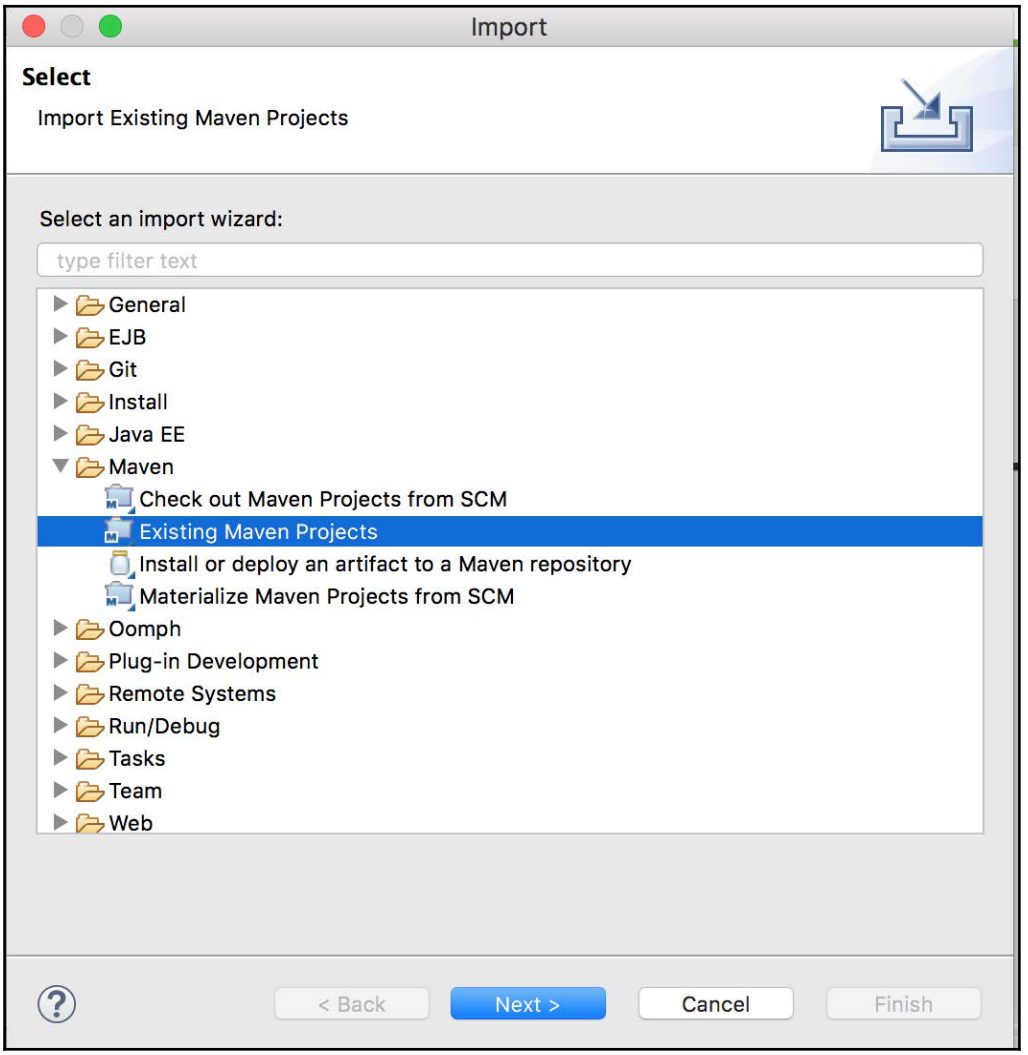
   Next, we need to download the source code from the GitHub repository.

2. Switch to the created folder `a1electronics` and clone the source code from the Git repository located at `https://github.com/a1electronics/ecommerce`.

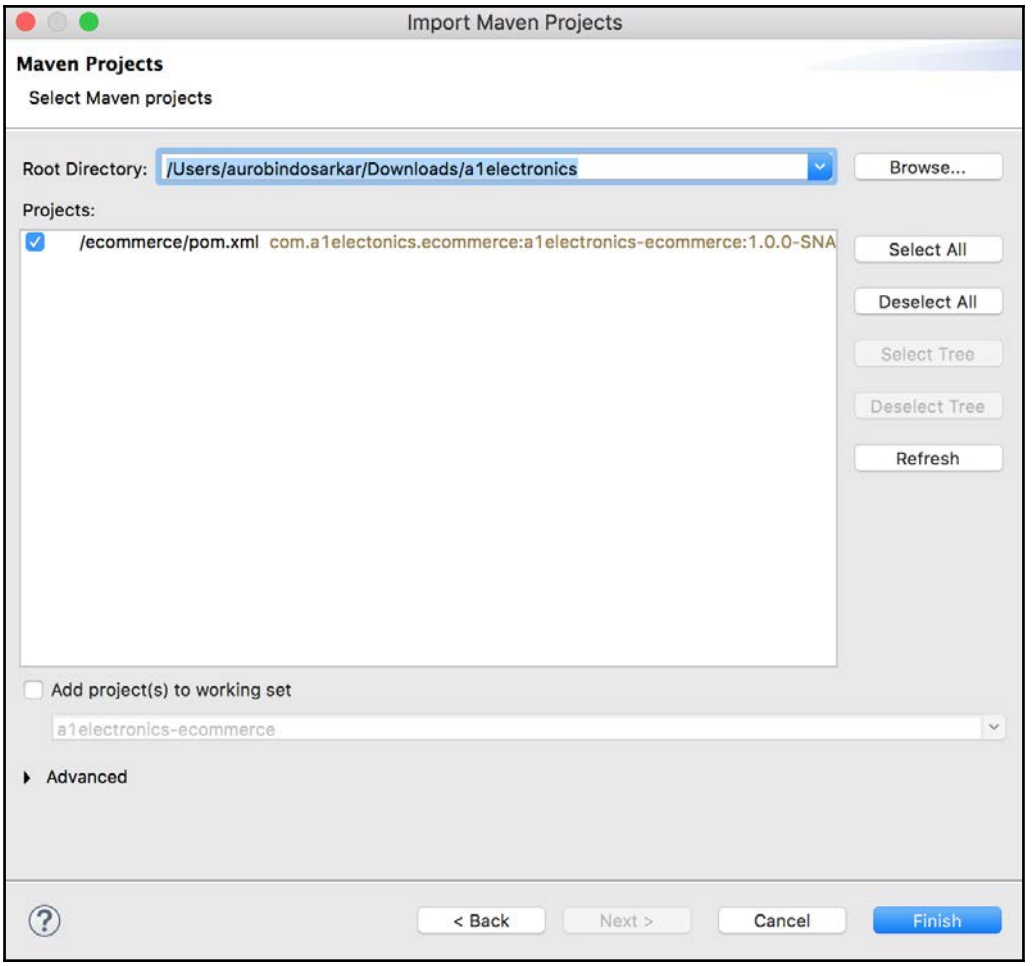
```
git clone https://github.com/a1electronics/ecommerce
```

3. Now you have the source code. The next step is to import the project in Eclipse or STS or, if you are one of the impatient types, you can run it directly from the command line.

4. To import the project into STS, you will need to go to menu **File** | **Import** and then select **Existing Maven Projects** as shown here and click on **Next**:



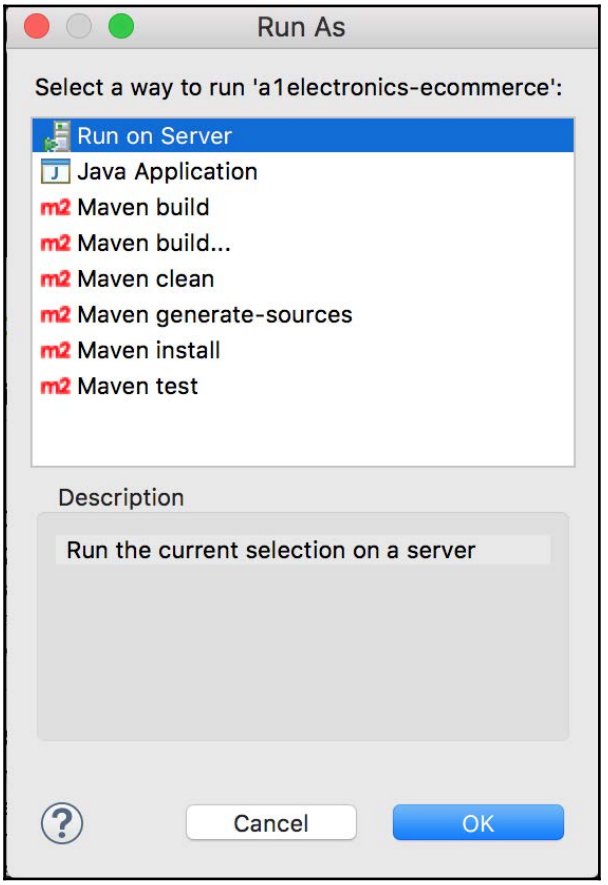
5. In the next step shown in the following figure, you will need to point to the folder where you checked out the application from the Git repository, as described in the earlier section. Clicking on **Finish** will successfully import your project into STS:



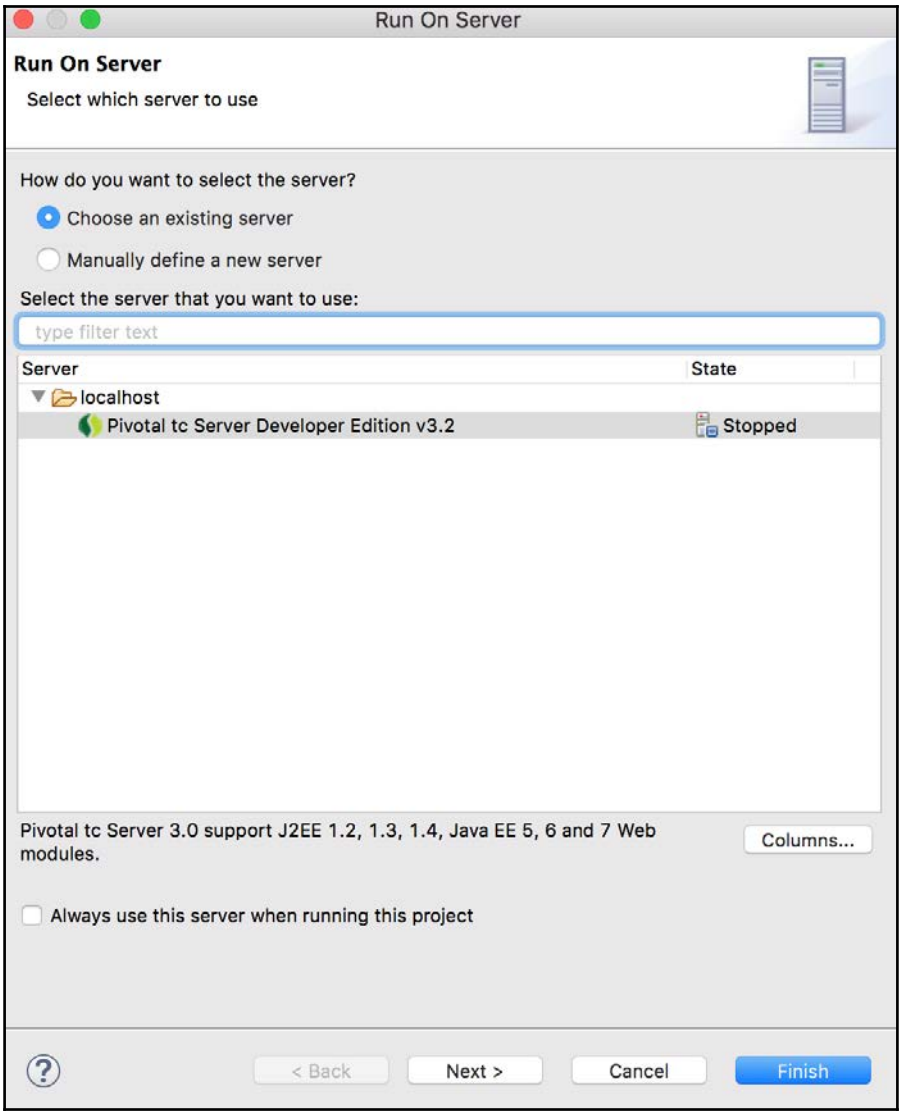
Running the application

Now in order to run our application, let's perform the following steps:

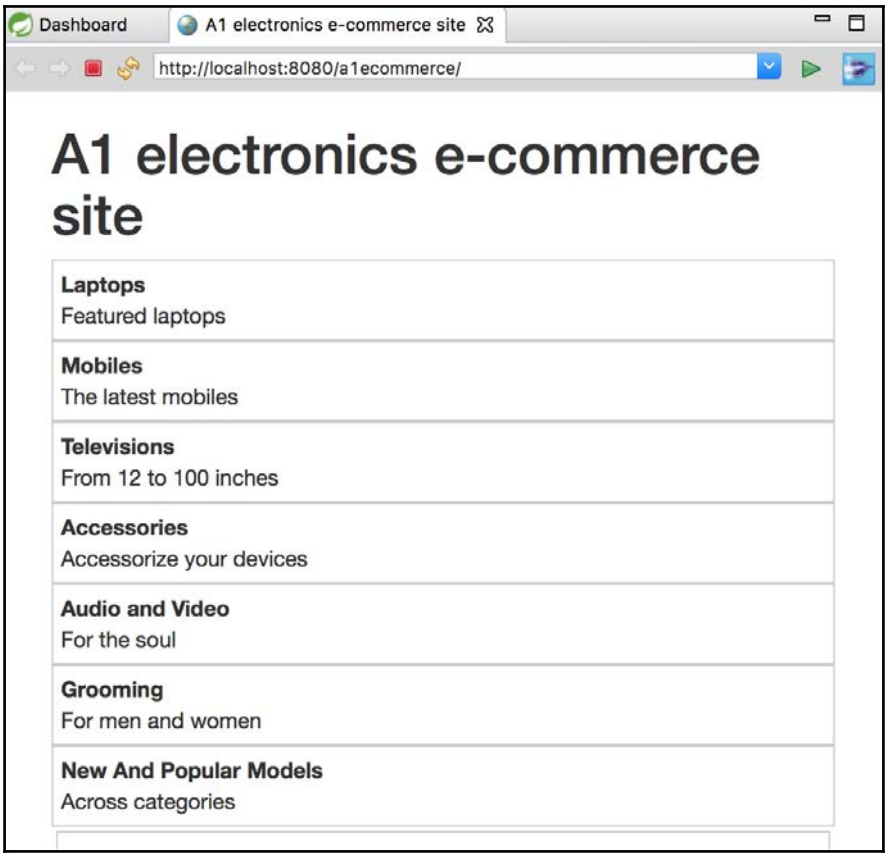
1. You can launch the imported project from within STS by selecting the project from the Package Explorer and selecting the menu option **Run**. This will open a pop-up window which is shown as follows:



2. Select the **Run On Server** option and click on **OK**. This will open another popup to select the installed web server from within STS, as shown in the following figure. Clicking on **Finish** will launch the A1 electronics e-commerce application from within the STS:



3. You should see the following screen show up in your STS environment:



Building a war file for deployment

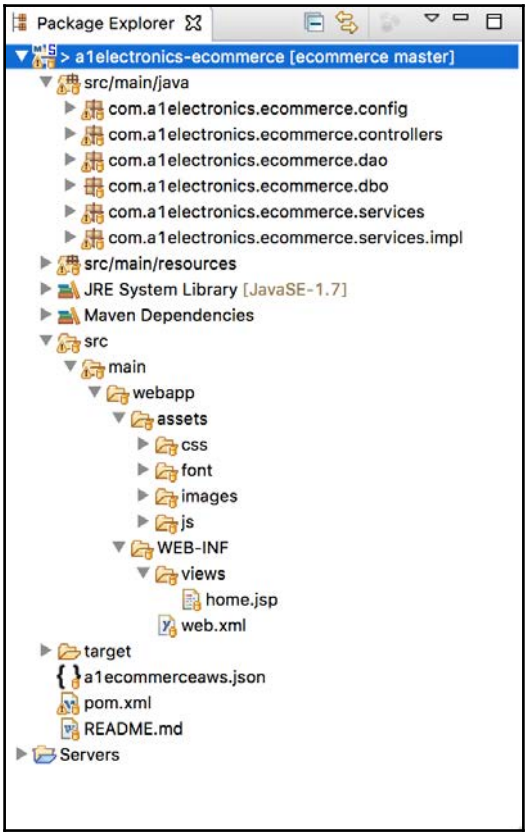
You have the option of creating a war file which can be deployed in any web server that supports servlet containers. This can be done via the command line from the root of A1 e-commerce project via a maven goal package.

```
mvn package
```

This will create a war file `a1ecommerce.war` in the folder called target, which is in the root of the A1 e-commerce project.

Application structure

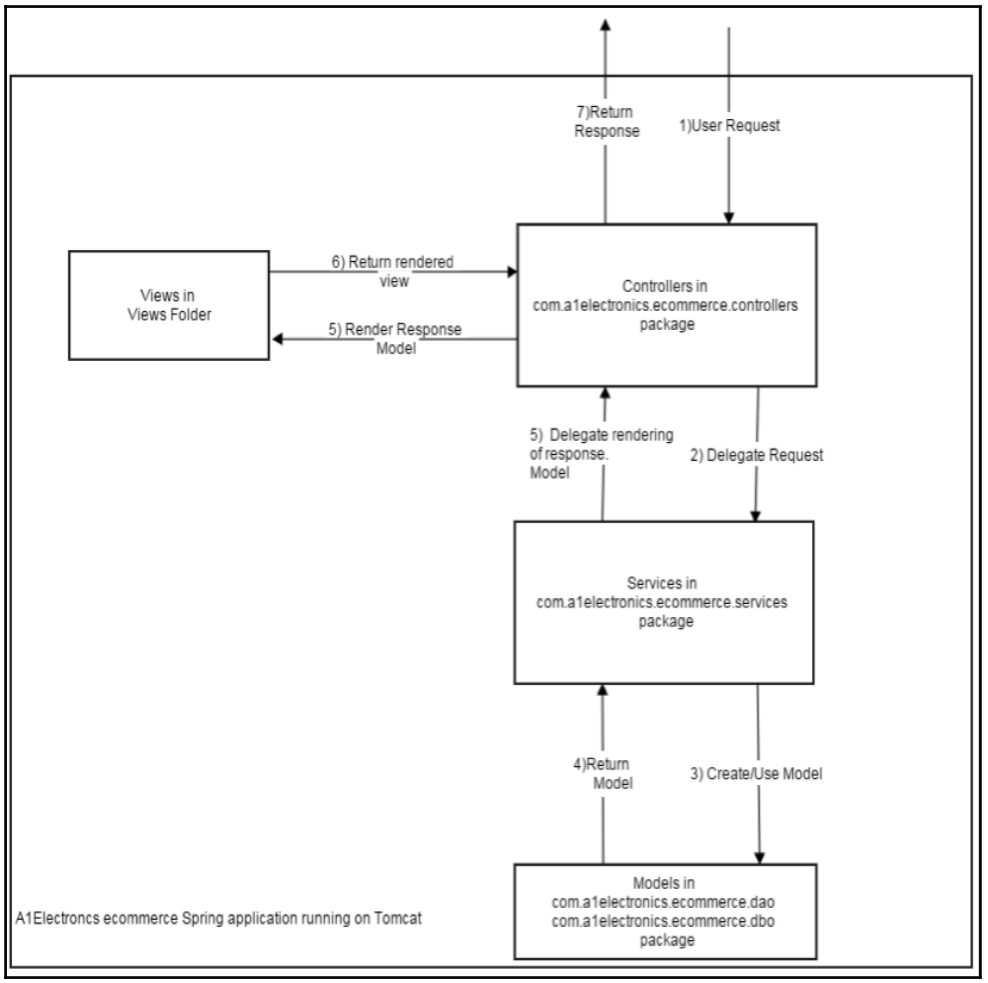
The A1 electronics e-commerce project uses a **Model-View-Controller** (**MVC**) architectural pattern. As the details of Spring and the MVC architecture are not in the scope of this book, only the relevant parts are explained here. This MVC architecture reflects in the structure of the code as shown in the following diagram:



- **Model**: The following packages are the Model part of MVC architecture:
  - `com.a1electronics.ecommerce.dbo`: All the entities from the database are mapped to Java objects along with their relationship
  - `com.a1electronics.ecommerce.dao`: A layer to access the objects in the dbo layer and has generic functions to `add`/`remove`/`delete`/`update` objects

- **View**: The `views` folder in the preceding figure is the View part of MVC architecture. These are JavaScript/HTML pages which are returned in response to users requests via the Controller. This is what gets rendered in the users' browser.
- **Controller**: The package, `com.a1electroncs.ecommerce.controllers`, is the Controller part of MVC architecture. This is where all the users' requests are accepted and, depending on the application logic, redirected to the correct service in the `com.a1electroncs.ecommerce.controllers` services layer package.

This MVC architecture is shown in the following figure:



Summary

In this chapter, we explained the differences in design and implementation of cloud-based applications. We reviewed some of the architectural best practices in the cloud context. More specifically, we described multitiered, loosely coupled, and service oriented scalable designs and their realizations on the cloud platform. We also went through the design consideration and implementation approaches to multi-tenancy, and explored some of the architectural patterns being used for streaming and machine learning applications. Finally, we created a simple application that we intend to expand and elaborate in the coming chapters to illustrate AWS concepts in detail.

After covering the cloud architectural principles in this chapter, we will get a lot more specific in our coverage of cloud computing in the next chapter. We will cover AWS specific cloud services, the AWS costing model, and provide guidance on your application development environments. In addition, we will have hands-on exposure to defining the cloud infrastructure required for the sample application created in this chapter.

3
Introducing AWS Components

This chapter will introduce you to some of the key AWS components and services. We will also cover strategies to lower your cloud infrastructure costs and their influence on your architectural decisions. Furthermore, this chapter will discuss the typical characteristics of AWS Cloud application development environments, including development, testing, staging, and production environments. Finally, we will walk you through the process of setting up the AWS infrastructure for our sample application.

In this chapter, we will cover the following topics:

- AWS components
- Managing costs on the AWS cloud
- Application development environments
- Setting up the AWS infrastructure

AWS components

AWS offers a wide variety of cloud-based services. There has been a continuously growing list of AWS services over the past few years with several of them in preview mode at any given time. These services include compute, storage, database, migration, networking and content delivery, security, analytics, AI, management tools, and many other products and services available. These ready-to-use AWS services and SDKs significantly simplify and accelerate the cloud application design, development, deployment, and maintenance-related activities.

In the following sections, we will describe some of the key AWS services used by developers and architects.

Amazon compute-related services

AWS offers several compute-related services to deploy and run your applications as virtual servers, containers, or even as code. For example, with EC2, you can provision and scale your compute infrastructure using a wide selection of instances that are optimized for various use cases. You can also run stateless or stateful applications packaged as Docker containers using Amazon EC2 Container Services (Amazon ECS), or run event-based stateless applications with the AWS Lambda service.

Amazon EC2

Amazon EC2 is a web service that provides compute capacity in the AWS cloud. You have several choices of instance types, operating systems, and software packages available. You will need to choose an appropriate mix of instance types based on your specific use cases. These instance types include general purpose, accelerated computing, compute-optimized, memory-optimized, and storage-optimized instances. For example, you would choose compute-optimized instances for your compute-intensive workloads and accelerated computing instances for GPU-based processing, typically, used for deep learning applications. In addition, each instance type includes one or more instance sizes to match the scalability requirements of your specific workloads.

Amazon EC2 allows you to configure the memory, CPU, instance storage, and your choice of operating system and applications. You can bundle the operating system, application software, and associated configuration settings into an Amazon Machine Image (AMI). Then, use it to provision or decommission multiple virtualized instances using web service calls. EC2 instances can be resized and the number of instances scaled, up or down, to match your requirements or demand.

Each AWS region comprises several Availability Zones (AZs) at distinct locations, connected by low latency networks. EC2 instances can be launched in one or more geographical locations or regions and also within one or more AZs belonging to a specific region.

Amazon Elastic Block Storage (EBS) volumes provide network-attached persistent storage to the EC2 instances. Elastic IP addresses allow you to allocate a static IP address and programmatically assign it to an instance. You can enable monitoring on EC2 instances using Amazon CloudWatch, and auto-scaling groups can be created, using the auto scaling feature, to automatically scale available capacity (based on the CloudWatch metrics).

Amazon EC2 container service

The **Amazon EC2 container service** is a cluster management and configuration management service. This service enables you to launch and stop container-enabled applications via API calls.

AWS Lambda

The **AWS Lambda** service supports executing your code in response to certain events within your application. Such events could include website clicks, image uploads, updates to certain data fields, document transformation, indexing, anomaly detection, errors detected in log files, sensitive or auditable events, unusual readings from sensors, and so on. You can also send notifications using SNS in response to these events.

Other AWS compute-related services include AWS Batch (to plan, schedule, and execute your batch jobs), AWS Elastic Beanstalk (for your application deployment and scalability requirements), Amazon EC2 Container Registry (to store, deploy, and manage Docker container images), and Amazon Lightsail (to launch and manage preconfigured virtual private servers).

Amazon storage-related services

Amazon's cloud storage services include object storage (Amazon S3), block storage (Amazon EBS), archival storage (Amazon Glacier), and file storage (Amazon Elastic File System) services. We will briefly describe some of these services in the following sections.

Amazon S3

Amazon S3 is a highly durable and distributed data store. Using a web services interface, you can store and retrieve large amounts of data as objects in buckets (containers). The stored objects are also accessible from the web via HTTP. It supports the implementation of stringent security and compliance policies on the stored data to ensure the security of data-at-rest. S3 is also an economical storage option for massive amounts of data typically used by analytics, IoT, and machine learning applications.

Amazon EBS

Amazon EBS is a highly available and durable persistent block-level storage volume for use with Amazon EC2 instances. You can configure EBS with SSD (general purpose or provisioned IOPS) or magnetic volumes. Each EBS volume is automatically replicated within its Availability Zone (AZ).

Amazon Glacier

Amazon Glacier is a low-cost storage service that is typically used for archival and backups. The retrieval time for data on Glacier, based on the option selected, varies from a few minutes to several hours.

Other AWS storage services include Amazon Storage Gateway (which enables integration between on-premise environment and AWS storage infrastructure) and AWS Import/Export service (which uses portable storage devices to enable movement of large amounts of data into and out of AWS cloud environment). Newer offerings from Amazon include petabyte to exabyte scale data transport services, such as AWS Snowball, AWS Snowball Edge, and AWS Snowmobile.

Amazon database-related services

There are several database-related services offered, including SQL and NoSQL databases, caching services, and a data warehouse service. In addition, an AWS database migration service is also available that can help migrate databases to AWS cloud with minimal downtime. Some of these services are described in the following sections.

Amazon Relational Database Service (RDS)

Amazon Relational Database Service (RDS) provides an easy way to set up, operate, and scale a relational database in the cloud. Database options available from AWS include MySQL, Oracle, SQL Server, PostgreSQL, and Amazon Aurora. With RDS, you can launch a DB instance and get access to a full-featured MySQL database and not worry about managing or administering it. Amazon RDS can significantly reduce effort on common database administration tasks, such as backups and patch management.

Launching a database requires you to select a database engine, license type, an instance class, and storage capacity. The RDS instances are preconfigured for the DB instance you choose. It is equally easy to monitor and scale your database instance (for both compute and storage capacities).

For production-grade availability, you can choose the Multi-AZ deployment option. In Multi-AZ deployments Amazon RDS automatically creates a primary DB instance and then synchronously replicates the data to another instance located in a different availability zone. In this setup, the RDS can automatically failover to the standby instance. For consistent IOPS, you can choose SSD storage and provision IOPS as per your requirements.

Amazon DynamoDB

Amazon DynamoDB is a NoSQL database service offered by AWS. It supports both document and key-value pairs data models and has a flexible schema. Integration with other AWS services, such as Amazon Elastic MapReduce (Amazon EMR) and Redshift, provide support for big data and BI applications, respectively. In addition, the integration with AWS Data Pipeline provides an efficient means of moving data into and out of DynamoDB.

To handle high data volumes, DynamoDB uses automatic partitioning. SSDs are used to provide high-throughput and low latencies at scale. The scaling can be implemented in a manner that matches the growth in the number of application requests. This service automatically replicates your data across three facilities within an Amazon region. Amazon DynamoDB is a fully managed service, which means that you don't have to sign-up for the full scope of database administration tasks.

Amazon Redshift

Amazon Redshift is a highly scalable data warehouse service offered by AWS. You can leverage your existing investments in BI tools because Redshift can work with them.

Amazon ElastiCache

If your application is read-intensive, then you can use AWS ElastiCache service to significantly boost the performance of your applications. ElastiCache supports Memcached and Redis in-memory caching solutions. AWS ElastiCache supports higher reliability through automatic detection and replacement of failed nodes and automatic patch management and enables monitoring through integration with Amazon CloudWatch. ElastiCache can be scaled-up/scaled-down in response to your application load.

Amazon messaging-related services

Cloud-based messaging services supports messaging to both external and internal users or systems. These services are key to ensuring scalable solution architectures, supporting monitoring-related alerts and messages, and implementing customer interaction and engagement business applications. We will introduce these services in the following sections.

Amazon SQS

Amazon Simple Queue Service (Amazon SQS) is a reliable, highly scalable, hosted distributed queue to store messages as they travel between computers and application components.

Amazon SNS

Amazon Simple Notification Service (SNS) provides a simple way to notify applications or people from an AWS cloud application. It uses the publish-subscribe protocol.

Amazon SES

Amazon Simple Email Service (SES) is a cloud-based email sending and receiving service. You can use Amazon's SMTP interface or integrate SES directly into your applications using AWS SDKs.

Amazon Pinpoint

Amazon Pinpoint can be used to collect information on your customers' devices and track usage information of your applications. It requires AWS mobile SDK to be integrated into your applications. This service is typically used to develop contextual customer engagement and interaction applications that trigger some action based on customer's behavior/usage.

Amazon networking and content delivery services

AWS networking and content delivery services enable isolation of your cloud infrastructure and help in scaling capacity and connecting your networks. These services are introduced in the following sections.

Amazon VPC (Virtual Private Cloud)

You can define a virtual network using Amazon Virtual Private Cloud (Amazon VPC). You can select IP address ranges, create subnets, and configure route tables and network gateways. For example, for a three-tier web application, you can create a public subnet for the web servers and a private subnet for the application and database servers. Amazon VPC allows you to extend your corporate network into a private cloud contained within AWS. The IPSec tunnel mode enables you to create a secure connection between a gateway in your data center and a gateway in AWS cloud.

Amazon Elastic Load Balancing

Elastic Load Balancing automatically distributes incoming application traffic across Amazon EC2 instances, containers, and IP addresses within and across a Availability Zones. Elastic Load Balancing offers three types of load balancers:

- Application Load Balancer (for HTTP/S traffic)
- Network Load Balancer (for TCP traffic)
- Classic Load Balancer (for applications built within EC2-Classic VPC).

Amazon Route 53

Amazon Route 53 is a highly scalable DNS service that allows you to manage your DNS records by creating a Hosted Zone for every domain you would like to manage.

Amazon CloudFront

The **Amazon CloudFront** service is a web CDN service for low-latency content delivery (static or streaming content). For example, copies of S3 objects can be distributed and cached at multiple edge locations around the world by creating a distribution network using Amazon CloudFront service.

AWS Direct Connect

AWS Direct Connect helps you to establish private connectivity between your data center, or office network, to the AWS cloud. This service can help you achieve higher bandwidth and consistent network performance.

Amazon management tools

Amazon management tools are used for provisioning, configuration management, monitoring, and enforcing security and compliance policies. We will introduce some of these services in the following sections.

AWS CloudFormation

The **AWS CloudFormation** service helps in creating and managing a collection of related AWS resources. We can create templates to describe the AWS resources and any associated dependencies or runtime parameters required to run your application. In addition to provisioning AWS resources, CloudFormation can also be used to update them in an orderly and predictable manner.

Amazon CloudWatch

CloudWatch is a monitoring service for your AWS resources. It enables you to retrieve monitoring data, set alarms, troubleshoot problems, and take actions based on the issues in your cloud environment.

AWS CloudTrail

AWS CloudTrail service helps with governance, compliance, operational auditing, and risk auditing of your AWS account. You can log and monitor account activity related to actions across your AWS infrastructure. CloudTrail provides a history of your AWS account activity, including actions taken through the AWS Management Console, AWS SDKs, command line tools, and other AWS services.

There are several other AWS management tools that can help you in the provisioning, configuration management, monitoring and performance, governance and compliance, and resource optimization of your cloud resources and environment.



For more details on AWS management tools, refer to `https://aws.amazon.com/`.

Amazon security, identity, and compliance services

Given the concerns enterprises have regarding the security of their data, Amazon has focused a lot on ensuring security of it's customers' data in the cloud. Aside from providing physical security and obtaining various industry-recognized security certifications, AWS offers a whole host of security and compliance services. We will introduce some of these in the following sections.

AWS Identity and Access Management (IAM)

AWS Identity and Access Management (**IAM**) enables you to control access to AWS services and resources. You can create users and groups with unique security credentials and manage permissions for each of these users. You can also define IAM roles so that your application can securely make API calls without creating and distributing your AWS credentials. IAM is natively integrated into AWS Services.

AWS Directory Service

AWS Directory Service enables you to use your corporate credentials to access AWS resources. You can use your existing directory service or set up a directory on the cloud for this purpose.

Amazon Certificate Manager

AWS Certificate Manager is a service that lets you easily provision, manage, and deploy Secure Sockets Layer/Transport Layer Security (SSL/TLS) certificates for use with AWS services. In addition, AWS Certificate Manager handles certificate renewals sparing you the headache of keeping track of them.

AWS Key Management Service

AWS Key Management Service (**KMS**) is a managed service that makes it easy for you to create and control the encryption keys used to encrypt your data. It is also integrated with AWS CloudTrail to provide you with logs of all key usage to help meet your regulatory and compliance needs. Also, it can help you implement key rotation policies with reasonable ease.

AWS WAF

AWS WAF is a web application firewall that helps protect your web applications from incidents that could affect application availability, compromise security, or consume excessive resources. AWS WAF gives you control over which traffic to allow or block to your web applications by defining appropriate rules.

Other security and compliance products from Amazon include AWS Artifact (for compliance reporting), AWS CloudHSM (for key storage and management), Amazon Inspector (for automatic security assessments), AWS Organizations (for policy-based management of multiple AWS accounts), and AWS Shield (for DDoS protection).

Amazon analytics-related services

AWS provides a set of services that can help implement highly scalable analytics applications quickly. These services include data streaming services, data pipelines, big data frameworks, and data preparation and loading tools. Some of these services are introduced in the following sections.

Amazon EMR

Amazon Elastic MapReduce (EMR) provides a hosted Hadoop framework running on Amazon EC2 and Amazon S3 that allows you to create customized MapReduce jobs.

Amazon Kinesis

The **Amazon Kinesis** service is designed for real-time streaming data ingestion and processing. Typical use cases for Amazon Kinesis include IoT applications.

Other Amazon Analytics-related services include Amazon Athena (a serverless query service for running SQL queries on S3 data), Amazon Elasticsearch service, Amazon QuickSight (BI tool for business analytics), AWS Glue (for ETL operations), and AWS Data Pipeline (for data workflow orchestration).

Amazon machine learning/AI-related services

New services in the areas of machine learning, deep learning, and AI are rapidly being made available by Amazon to support the latest industry trends in enterprise applications. These services include frameworks (Apache MXNet, TensorFlow, Caffe, Theano, Torch, Keras, and others), API-driven services (Amazon Lex, Amazon Polly, and Amazon Rekognition) to incorporate intelligence in your applications, and machine learning platforms (Apache Spark, visualization tools, and wizards for integrating various ML algorithms). Also, Amazon provides deep learning AMIs (with GPUs) for supporting deep learning applications development.

Some of these emerging services are introduced in the following sections.

Amazon Machine Learning

Amazon Machine Learning provides developers with visualization tools and wizards to help create ML models without having to learn complex ML algorithms or manage the underlying infrastructure. It allows you to start small and then scale, as your application grows. Typically, after the modeling phase is completed, Amazon Machine Learning makes it easier to obtain the results using simple APIs. These APIs assist in serving these results in real time. You can also use Amazon Batch for processing predictions based on large batches of data.

Apache Spark is an open source, distributed processing system that is increasingly becoming the platform of choice for many emerging applications, including streaming analytics, machine learning, and graph applications. You can quickly create managed Apache Spark clusters and use auto scaling to increase or decrease the size of your clusters.

Other Amazon AI-related services

Amazon Lex provides features such as **automatic speech recognition** (**ASR**) and **natural language understanding** (**NLU**) to help build applications such as chatbots. Amazon Polly is a service that turns text into lifelike speech that enables you to build speech-enabled products. Amazon Rekognition is a service that allows you to add image analysis functionality in your applications.

AWS services supports frameworks, such as TensorFlow and MXNet. Several organizations have started running their production applications with TensorFlow and Apache MXNet on AWS, and this trend is likely to continue to become a lot more popular.

Finally, from an infrastructure's perspective, Amazon EC2 P2 instances provide powerful Nvidia GPUs to significantly reduce the training time of your models. You can also use the Deep Learning CloudFormation templates to create P2 clusters using Amazon Deep Learning AMIs for your large-scale model training requirements.

Other Amazon services

AWS provides many other services for development, deployment, cloud migration, and management of your applications. Detailed documentation for these services is available on the AWS website.



For more details on AWS products and services, refer to `https://aws.amazon.com/`.

Aside from all the services provided by Amazon, there are many software products and services offered by third-party vendors through the Amazon Marketplace. Depending on your application requirements, you can choose to integrate these services into your applications instead of building them yourself.

Managing costs on AWS cloud

It is important to understand the AWS costing model and parameters so that you are able to track and manage your expenses better. However, there are many different options available and trade-offs involved in terms of infrastructure, services, and their associated costs. Hence, it is key to understand the business requirements and set costs-related objectives to guide your decision-making.

Setting costs-related objectives

There can be different perspectives on objectives depending on who you talk to in an organization. These perspectives are not necessarily conflicting with each other, but it is important to understand them well to approach your decision-making whether from a business, architecture, or operation's perspective. For example, the business perspective could be to pay as little as possible for whatever is used. Architectural goals may direct you toward avoiding waste as much as possible while achieving a more scalable and robust architecture for your applications. Similarly, operational goals may want you to focus on reducing the number of custom-developed components to be managed while using AWS services where possible to the maximum extent possible, thereby, minimizing the time and effort spent toward managing and maintaining the infrastructure and enabling more time to be spent on the business of the organization and/or to innovate.

Optimizing costs on the cloud

There are broadly three areas for cost optimizations on the cloud: operational, infrastructural, and architectural optimizations. It is important to note that costs should not focus on infrastructure alone. You should include code changes in your deliberations because sometimes, it makes sense to focus on improving the code rather than infrastructure alone. There are many architectural decisions and trade-offs to be made to achieve the best results from a cost saving's perspective. The good part is that on cloud, you can test these decisions and trade-offs immediately to decide, if your decisions make sense.

Costs are a big motivation to use cloud infrastructure, and AWS provides many different ways of saving on your AWS bills. However, it is up to you to take advantage of all the saving opportunities available. As a simple guideline start with minimal infrastructure and iterate from there to optimize your infrastructure costs.

Aside from architecting for scale, availability, security, and functionality—in the cloud you also need to architect for economy or costs. Cloud enables making infrastructural and AWS service usage changes more easily toward achieving lower transactional and/or operational costs. Most often pursuing these cost-cutting measures can lead to a leaner, more robust architecture.

The infrastructure setup process in the pre-cloud era consisted of plan-build-run steps where mistakes were often expensive and hard to correct. With the advent of cloud computing, this process has become cyclical where we iterate quickly through architect-build-monitor steps. Cloud infrastructure is dynamically allocated and de-allocated, so we do not have to be 100% right, in all our infrastructure design decisions, the first time. We can iteratively improve our architecture while meeting our infrastructure cost objectives.

The typical stages you would go through to manage your costs per transaction starts with calculating the costs by hand (using the Cost Explorer). In the next stage, you instrument your application to collect the data with respect to your transaction volumes and calculate the transactional costs from the corresponding billing data, periodically. Finally, you would do real-time transaction volumes monitoring and corresponding costs to drive further optimizations.

Typically, a substantial part of your bill comprises costs of EC2 compute instances, database instances (especially, if you are using Provisioned IOPS), and the usage of any specialized application services (Amazon EMR, Amazon Redshift, and so on). However, storage costs can also be significant for big data and machine learning applications that operate on vast amounts data. There are several strategies that can result in substantial savings, and most of these are relatively easy to implement.

In this section, we will focus on several strategies that will help you cut your cloud bills. We will expose areas of opportunity where significant savings can be achieved over the next few sections. These savings can vary anywhere from 30 to 80% of your current costs. These optimizations also include some quick wins that can be achieved with no changes to your application code.

Strategies to lower AWS costs

On the cloud, we are in a different world now where it is a lot easier to evolve architectures as you go along. Typically, running lean architectures are not only easy to evolve and operate but also helps you optimize your costs. We will explore some techniques and best practices to lower your AWS bills in the following sections. Interestingly, note that a simple do-nothing strategy would have also led to savings because over a period of time, Amazon has been able to leverage economies of scale and pass on additional savings to their customers. In fact, they have reduced prices over 60 times since 2006, and their prices continue to be revised downwards for various services on an on-going basis.

Monitoring and analyzing costs

AWS platform provides a set of tools to help monitor and analyze your costs. These include the AWS TCO Calculator, the simple monthly calculator (described in `Chapter 2`, *Designing Cloud Applications – An Architect's Perspective*), AWS billing console, which shows you an itemized bill, and AWS Cost Explorer, which gives you costs trends information across different time periods.

EC2, S3, CloudFront, and so on offer volume pricing tiers at lower price points based on usage. If the payer account combines usage from all the subaccounts, then you can get these discounts automatically using consolidated billing (with no additional effort involved).

The TCO calculator can be used to compare on-premise versus on-cloud costs. The AWS simple monthly calculator is a useful tool to select the various AWS services and options to compute your costs. AWS Trusted Advisor provides an automated way to save money. The tool comes free with business and enterprise support subscriptions. It scans your AWS infrastructure and identifies targets for further savings by generating a cost optimization report. It will tell you about idle instances that you may want to shut down and also the amounts you will be able to save by switching to reserved instances.

You can use the AWS billing console to drill down deeper into the AWS bill to see an itemized list of components and their costs. This can help us identify optimization targets. You can also set AWS billing alerts, which send you automatic notifications when your bill hits some preset threshold. These thresholds can also be used for auto scaling where you can shut down instances, automatically, if your bill reaches a certain level. You can also enable detailed billing to break down costs by hour, day, or month; or by each account. AWS will publish these reports as CSV files and store them to a S3 bucket.

There are several third-party open source (for example, Netflix ICE) and commercial tools (for example, Cloudability). These tools provide cost and usage reporting, information related to accounts, and comparison across time periods and underutilized instances.

Choosing the right EC2 Instance

The EC2 instances you choose are directly dependent on your application characteristics. Ensure that you have a good understanding of these characteristics, for example, is the application CPU-intensive, or is it more IO bound? What are the typical workloads? Is there variability in demand over a period of time? Are there any special events when the demand is unusually high? What are the average resource requirements for good user experience?

Based on the application characteristics shortlist a few instance types available from AWS. EC2 types include several families of related instances available in sizes ranging from micro to extra large. These classes include general purpose, compute-optimized, memory-optimized, storage-optimized, and accelerated computing instances.

You should then do a few tests to analyze the performance of the shortlisted instances against increasing loads. It is a good idea to understand the upper limit of these instances in terms of number of concurrent users or throughput they can support.

For example, let's assume you want to select EC2 instances for your web servers. These web servers proxy API calls to the application servers, that is, handle CPU-intensive traffic and support heavy payloads. Based on these requirements, let's say that you shortlist two instance types – one a CPU-optimized (say, c3.xlarge) and a general purpose (m3.xlarge) instance type. Typically, you should choose a general purpose and a special purpose instance types for comparison purposes. In order to conduct the performance analysis, create a set of test cases to simulate the expected scenarios in your application. Monitor the CPU utilization for these instances at different loads (say 1,000, 2,000, and 3,000 users). Increase the load to a point where you max out on the CPU. It is very likely that you will hit max CPU utilization at different loads for each of the chosen instances.



For the latest details on instance types, use cases and pricing, refer to `https://aws.amazon.com/ec2/instance-types/`.

In cases where you want to test at very high loads, you should contact Amazon before conducting the load test to have your load balancer "pre-warmed". They can configure the load balancer to have the appropriate level of capacity based on the expected traffic during your load tests. It is also important to load test from multiple distributed agents to ensure your IP address is not flagged. At this stage, you should provision multiple smaller instances (from the same families) that match the xlarge instance's compute power and conduct the same load tests. This is done to check whether we can achieve the same performance, at a higher level of resiliency, using multiple smaller instances in place of a bigger instance.

Instance selection is not only about the instance size or type alone but also about the available network bandwidth. Hence, you should also compare the results of your network bandwidth assessment for each of your instance types. If your application requires increased network bandwidth turn on the enhanced networking option, which is available on certain instance types, for example, enhanced networking option is available on compute-optimized C3, C4, D2, I2, and R3 instances, but not on general purpose instances.



For more details on options for enhanced networking, refer to `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/enhanced-networking.html`.

Compare the costs against different throughput levels. It is possible that the general purpose instance type costs more than the compute optimized instance type for your application's expected workload.

Availability and costs of instance types differ by region. Hence, your EC2 instance type and size decision will need to take into consideration the availability of instance types and then strike the right balance in terms of performance, resiliency, and costs.

Typically, in start-ups, the development and test environments are provisioned in the most economical region using minimum sizes of general purpose EC2 instances to minimize development infrastructure costs.

Finally, revisit and rightsize your EC2 instances choices every 3 to 6 months as instance families and workloads can change. New instance types that are more appropriate for your workloads can reduce your overall costs. You can use the Amazon Trusted Advisor for cost optimization hints for low network/CPU utilization, unused or low utilization instances, and so on. For example, if you observe a 15-20% CPU maximum utilization on an instance, then it is a trivial change to lower instance. If this step leads to an improvement in CPU utilization to 60-70%, then you would saved on costs with sufficient headroom for growth and/or auto scaling.

Choosing the right distribution of instances across AZs can also make a significant difference to your overall costs while meeting your availability requirements. For example, if 12 instances are required in order to maintain HA, then instead of splitting 24 instances across two AZs, we split them across three AZs containing 6 instances each so that the required 12 are available even when an AZ is down. This might sound a little obvious but make yourself well aware of various facilities offered by AWS to drive additional savings.

Turn-off unused instances

Typically, there is an inordinate amount of focus on production spends but a lot of that spend (in some cases, more than production) is on dev and test. It is surprising how many times you find unused instances adding to your bills. This usually happens when someone provisions an instance to conduct an experiment or check out a new feature and then fails to delete the instance, when done. It also happens a lot during testing when the test environment is, carelessly, left running through the weekend, or after the testing activity has been completed. It is important to check your bills and usage data to minimize such costs. Tag your instances with environment name, owner's name, and so on to identify the instances and the primary owner or cost center quickly.

Instances are disposable on the cloud, and they can be switched on and off, easily, so ensure that you switch off your dev, test, and training instances after office hours and through the holidays and weekends. There are several ways to achieve this including scripts, starting up based on time or on request, scheduled auto scaling groups, and so on. On cloud, instances are billed for usage; hence, you do not need to keep them on when they are not being used. You can easily save 30-40% or higher on your bills this way.

Furthermore, you can leverage AWS Lambda to get rid of idle time on servers. AWS Lambda comes with automatic provisioning and scaling, and there is no need to manage any infrastructure. Typically, if your server utilization is less than 40%, then consider using AWS Lambda instead.

Using Auto Scaling

Auto Scaling scale your compute instances to the extent required otherwise scale down, automatically. Auto Scaling aligns your deployed infrastructure to the demand at any given point in time. You can define launch configurations for your EC2 instances and then set up appropriate auto scaling groups for them. This helps automate the process of saving money by turning off unused instances during scale down. You can set parameters, such as the minimum and maximum number of instances, to meet your functional and nonfunctional requirements while controlling your overall costs.

It can take a few minutes for your new instances to come online during a scale-up. So, make sure that you account for this lag while establishing your thresholds. Do not set the threshold too high in production (for example, at 90% CPU utilization) because there is a high likelihood that your existing instances will hit 100% utilization before the new instances have spun up. It is a good practice to set the production CPU utilization thresholds to be between 60-70% to give you sufficient headroom. At the same time, in order to guard against inadvertent scale up, due to a random spike, you should also specify a duration of say 2 or 5 minutes at the threshold CPU utilization before the scale-up process kicks in. As EC2 instances are charged by the hour, do not rush to scale down immediately after you have scaled up (against an immediate dip in utilization below the threshold). You can set a longer duration say 10-20 minutes at the reduced CPU utilization threshold before scaling down.

You can also set thresholds for network and memory utilization based on profiling your application or working with an initial best guess and iteratively adjusting it to arrive at the right threshold values. However, avoid setting multiple scaling triggers per auto scaling group because this increases the chance of conflict in your triggers. This could lead to a situation where you are scaling up based on one trigger while scaling down due to another. You should also specify a cooling down period upon a scale down.

If you have implemented a multi-AZ architecture, then scale up and scale down should be in increments of two instances at a time. This helps keep your AZs balanced with equal numbers of instances in each.

Sometimes, massive scaling is required in response to certain planned or scheduled events. Special events such as a big sales event or flash sales events on popular e-commerce sites, or during sporting events and elections reporting on news sites can lead to disruptions due to the huge spikes in resource usage and demand during these events. In such cases, it may be a better approach to over-provision instances for the sharp increase in traffic rather than relying on auto scaling alone. After the event is over, the instances can be scaled down, appropriately.

You can do schedule-based scaling where you can match scaling with the workload at different times during the day and/or weekends. This approach can also be used to scale down development and test environments during off-peak hours.

After you have architected your application environment, the next step is to monitor it. Monitoring is important because it helps you validate your architectural decisions. If your focus is both costs and usage, then you need to monitor them closely, as they are both necessary to identify targets for further optimizations. Tag your instances with identifying information with respect to the environment, owner, cost center, and so on for reporting purposes. You also need to establish various monitoring thresholds and alerts. Analyze this information frequently to iterate on your architecture for further savings.

Using reserved instances

Reserved instances can help save 70% or higher on your instance costs (versus using on-demand instances). You have the flexibility to pay all, part or nothing as upfront fee, and they are available for 1 year or 3 years' duration at much lower hourly rates. You can also modify or sell your reserved instances, if your requirements change, subsequently. Typical breakeven on these instances vary between 5 months and 7 months depending on the duration of the contract.

Typically, you would run your systems for 3 to 4 months to understand your workloads, and tune and optimize your instances types, OS, tenancy, and so on before switching to Reserved Instances. As soon as you know your instance type and size, try to swap your on-demand instances with reserved instance.

Reserved instances are flexible, for example, they can be moved between AZs, and their sizes can be modified. In fact, you don't have to choose exactly where you want to run your reserved instances—this essentially decouples capacity reservation from cost optimization. You can also have convertible reserved instances (available for 3 years only) that allow you to change instance families/type to leverage the latest offerings from AWS. The effective rate of 3-year convertible reserved instances is better than 1-year standard reserved instances. So, even if you run them for a year and a half to two years, it is fine.

As production instances are typically required to run 24x7x365 in a reliable manner, Reserved Instances are a good fit for enterprise applications (in production). For dev/test environments (and in startups), you might want to experiment with and spend more time evaluating spot instances because spot prices can be a fraction of the regular on-demand prices.

Using spot instances

Instances used for experimentation, learning, and in highly price-sensitive situations, using spot instances can be the most cost-effective option. These instances offer incredible value for the right workloads and are commonly used for dev/test, and embarrassingly parallel workloads. AWS carries extra capacity in terms of unused instances, and they sell these instances on a spot market. The pricing is dynamic and based on supply/demand. If you check the price history of spot instances, you will note that they do not vary too much. Hence, you can balance price and availability with the right combination of On-Demand, Reserved, and Spot instances.

You can set the maximum price you want to pay for an instance, and that price can be much lower than the regular on-demand price (the price difference can sometimes be as high as 90%). If capacity is available, then Amazon will fulfill that request. However, your instance is terminated (with 2 minutes notice), if the spot price is higher than your price. If your application is architected against failures, then such terminations should not impact the running of your application.

Availability and costs can vary between different AZs. When the demand goes up, the price can go even higher than on-demand instances. To guard against this situation, you need to set your price carefully and start your instances in another AZ, if the price in your current AZ goes higher than your set price. For example, you can set your spot price to be greater than the market rate and less than the on-demand prices to always get the market rate, at or under your bid price.

Spot instances give you an opportunity to name your own price, and they can potentially save you 80-90% on your instance costs. But understand and plan for the risks associated with using them. You can leverage auto scaling to reduce your overall risks. For example, you can create two auto scaling groups: one with on-demand instances and the other with Spot instances. As Spot instances are not always guaranteed, you can set a CloudWatch alarm on the number of Spot instances in the group and auto scale the on-demand group, if the number of spot instances is below the threshold you set. The spot instances fleet saves you a lot of money automatically; while the fleet of on-demand instances scales up to compensate for any terminated spot instances. This way, you can have the best of both worlds—lower costs while maintaining a HA architecture. You can use 2 minutes available during the termination of spot instances for running scripts to write out any data, log files, and so on.

Spot instances are a great choice for stateless web/app server fleets, Amazon EMR, continuous integration, high-performance computing (HPC), grid computing, media rendering/transcoding, and so on use cases. You can also run batch jobs using AWS batch service that leverages spot instances.

The Spot Bid Advisor can tell you—how likely it is that you will lose your instances against other bids (for a given instance type and number of instances). If the likelihood is low over the past 1 month, then you are likely to keep your instances.

So far, we have primarily focused on cost savings related to EC2 instances. However, Amazon S3 offers additional opportunities to cut storage costs.

Using Amazon S3 storage

Amazon S3 offers a range of storage classes designed for different use cases. The Amazon S3 storage classes are listed here:

- **Amazon S3 Standard**: For general purpose storage of frequently accessed data
- **Amazon S3 Standard - Infrequent Access**: For long-lived, but less frequently accessed data
- **Amazon Glacier**: For long-term archive

Taking advantage of Amazon S3 - Infrequent Access (IA) instead of Standard S3 can lead to immediate storage costs savings up to 30% with no code changes. You will need to understand your storage usage patterns and then configure policies to automatically move all or substantial portion of your data to the lower cost storage class, appropriately.

You can use S3 "Static" Website Hosting to even eliminate your web server tier. Static in quotes because it is not exactly static in nature – you can do a bunch of active stuff as it does support JavaScript (includes AWS SDK) and **Cross-Origin Resource Sharing** (**CORS**). The advantages include no servers, no patching, and no scaling rules required. Given that web page sizes are increasing, this approach can avoid costs associated with web server patching, capacity planning, and security scanning, while making content rollouts/updates easier (less testing required).

Amazon Glacier storage class can be used to store backups and archive old data. Amazon Glacier is low-cost storage with 99.999999999% data durability. Data restores from Glacier storage can take anywhere from a few minutes to a few hours. However, this can result in 50 to 60% savings on storage. You can also specify life cycle rules to automate data movement from S3 to Glacier storage.

Using the Reduced Redundancy Storage (RRS) option in Amazon S3 storage can reduce your costs by storing noncritical and easily restorable data at lower levels of redundancy than the standard storage option. Amazon S3's reduced redundancy option stores data in multiple facilities and on multiple devices, but the data is replicated fewer times. RRS provides 99.99% data durability versus 99.999999999% using the standard option. This can lead to savings of additional 15 to 20% on storage.

Databases (particularly RDBMSs) make poor BLOB stores and are a poor choice in terms of performance, management, and costs especially Multi-AZ settings. Store blobs on S3 with reference URLs in DB.

Optimizing database utilization and costs

Caching and Read Replicas can reduce the capacity required for your database instance in read-intensive transactions/applications. For example, for read-intensive workloads in particular, using caching instead of PIOPs on a DB server can cut down your expenses, substantially (sometimes up to 80-90%). For caching the data, you can leverage the spare local RAM caches available in your application server instances or use Amazon ElastiCache (there is a cost involved but that may be lower than additional capacity allocation for your database instance for your application type).

The most common and the simplest optimization for databases is to cache as much data as you can because caching saves money. For example, if you are using DynamoDB, then as DynamoDB is charged by provisioned throughput, you can scale that down significantly using ElastiCache (which not only makes for faster responses but is also cheaper because it is charged per hour). In addition, use negative caching, if you are executing the same queries repeatedly that do not return results every time the query is executed. If there is no result returned for a specific query from the database, then save that information in the cache too. This way, you can achieve 80-95% cache hit ratio for queries that return no result most of the time.

You can also use Amazon SQS to buffer writes that exceed your provisioned capacity for the database instance. This approach allows you to provision for average capacity rather than the peak.

You can use dynamic DynamoDB, an open source tool (Python script) that automatically resizes read and write capacity by consuming CloudWatch metrics to decide the best capacity to be booked. It achieves a kind of auto scaling for DynamoDB. In addition, you will typically have hot spot tables in your databases—look to put these in NoSQL or cache storage. Replace hot spots in RDS with DynamoDB because in large TB-scale databases with Multi-AZ configurations, this can yield even more savings – 30 to 40%.

Try offloading your traffic for popular content to Amazon S3 and/or Amazon CloudFront (they are like proxy caches) to scale down your backend DB infrastructure.

Using AWS services

AWS makes available a bunch of ready-to-use services that you can integrate into your application. Leveraging existing services can help reduce the infrastructure, you need to maintain, scale, and pay for, while getting the benefits of scalability and high-availability out of the box. In most cases, this will result in a leaner and more efficient architecture. For example, use Amazon RDS, DynamoDB, ElastiCache for Redis or Amazon Redshift instead of running your own databases. Similarly, use Amazon Elasticsearch service instead of setting up your own Elasticsearch cluster. This strategy makes a lot of sense as AWS has experts for each service who have loads of experience running these services at scale on the cloud.

There are additional operational costs associated with system and database administration tasks. For example, there are costs associated with DB upgrades, including taking backups, creating rollback plans, running applications in a staging environment, migrating to the production environment, and testing, verifying, and releasing the application, and so on. A switch to RDS significantly lowers the time, effort, and costs of such upgrades. Also, doing a more granular scale out of a table in DynamoDB instead of the whole MongoDB cluster that allows you to scale for what you need to scale and not the whole cluster underneath, thereby saving you time and money.

A good side effect of pushing things to AWS services is that it helps you pick the right service for the job. In most modern applications, there isn't one database solution that meets all the different and specialized requirements of your application. For example, choosing the data storage for your application can involve several different decisions based on your specific requirements. You can select DynamoDB as the Key/Value store for scalable throughput and low latency storage requirements. Amazon Aurora for more complex relational data and queries. Amazon Redshift for big (complex) data with higher latency and ElastiCache for Redis as a in-memory store with very low latency.

If your applications are too small for using auto scaling, you leverage Amazon EC2 Container Service to save on those very small applications. You can consolidate your processing needs with Amazon ECS. ECS is designed to be used with other AWS services; it is extensible and secure; and it provides performance at scale.

Using queues

SQS gives you tremendous power to decouple your architecture. However, resilience is only one part of the story. You can use queues to manage transaction costs. For example, in an application that uses the freemium business model, SQS can trigger auto scaling groups based on your customer types, free or paying. In order to pay customers, you can scale the fleet when there are more than a certain number of requests in the queue or the age of the requests exceed a certain threshold (based on SLAs).

Instead of using queues, you can simplify your architecture further using AWS Lambda service based on S3 event triggers. You can use two buckets—one for incoming input that uses an S3 event to trigger Lambda function for processing and then store the output to another bucket. The main advantages of using Lambda in this scenario include elimination of your response time-based SLAs and getting to enjoy the cost benefits of the free tier available for it, forever. In addition, you can optimize your costs further by using spot instances for your free customers while the Lambda service is used to process requests from your paying customers.

In the next section, we briefly discuss the various environments you should provision for effective cloud-based application development.

Application development environments

You will need to provision several environments in the course of your application development. These environments should be provisioned only when they are actually required. This section discusses these environments and their features.

Development environment

The primary purpose of the development environment is to support development and unit testing activities. This environment is usually provisioned with the smallest instances to support your developers. In addition, you can use a shared database instance with schema space for each of your developers. Depending on the standards within your organization, you can do daily, weekly, or on-demand deployments in this environment. You may or may not provision for HA or configure auto scaling in your development environment. Typically, the development instances are shut down at the end of each day and through the holidays and weekends. However, during crunch periods and to support development teams across different time zones, these environments are kept running for extended hours.

QA/test environment

This environment is typically provisioned to support functional and nonfunctional testing activities. They use smaller instances, and they are not configured for HA and auto scaling (during functional testing). This environment can be configured for auto scaling and HA (only when required) for nonfunctional testing. Like the development environment, this environment is shut down on a daily basis and during holidays and weekends. Application deployments in this environment are as per the planned project schedule (to match testing cycles).

Staging environment

Staging environment should mirror the production environment in terms of configuration. This environment is typically used for User Acceptance Testing (UAT), recreating production issues, testing application patches, and load testing. As this environment mirrors production, it is expensive to keep it running continuously. Hence, it should be spun up only when necessary to support the aforementioned activities. Application deployments occur only when required, for instance, to test the application before a production migrations.

Production environment

Production environments are highly scalable and HA-enabled environments. Auto Scaling is enabled, backups are maintained according to the organization's backup policy, and the environment is monitored, continuously. The instances run continuously and a specific version of the application remains deployed at all times.

Additional environments can be created for the purposes of customer training, demos, and so on.

In the next section, we take you through the process of setting up the infrastructure for our sample application.

Setting up the AWS infrastructure

This section introduces you to provision AWS infrastructure in order to deploy and run the A1Electronics e-commerce application securely on AWS. You will also see the configuration changes required at the application level when deploying it to the cloud.

By the end of this section, you will be familiar with creating EC2 and RDS instances, and the choices you need to make for configuring them for you own deployments. So, let's begin.

AWS Cloud deployment architecture

Before we start, we need to have a deployment architecture in place. The term deployment architecture here describes the manner in which a set of resources such as the web server, the application server, databases, DNS servers, load balancers, or any other specific AWS resources are configured on the network to fulfill the system requirements (and ultimately satisfy your business goals).

Let's get familiar with the AWS-specific terms:

- **Region**: AWS products and services are hosted in multiple locations worldwide. The regions are connected through the public internet. The main criteria to choose a specific AWS region are:
    - Location of a majority of your customers. This reduces network latency and makes for responsive web applications. For our example, since a majority of A1Electronics customers are located in the US; hence, US **West (Oregon)** region is selected.
    - Not all AWS products and services are available across all the regions. A list of AWS services and products available by region is available at `http://aws.amazon.com/about-aws/global-infrastructure/regional-product-services`.
    - The products and services offered by Amazon are priced differently across the regions. For example, we can choose a region with the lowest price for our development work, but for production deployment, we can do a cost benefit analysis to choose the most appropriate region. Pricing of all the AWS products and services is available at `http://aws.amazon.com/products/`.
    - Availability Zone: An AZ within a region can be treated as a traditional data center. AZs in the same region are designed to provide infrastructure redundancy in the event of a catastrophic outage, such as earthquakes, snowstorms, Godzilla attack and so on. The number of AZs in a region is region-specific. In our example, we will select the default AZ.

- **EC2 instance**: It is a virtual server on which you run your applications. These come in various flavors to meet your computing demand. A high compute EC2 instance also has higher network bandwidth and memory associated with it. You cannot have a low compute EC2 instance with high memory and network bandwidth. EC2 instances have fixed CPU to memory ratios. It is best to select a micro instance for our development since it is free. More on EC2 instance types is available at `http://aws.amazon.com/ec2/instance-types/`.
- **Amazon Relational Database Service** (**RDS**): RDS is a fully managed SQL database service. It is nothing but an EC2 instance running a SQL engine of your choice. MySQL, PostgreSQL, Oracle, Microsoft SQL Server plus Amazon's own MySQL-compatible Amazon Aurora DB engine are supported.
- **Security groups**: A security group acts as a virtual firewall for your instance to control inbound and outbound traffic. The security group can be configured by a set of rules for inbound and outbound traffic. The rules define the network protocol, port, and the source and destination IP address ranges to accept or send your data to.
- **Virtual Private Cloud** (**VPC**): Virtual Private Cloud (VPC) lets you provision a private, isolated section of the AWS cloud where you can launch AWS resources in a virtual network using custom-defined IP address ranges. It is like your own private data center. It also provides you with several options on connecting VPC with other remote networks. For our example, we have chosen a default VPC `172.31.0.0/16` CIDR block, which allows us define total 65536 subnets or total 65534 addressable resources.
- AWS resources launched within a VPC aren't addressable via the global internet, EC2 instances, or by resources in any other VPC. Resources can be accessed only by resources running within the same VPC.

- **Subnet**: Subnets are logical segments of a VPC's address range that allow you to designate to a group of your resources based on security and operational needs.
- **Router**: Each VPC comes with a default router in order to communicate with resources outside VPC. For example, connecting to a database server in another VPC.
- **Internet gateway**: Each VPC also comes with a default Internet gateway to connect to the public internet.

Let's begin the construction.

AWS cloud construction

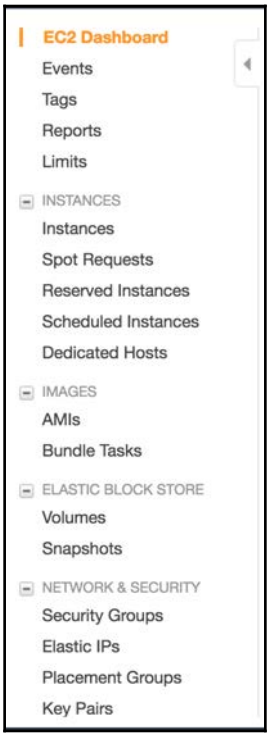
To create a working AWS cloud infrastructure, you will need to create security groups, key pairs, users and roles, MySQL RDS instance, EC2 instances, Elastic IP's and then wire them together. We will create this environment in a bottom up manner where we first create the base AWS constructs such as security groups, key pairs, users, and roles and then we wire them to the EC2 and RDS instances.

Perform the steps in the following sections to set up the infrastructure for our example application.

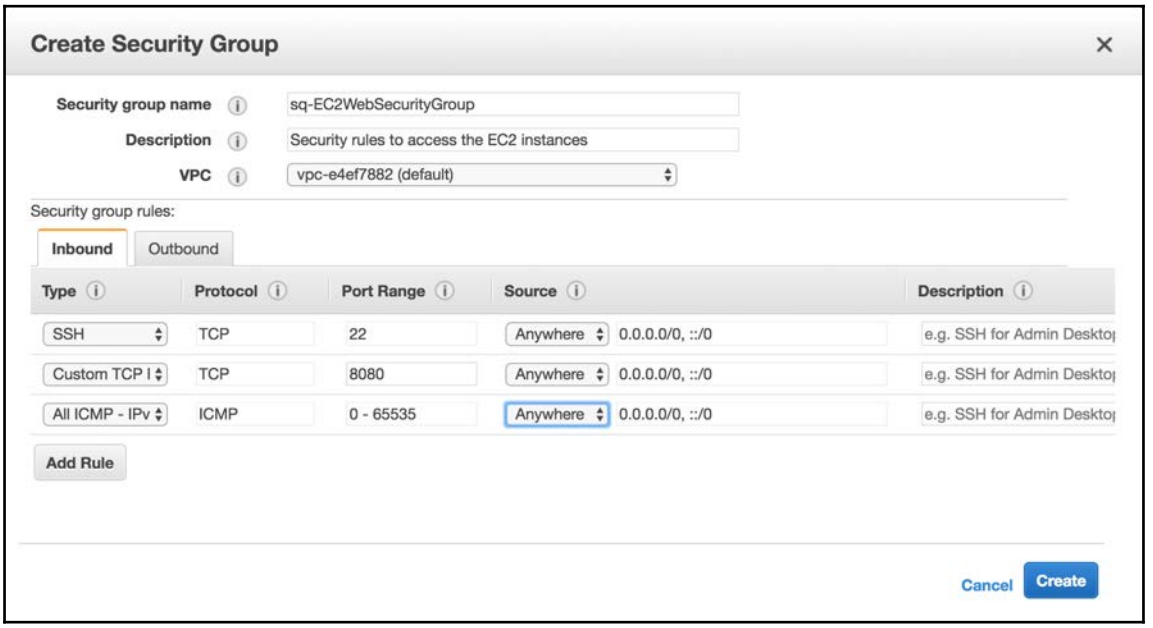
Creating security groups

For our requirements, we will create two security groups: one for the EC2 instance and the other for the RDS MySQL instance.

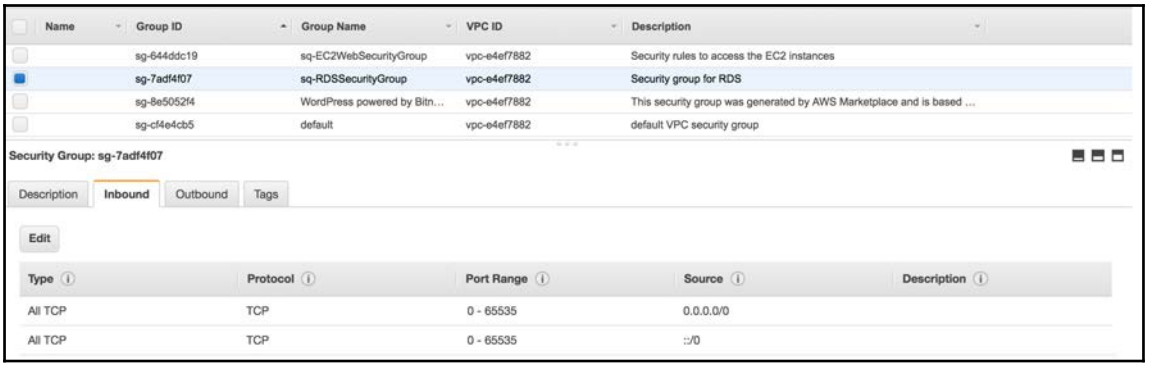
1. From the EC2 dashboard, click on the **Security Groups** link from the navigation pane link and then click on the **Create Security Group** button:



2. Create a security group for EC2 instances to allow the following:
    - Web traffic from any IP address on port 8080 (default Tomcat server port)
    - SSH traffic for remote login from any IP address.
    - ICMP traffic to ping the EC2 instance from public internet.



3. Create a security group for MySQL RDS instances to allow access from the internet. In our example, we can configure direct access to the databases from our development environment. This makes it is easy to make frequent changes and monitor the database without logging in to the EC2 instance, or setting up complex SSH tunnels. In addition, there is the added advantage of not having to install a local MySQL server on your development machine. For your real life AWS environments, it is recommended to allow database access only from within VPC.
4. Select Anywhere from the Source and `0.0.0.0/0` to allow access from any IP address. If you have a static IP address from your ISP, you can enter it here to allow access to all machines from your static IP address, only. If you have dynamic IP address, then you will need to update this rule to the most recent. The figure here displays the list of security groups and the details of the RDS security group.

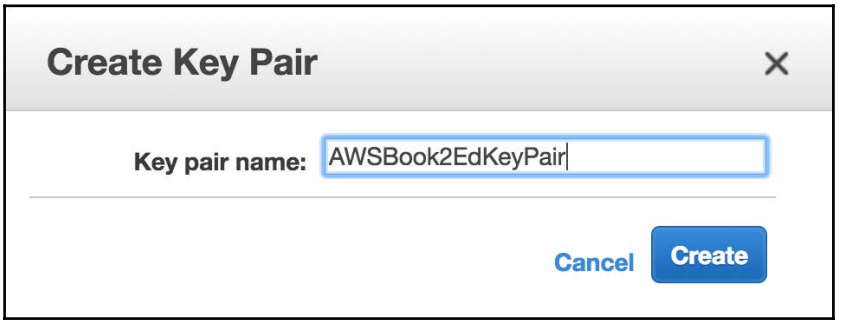


Creating EC2 instance key pairs

AWS uses public/private keys to securely connect to your instances. The public key will be retained by AWS, whereas the private key is downloaded to your computer as soon as it is created.

From EC2 dashboard, click on **Key Pairs** from the navigation pane and then on the **Create Key Pair** button.

You will be prompted with a dialog box asking to enter **Key pair name** as shown. This key pair name will be later used while configuring the EC2 instances.



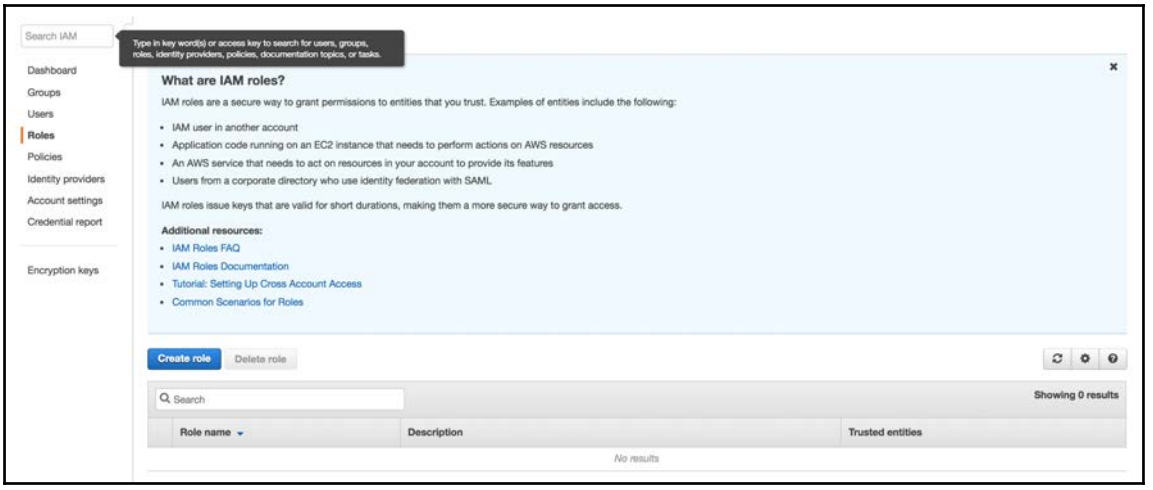
Make sure that you select the correct AWS region from the EC2 dashboard to create the keys because key pairs can't be shared across regions.

As soon as you create the key pair, your private key will be immediately downloaded to your computer. Secure this private key. This private key file can be only downloaded once during creation of the keys. You cannot change access keys in your EC2 instances once they have been assigned.

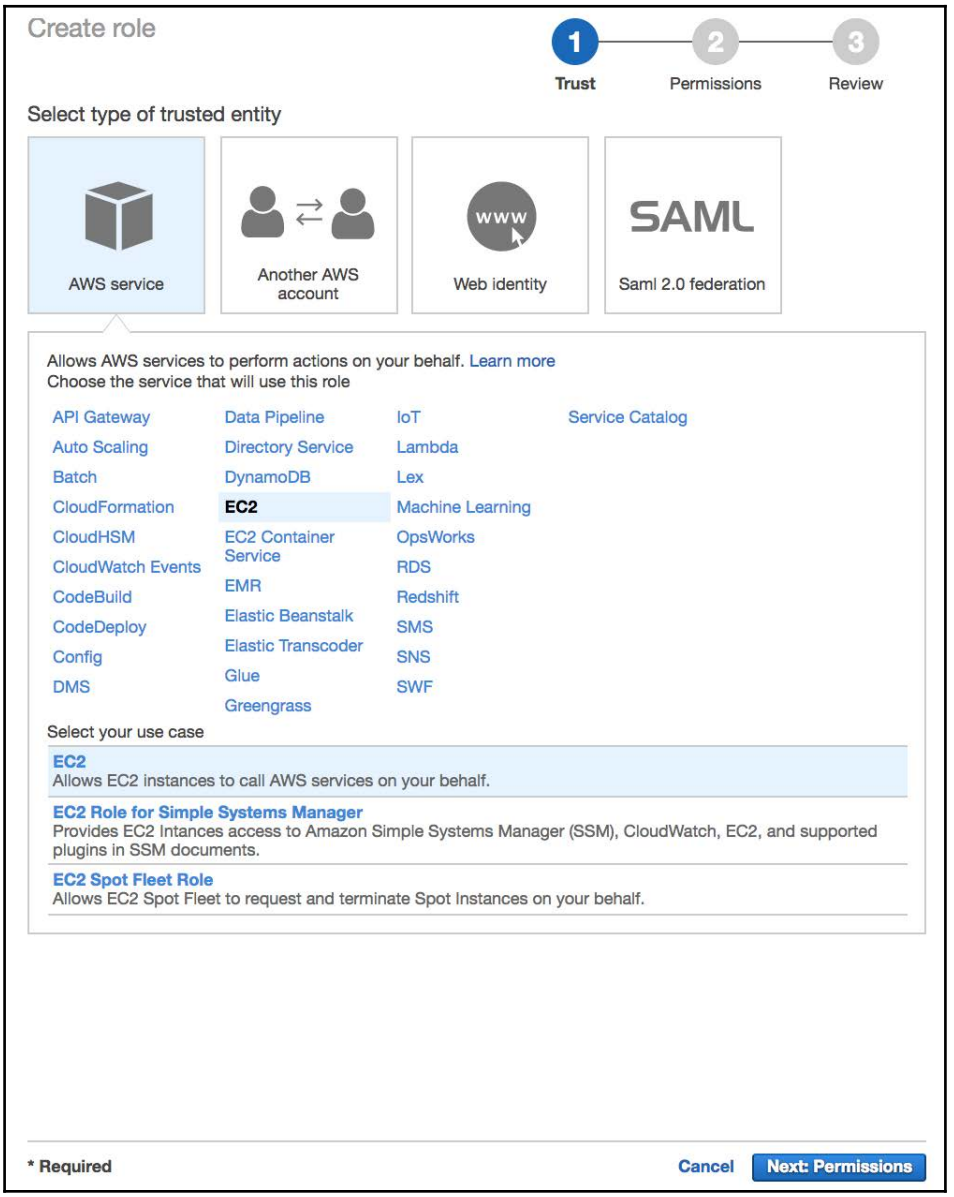
Creating roles

AWS provides a plethora of services to access these services. You will need a strategy to distribute and rotate the credentials to your EC2 instances, especially the ones which AWS creates on your behalf like Spot instances or Auto Scaling groups. A good security practice is credential scoping - granting access only to the services your application requires. AWS solves this issue via IAM roles.

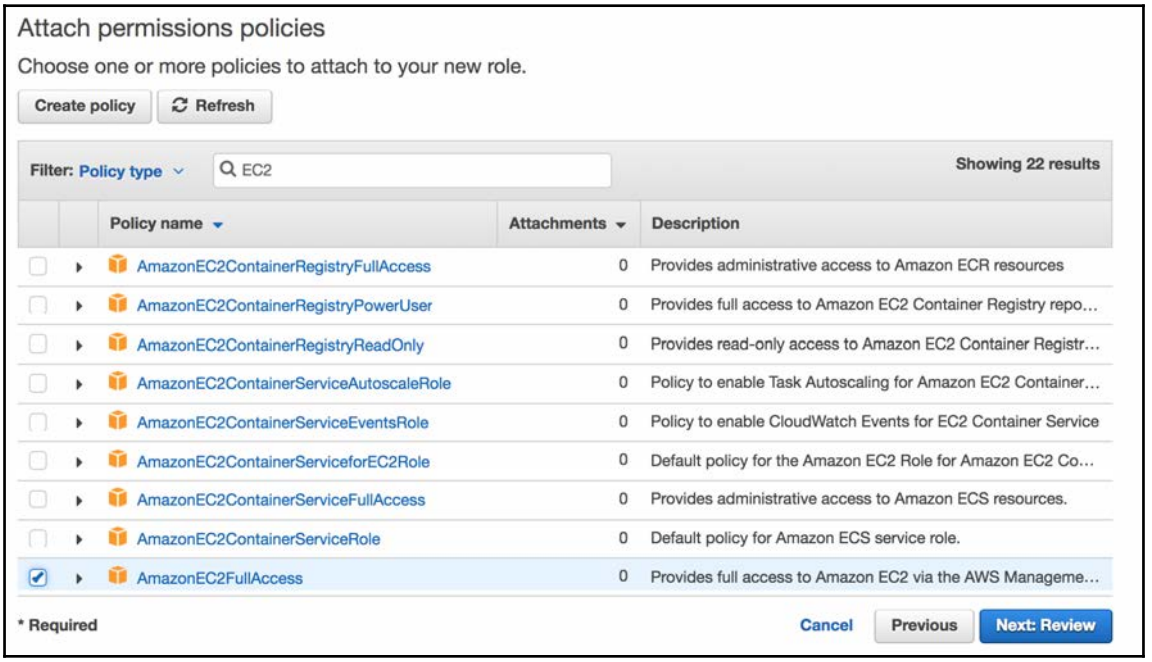
1. From the IAM dashboard, click on **Roles** in the navigation pane link and then on the **Create role** button:.



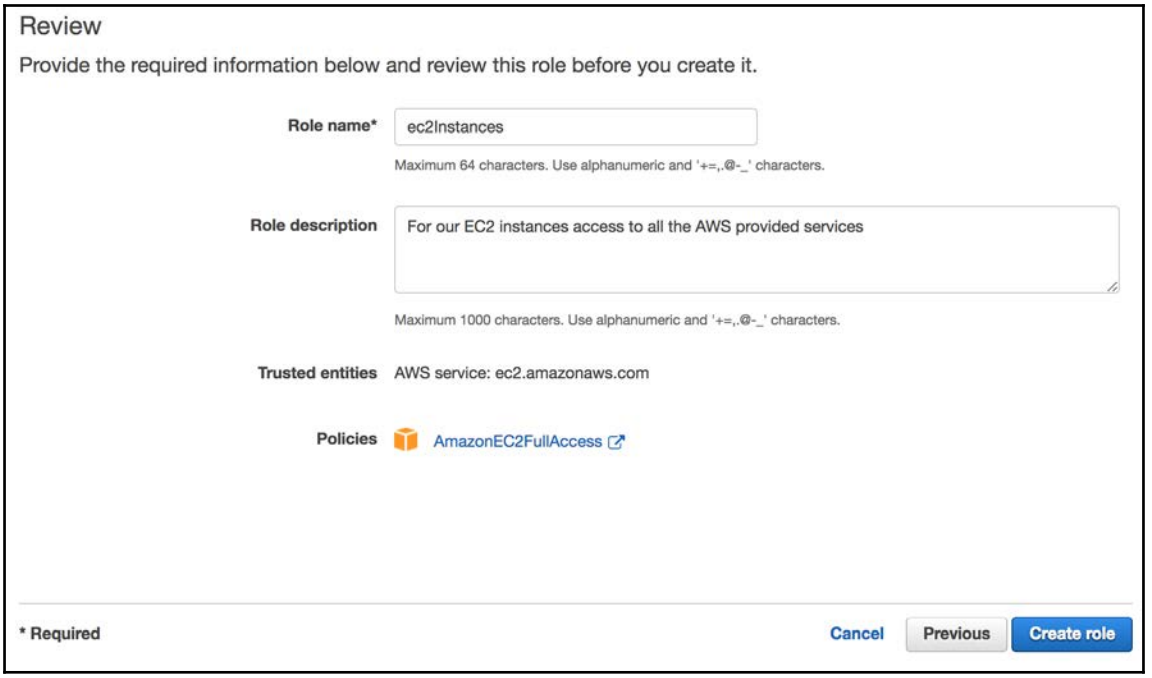
2. Select the EC2 service and then the use case as shown here, and click on the **Next: Permissions** button:



3. Next, we will assign permissions for the selected role. For now, we do not have any credential scoping. Read and write permissions for all AWS services are granted to the role. Permissions to the role can be reassigned even when the EC2 instance is running. Select **Policy name** as **AmazonEC2FullAccess** for our EC2 instances that have access to all the AWS provided services, and click on the **Next: Review** button:



4. Name the role as `ec2Instances` and provide a brief description in the **Role description** field. Click on the **Create role** button:



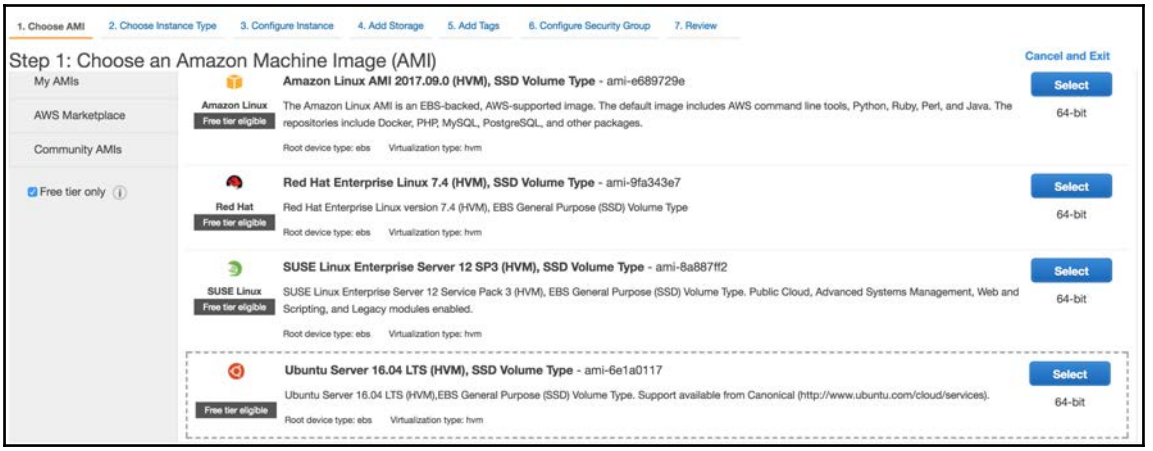
5. After the role is created, it should be listed as shown:



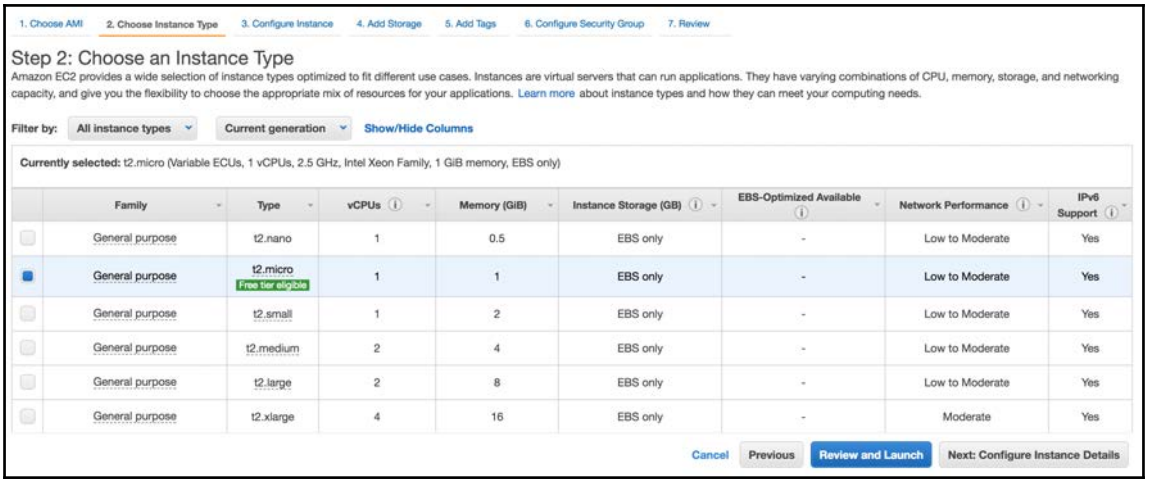
Creating an EC2 instance

Since we have already done the groundwork in the previous steps. Now, it is just a matter of creating an EC2 instance. From the EC2 dashboard, click on **Instances** in the navigation pane and then on **Launch instance**. This will start a process of provisioning an EC2 instance.

1. The next step is to choose an EC2 instance, and this is done by choosing the right **Amazon Machine Image (AMI)** as per our requirements. Select the **Ubuntu Server 16.04 LTS (HVM) SSD Volume Type AMI**:

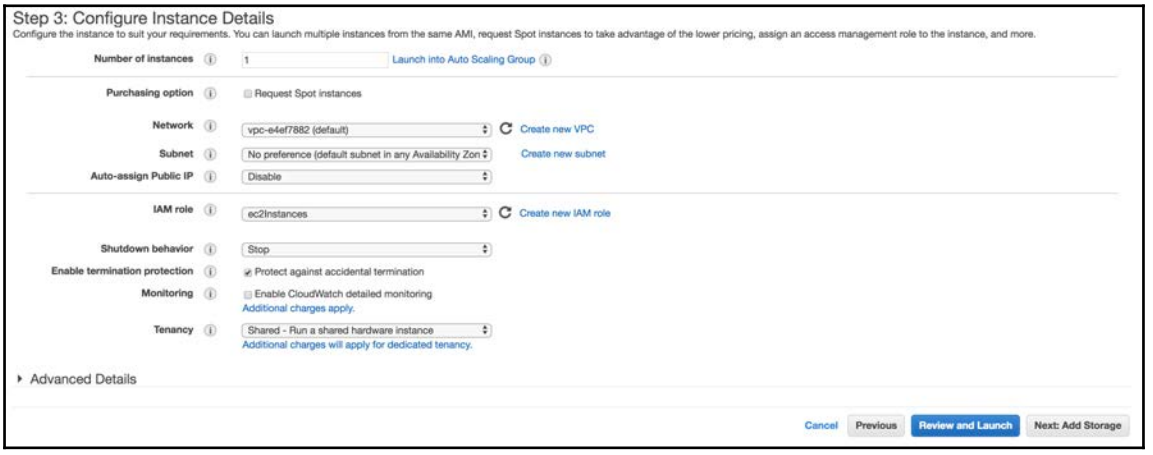


2. After selecting an AMI image, the next option is to choose an instance type. The instance is the virtual server that will run our application. Select the **t2.micro** instance, which is included in the free-tier for a period of 1 year from the date you created your AWS account. Click on the **Next: Configure Instance Details** button:

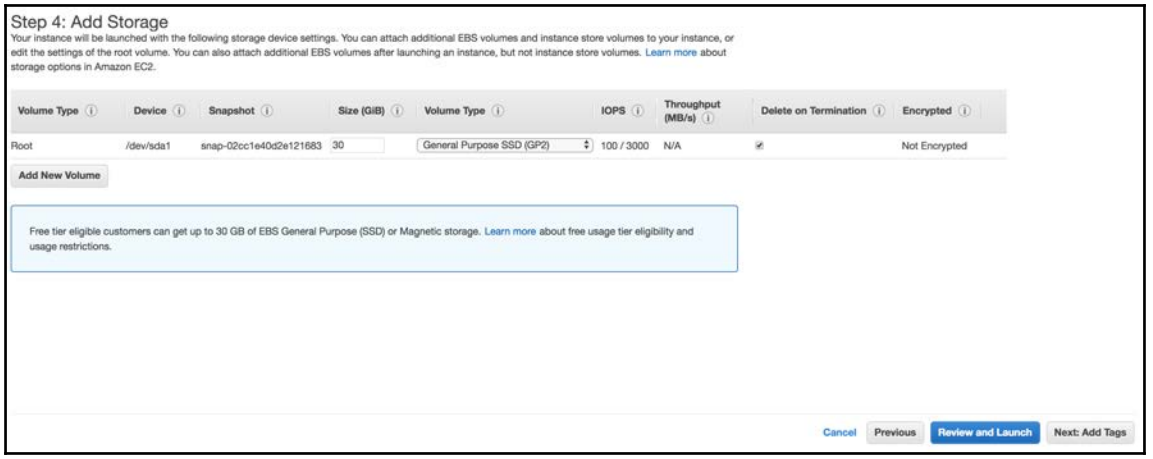


3. Next, we configure the EC2 instance. There are several options available at each step, and we need to make the most appropriate choices for our purposes:
    1. **Number of instances**: This allows launching of multiple AMI instances. By default, it is set to `1` (no need to change that). You can always launch multiple instances via the EC2 dashboard.

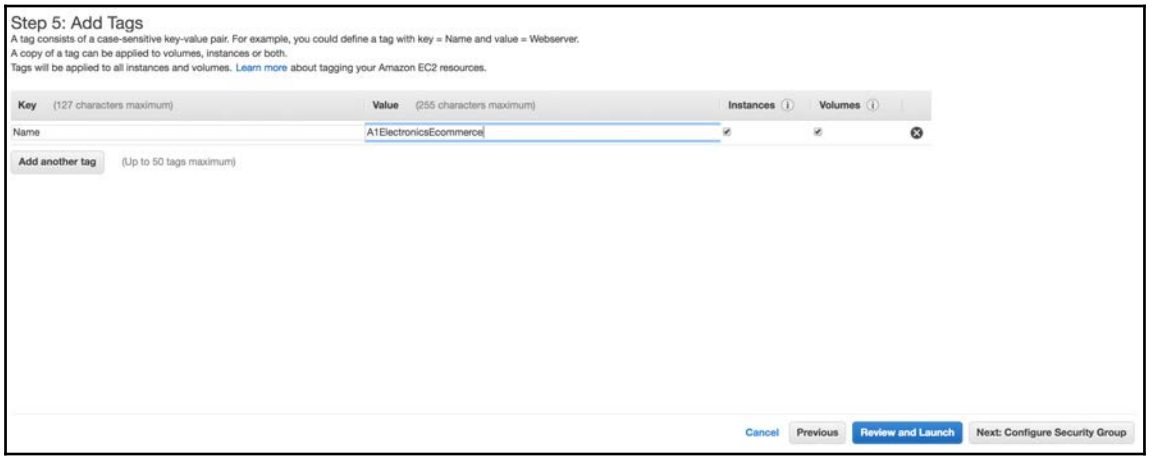
2. **Purchasing option**: Since we are using the free tier, we can ignore this. The idea of purchasing option relates to excess capacity for an instance type in an AWS region made available to use at a lower price point.
3. **Network**: By default, all EC2 instances are launched in VPC. We use the default VPC.
4. **Subnet**: By default, each subnet is associated with an availability zone within a region. Select the **No preference** (default subnet in any AZ).
5. **Auto assign Public IP**: When an EC2 instance starts, it can request a public IP address from Amazon's pool of public IP addresses (so that it can be a part of the public internet). This public IP address will be available as long as the EC2 instance is on. Each time the EC2 instance starts, it will get a public IP address from the Amazon's pool of public IP address. The public IP is not persistent. If we want the public IP address to be persistent across restarts, then we have to use Elastic IP, which we will set up in a later step. Set this to **Disable** for now.
6. **IAM role**: Select the role ec2Instances created earlier.
7. **Shutdown behavior**: An instance can be either stopped or terminated on shutdown. Select **Stop**.
8. **Enable termination protection**: It is a mean to disable the terminate option for the EC2 instance in the EC2 dashboard. Select this option.



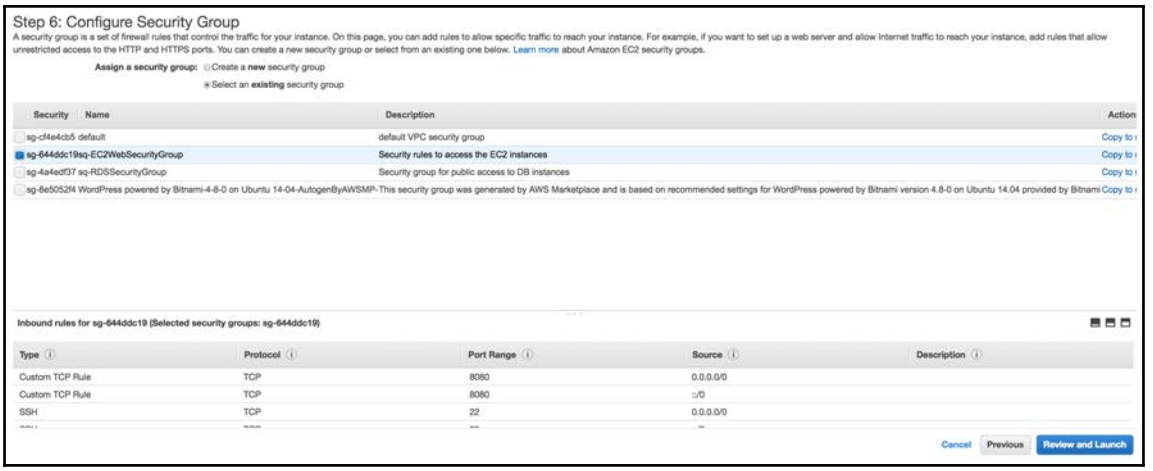
9. **Monitoring** (Enable): This is to enable collection of metrics and analysis via AWS CloudWatch. Logging of basic metrics are free (with some restrictions). Refer to `https://aws.amazon.com/cloudwatch/pricing/` for what you get free and what you have to pay for. For our purposes, you do not need to select this option.
10. **Tenancy**: Shared tenancy uses an over-subscription model to rent the hardware among the customers. This makes the performance of the EC2 instance unpredictable at times. To overcome this problem, Amazon also provides a dedicated tenancy option, which costs more but reserves the instance exclusively for your use. Select the Shared tenancy option from the dropdown.
11. **Advanced Details**: This option is used to pass user data or scripts to the EC2 instance. Right now, we do not pass any user data or scripts to the EC2 instance. So, no changes are needed.
12. Click on **Next: Add Storage** to provision persistent storage.



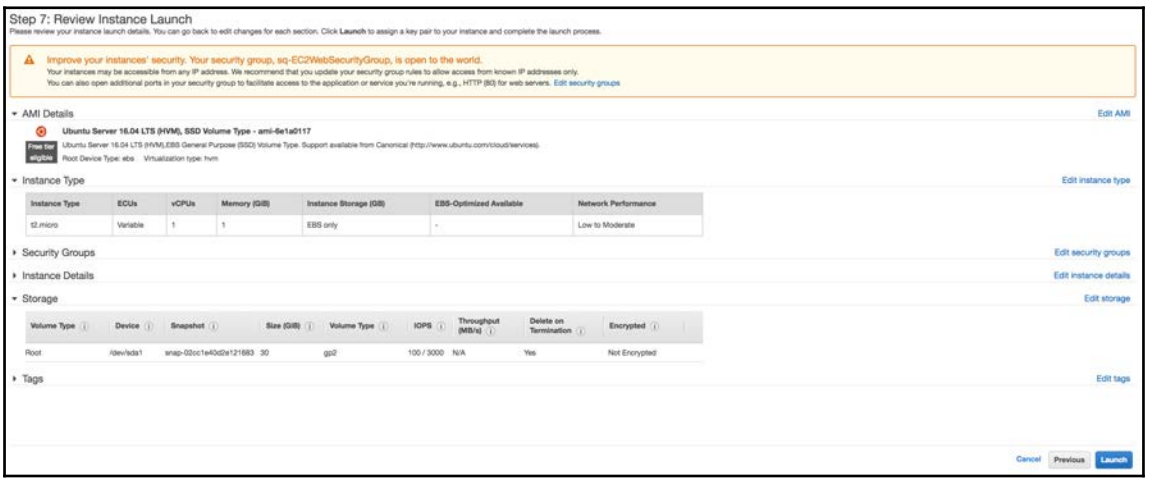
4. Next, we configure the persistent storage also known as Elastic Block Storage (EBS). EBS does not go away when the system reboots or crashes. It is the hard disk for your EC2 instance. Up to 30 GB of disk is available in the free tier, which is sufficient for most applications. Select **General Purpose SSD (GP2)** from **Volume Type** column. The data access speed of the disk is proportional to the size of the disk. It is defined in terms of IOPS which stands for input output operations per second. One IOP is defined as a block of 256 KB data written per second. Click on **Next: Add Tags**:



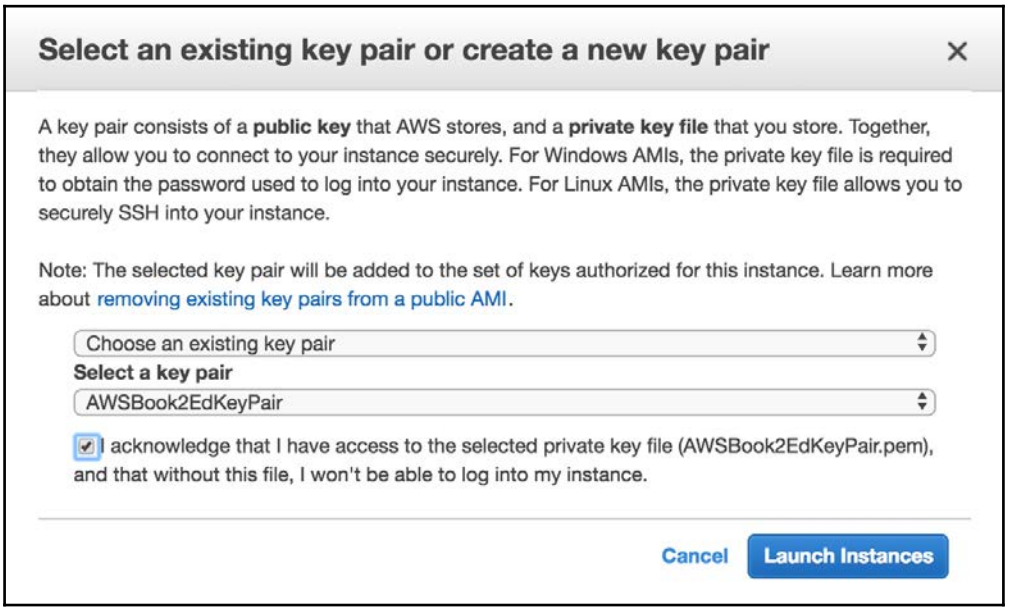
5. Next, we tag the EC2 instance. Tags do not have any semantic value and are treated purely as strings in a key-value form. You can work using the tags with the AWS management console, EC2 API and EC2 command-line interface tools. Click on **Next: Configure Security Group**:



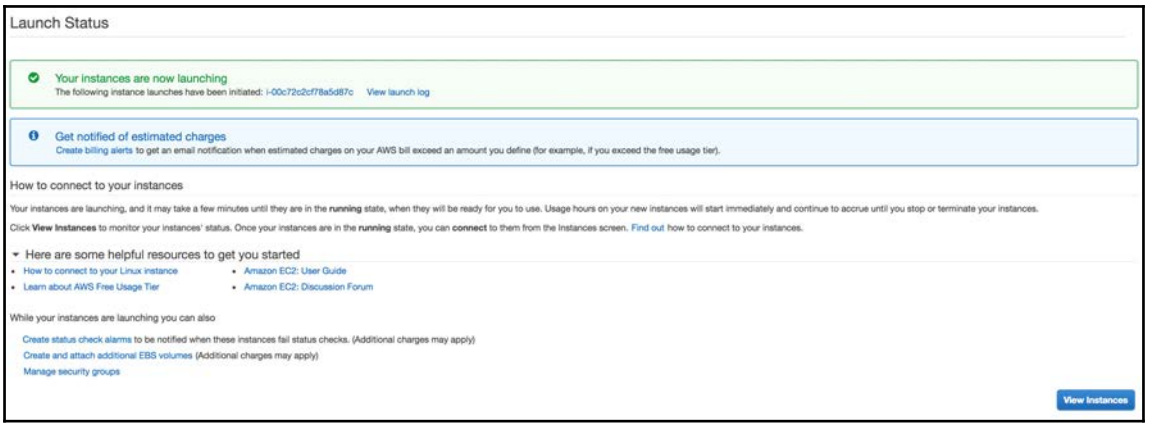
6. Next, we assign the security group **sq-EC2WebSecurityGroup** we defined earlier in step 1. Click on the **Select an existing security group** radio button to view all the available predefined security groups. Select the **sq-EC2WebSecurityGroup** from the list. Click on **Review and Launch**:



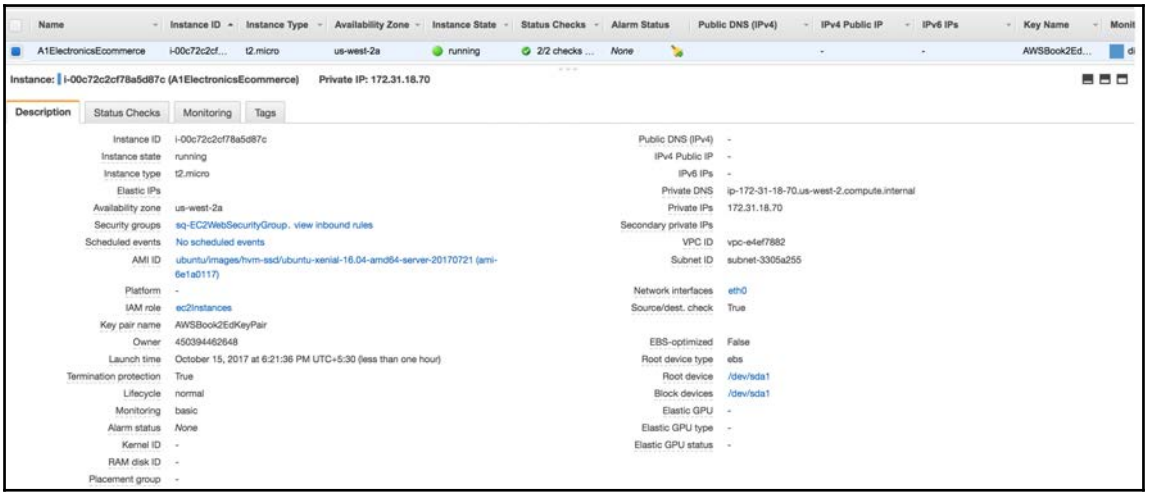
7. Next, we can review the options we have selected, and modify them, if required. Click on **Launch** to launch the instance.
8. Upon launch, the EC2 instance will prompt you to select the public/private key pair that was created earlier. Select the **ec2AccessKey** from the drop-down list box. Click on **Launch Instances** to launch the EC2 instance. The key pair once assigned to instance cannot be changed. Make sure you store your private key, securely.



9. Your EC2 instance will take some time to start.



10. After the EC2 instance is up and running, you should see it listed in the console. You can review the details of the instance to ensure that it is as per what you configured:



You cannot access the instance as it does not have a public IP associated with it yet. The EC2 instance is assigned an IP address from the VPC subnet, in this case, it is `172.31.18.70`. This EC2 instance can be used only for communication between the instances in your VPC.

Next, we create an elastic IP and assign it to the EC2 instance so that it can be accessed via the public internet.

Creating and associating Elastic IPs (EIP)

EIPs are dynamically re-mappable static public IP addresses that make it easier to manage EC2 instances. Each EIP can be re-assigned to a different EC2 instance when needed. You control the EIP address until you choose to explicitly release it. An Elastic IP is associated with your account and not a particular EC2 instance. Since, public IP addresses are a scarce resource you are limited to 5. If you need more EIPs then you have to apply for your limit to be raised. If you have a large deployment then an elastic load balancer (ELB part of AWS) is placed in front of all the instances thereby consuming one EIP only.

You will be charged for all EIPs not associated with running EC2 instances. It is charged at 0.01$/hour for each unassociated EIP.

Following are the steps to create an Elastic IP:

1. From the EC2 dashboard, click on **Elastic IPs** in the navigation pane and then on **Allocate New Address**. Click on the **Allocate** button:



2. You should see the following message that the new address request has been successfully processed. This step allocates a new EIP associated with your account:

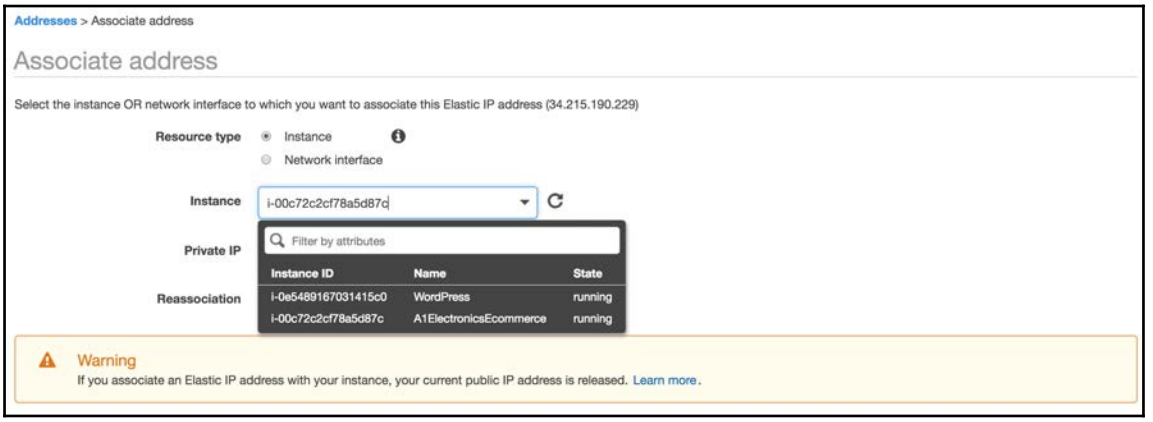


Following are the steps to create an Associate IP:

1. The next step is to associate the EIP to an instance. Click on the **Associate Address** menu item:



4. Select the **A1ElectronicsEcommerce** instance from the drop-down list:



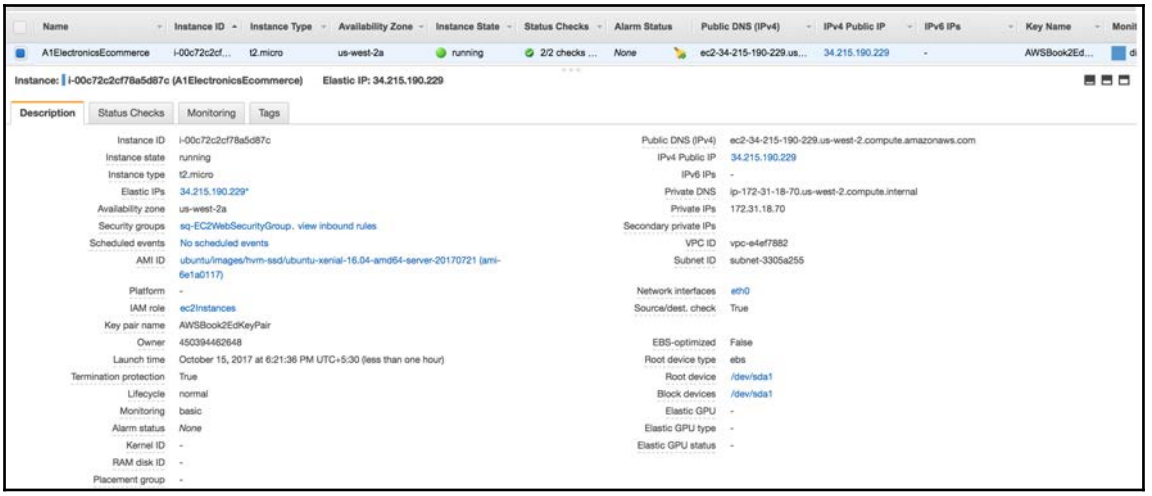
5. Click on the **Associate** button:



6. You should see the following message confirming that the associate address request was successfully completed:



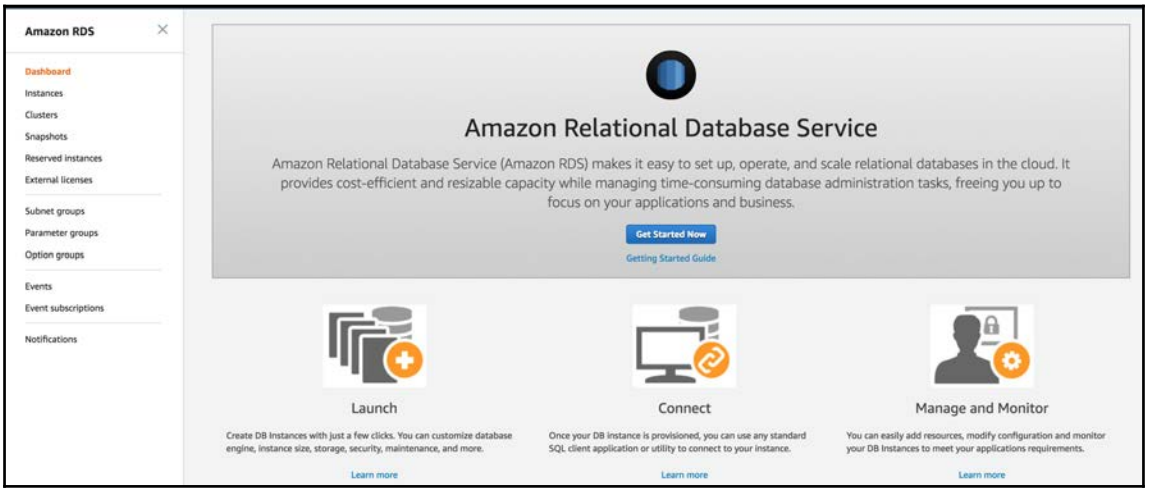
7. From the EC2 dashboard, click on **Instances** in the navigation pane then on **A1ElectronicsEcommerce** to view the details. The EIP is assigned to the instance. Use ping to test the instance either by the EIP address `34.215.190.229` or by the domain name **ec2-34-215-190-229.us-west-2.compute.amazonaws.com** from the terminal:



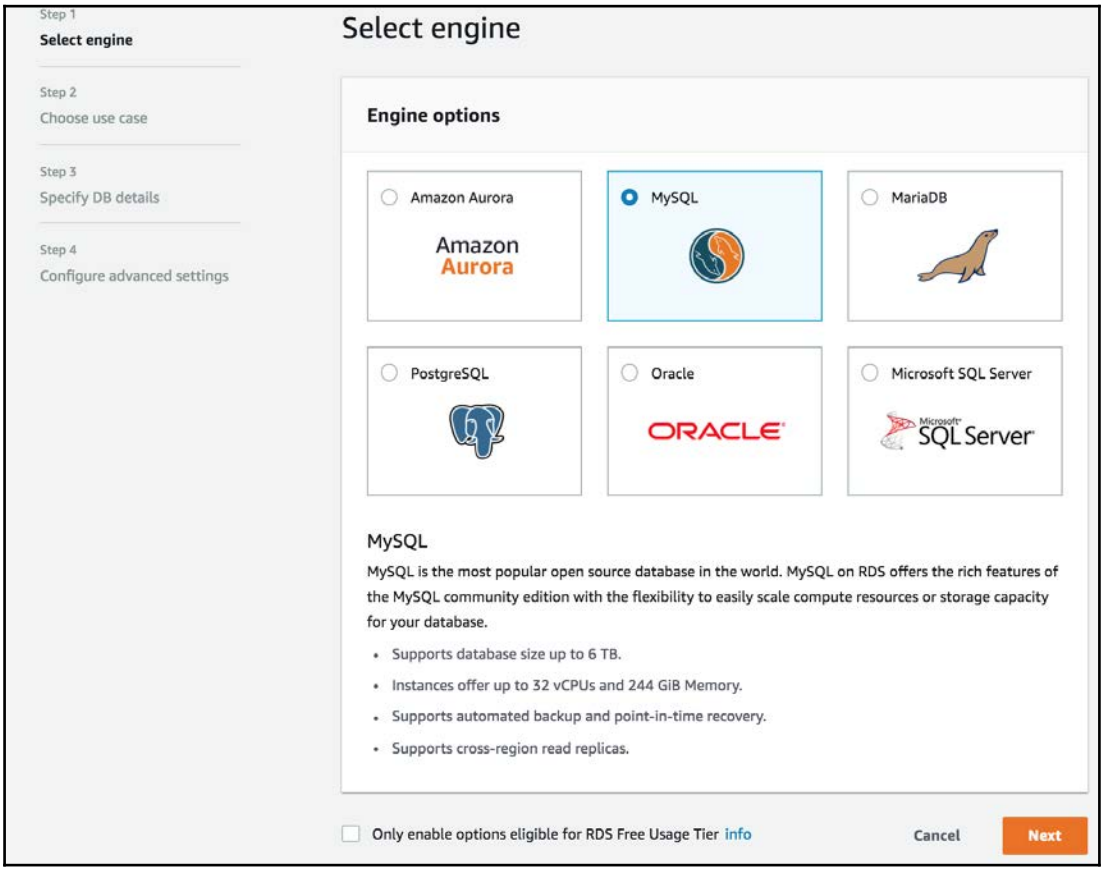
Configuring the Amazon Relational Database Service (RDS)

As we have the A1ElectronicsEcommerce EC2 instance is up on the cloud, we now need to create a RDS instance within our VPC for our A1ElectronicsEcommerce web application.

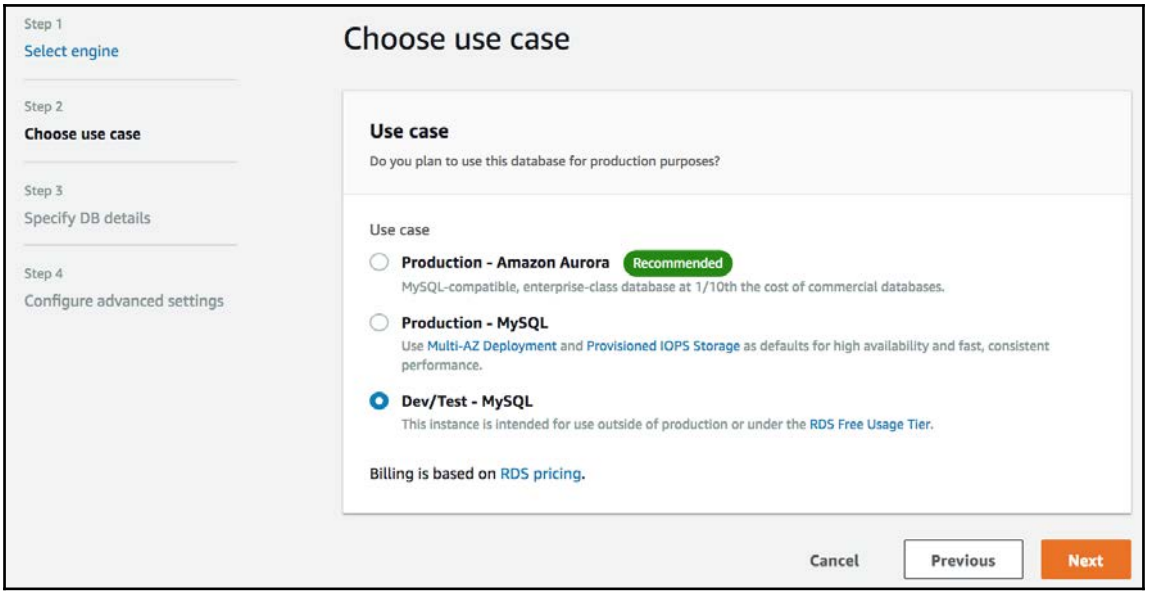
1. From the RDS dashboard click on **Launch a DB Instance** instance. This will start a process of provisioning a RDS instance:



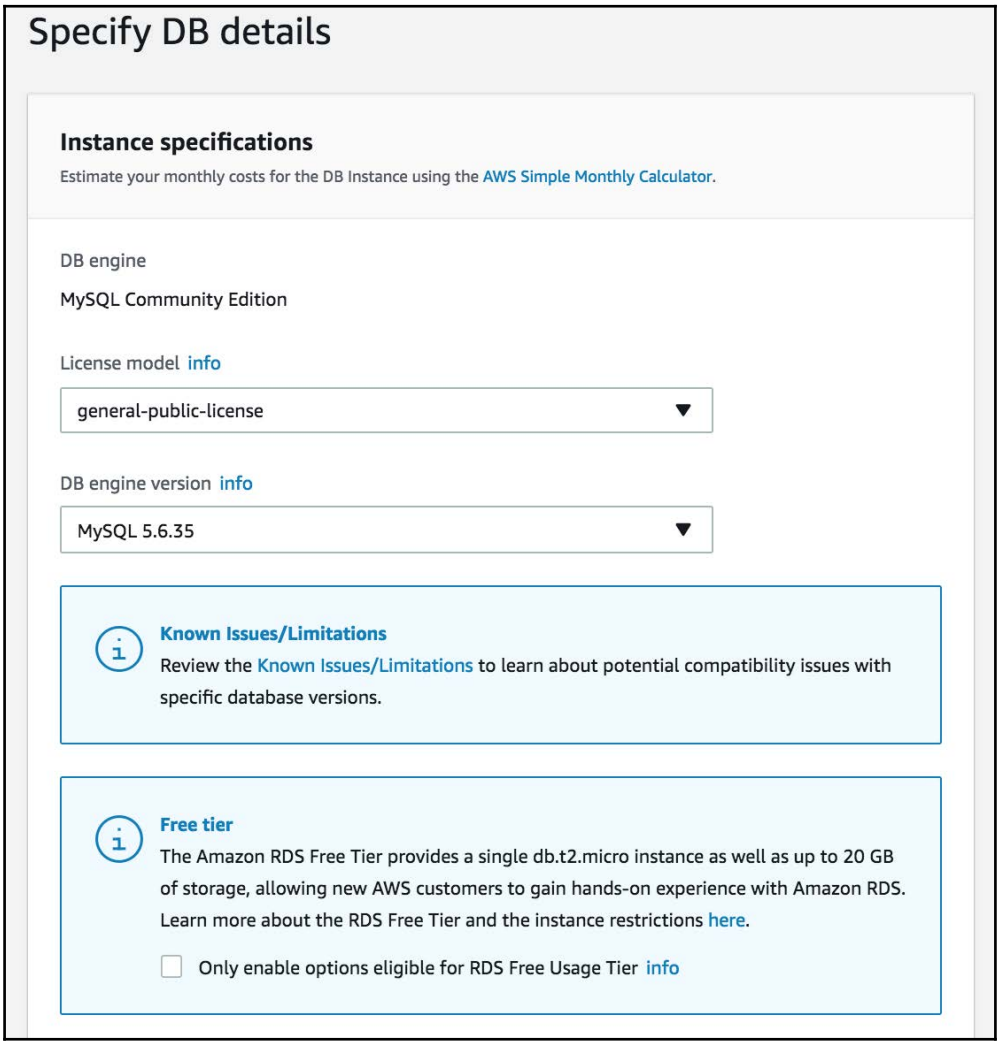
2. The next step is to select the SQL database engine. For our application, we will select MySQL. Click on the **Next** button:



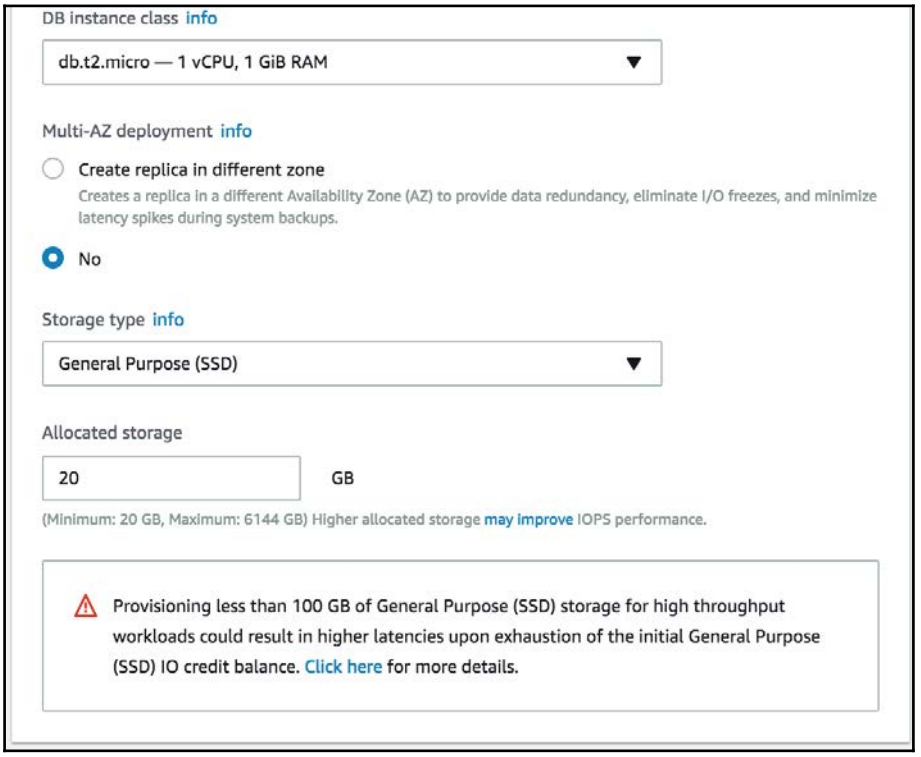
3. The next step is to decide if the RDS DB instance will be used for production environment or outside of it. Under production environment RDS provides option for high availability RDS instance. It also provides an option of provisioning IOPS for your RDS DB instance as per your application's need. All this sounds good but the costs can add up, quickly. As we are in a development mode currently, we will ignore the production choices for now. We select the **Dev/Test - MySQL** option, and click on **Next**:



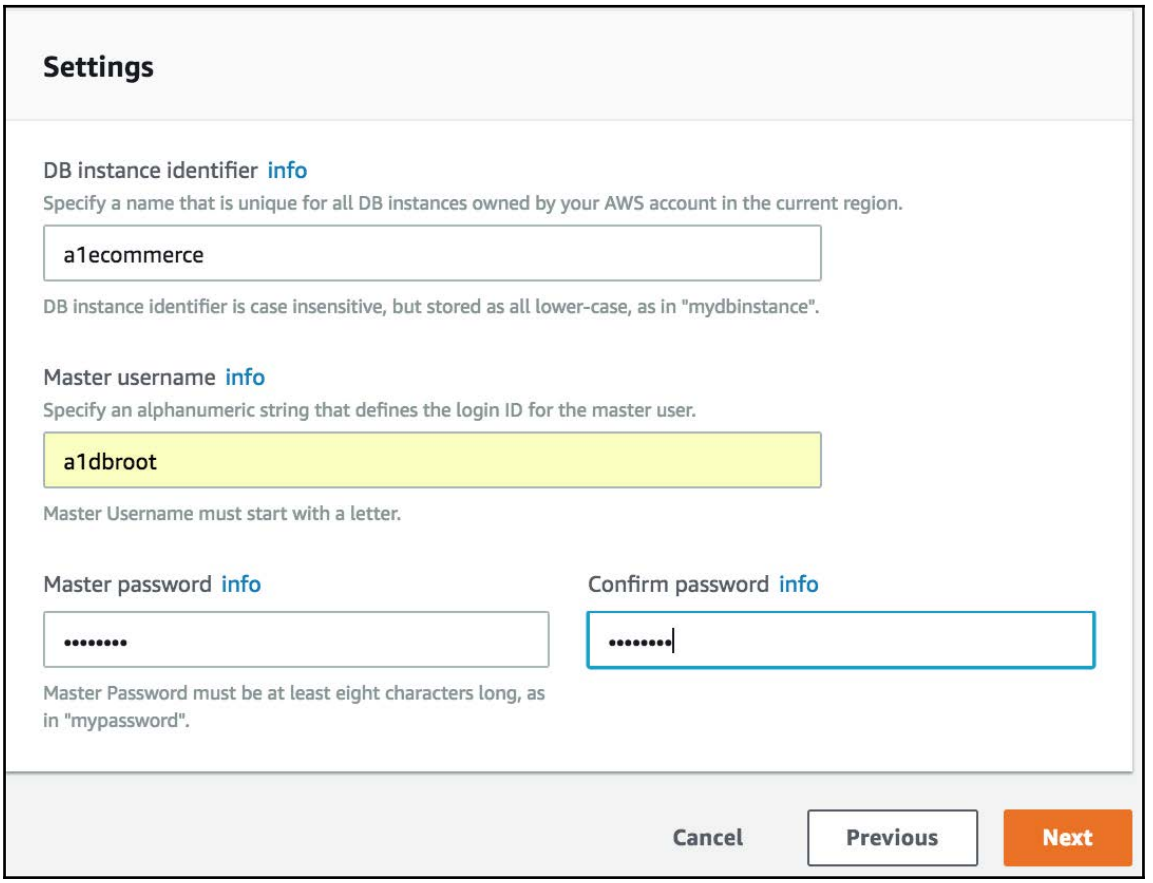
4. The next step is to configure the RDS instance.
    1. **License model**: Since we are using the **MySQL Community Edition**, general-public-license is the only option available.
    2. **DB engine version**: This option allows you to select a specific version of MySQL. Choose the latest unless you have MySQL specific code that runs for a specific version:



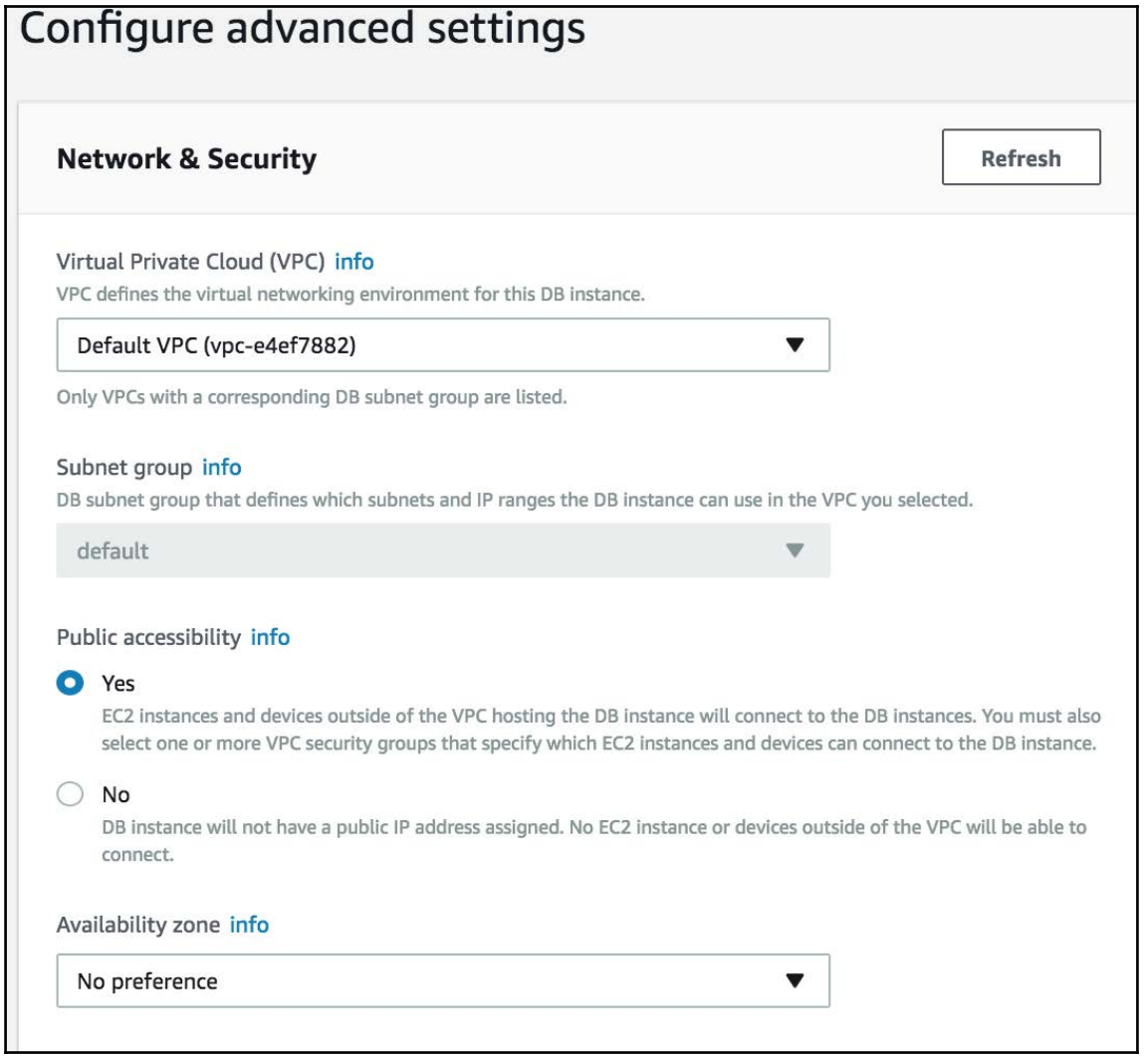
3. **DB instance class**: This is the same as choosing the EC2 instance type. This will select the virtual server, which will run your MySQL database engine, faster DB instances can be chosen as per your database workload after profiling them, **db.t2.micro** is the only one which is available for the free tier so we select it.
4. **Multi-AZ deployment**: This option is for high availability as discussed earlier. Select **No** from the dropdown list.
5. **Storage type**: Select **General Purpose (SSD)**. The other option is **Provisioned IOPS** which kicks in only if your allocated storage is 100 GB or more, and Magnetic which is slower.
6. **Allocated storage**: You use the minimum, which is 20 GB. The free tier allows storage up to 20 GB:



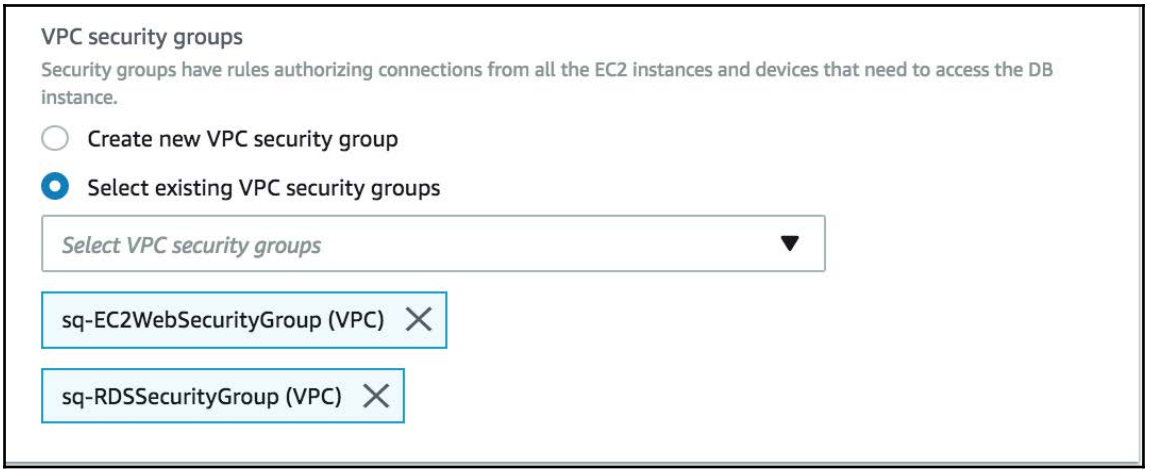
7. **DB instance identifier**: This is the identifier for the MySQL server database instance, and this identifier is used to define the DNS entry for the DB instance. Type `a1ecommerce` in the text field.
8. **Master username**: Master login name to access the DB instance, it needs to start with an alphabet. Enter `a1dbroot` for the master username.
9. **Master password**: Password for the master username.
10. **Confirm password**: Type in the master password again.
11. Click on the **Next** button:



5. Next, we configure some advanced settings:
    1. **Virtual Private Cloud (VPC)**: This is the VPC network where the DB instance will reside. It is the same default VPC network in which our EC2 instance resides. Since there is only one VPC defined, select the Default VPC from the drop down.
    2. **Subnet group**: This allows for the selection of a DB subnet. A DB subnet is a logical subdivision of the VPC network space. This is useful in large implementations where you might have a use case for different DB instances being logically separated from each other. Here the DB instance is in same subnet as the EC2 instance. This can be achieved by selecting the correct availability zone. Select default from the dropdown.
    3. **Public accessibility**: It is a good security practice to hide your databases from the internet. But then access to the DB instance is only possible after remotely logging into the EC2 instances running within the same VPC or by setting up SSH tunnels. During the development phase, this becomes very inconvenient and frustrating to manage database schema changes, viewing data, and debugging. So by keeping things simple select the **Yes** option. For production DB instances, this should be set to **No** and a VPC security group that allows access from within the VPC should be created and assigned.
    4. **Availability zone**: Select **No preference** from the drop-down box. It assigns the appropriate subnet to the DB instance:

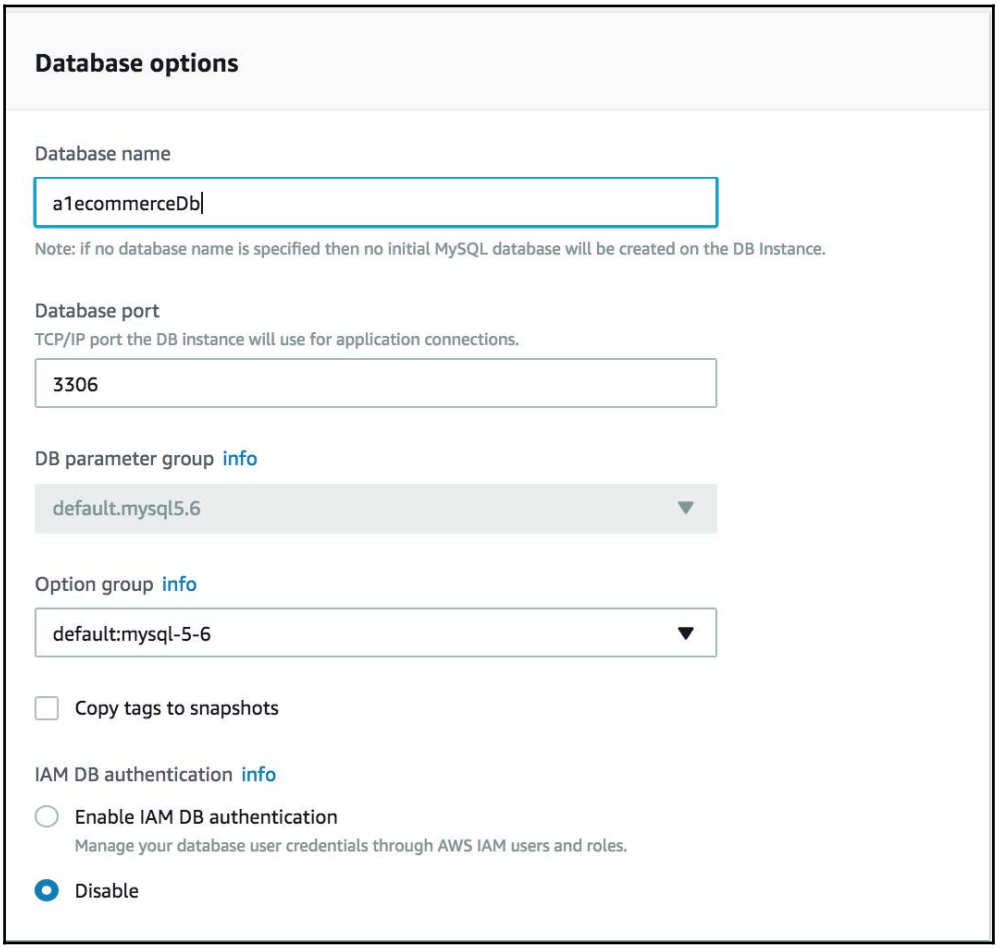


6. **VPC security groups**: Select **sq-RDSSecurityGroup** and s**q-EC2WebSecurityGroup** from the list (these security groups were created earlier):

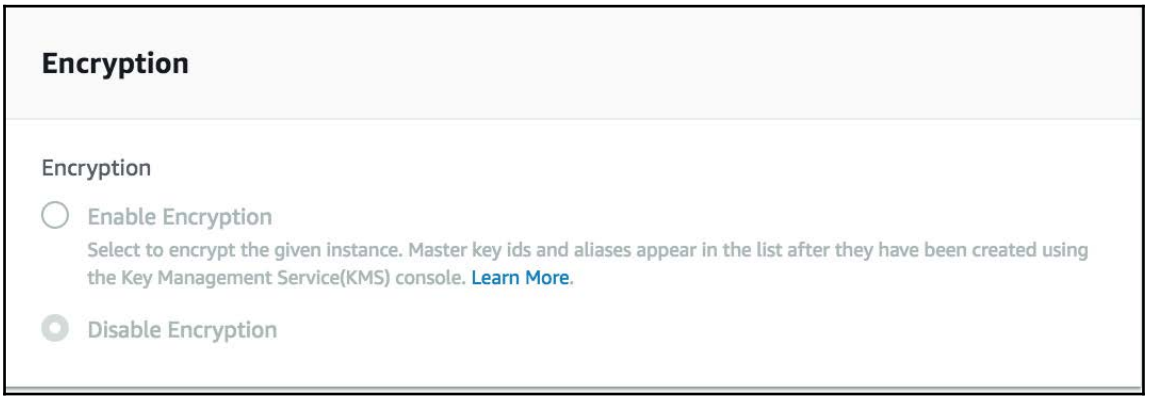


7. In the next set of fields, we specify the **Database options**.
    1. **Database name**: The name of the database to which an application connects to. Name it `a1ecommerceDb`.
    2. **Database port**: The default MySQL port. Do not change the default port number, which is set to `3306`.
    3. **DB parameter group**: Management of DB engine configuration is done via the parameter group. This allows you to change the default DB configuration. Since we have not created any parameter group, select the default `default.mysql5.6`.

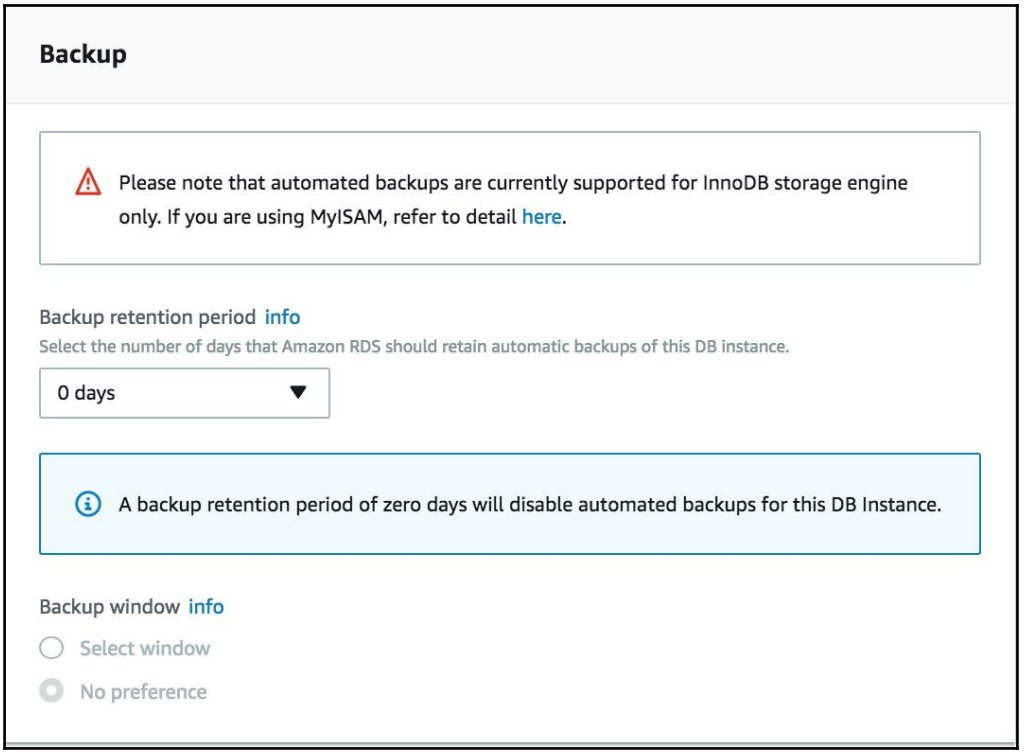
4. **Option group**: An option group allows to set additional features provided by the DB engine to manage the data and the database and to provide additional security to your database. Since we have not created any option group, select the default `default.mysql.5.6`:



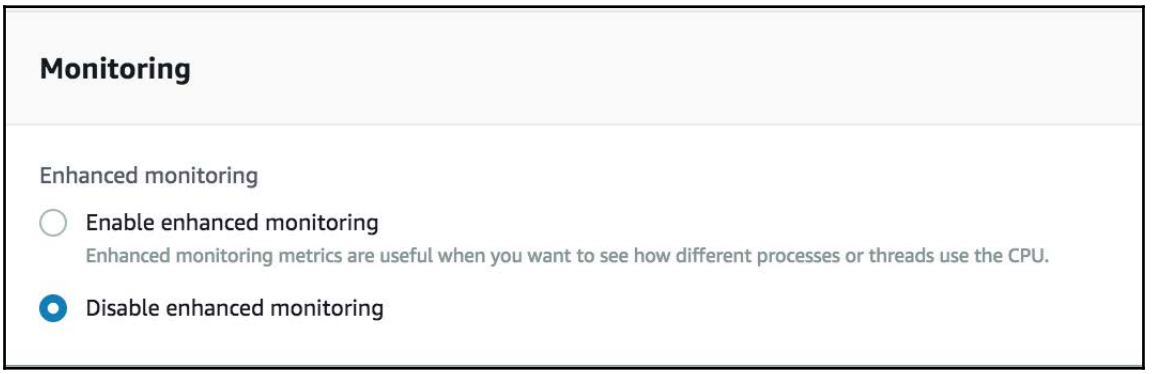
8. We disable **Encryption** option as shown here as we do not want to encrypt the data stored in the database:



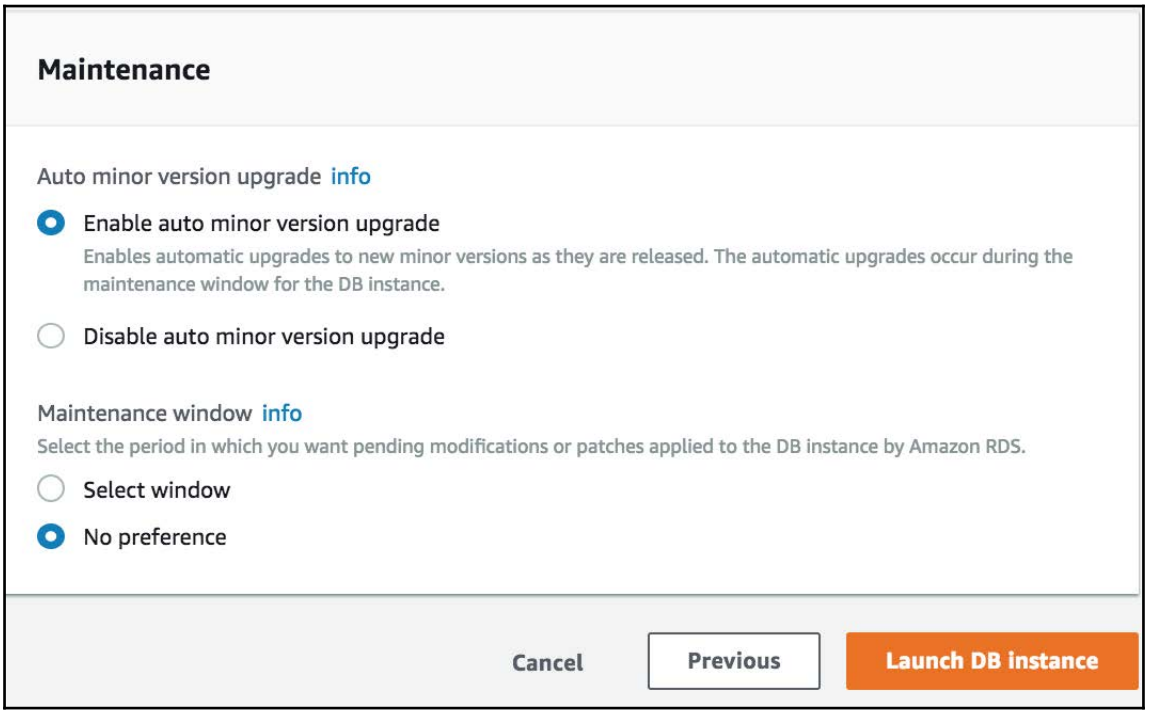
9. Next, we specify the backup parameters for our database.
    1. **Backup retention period**: The number of days Amazon RDS keeps the automatic backup for the instance. The range is between in 1 and 35 days. This helps enable one-click restoration of the data in case of disaster recovery. Selection of 0 days disables backup retention. Select **0** from the drop down.
    2. **Backup window**: The time slot during which the automatic backups take place. The selected time period should be such during which the database load is least. It is normally set when we deploy the database in production. During development cycle this can be set to **No Preference**:



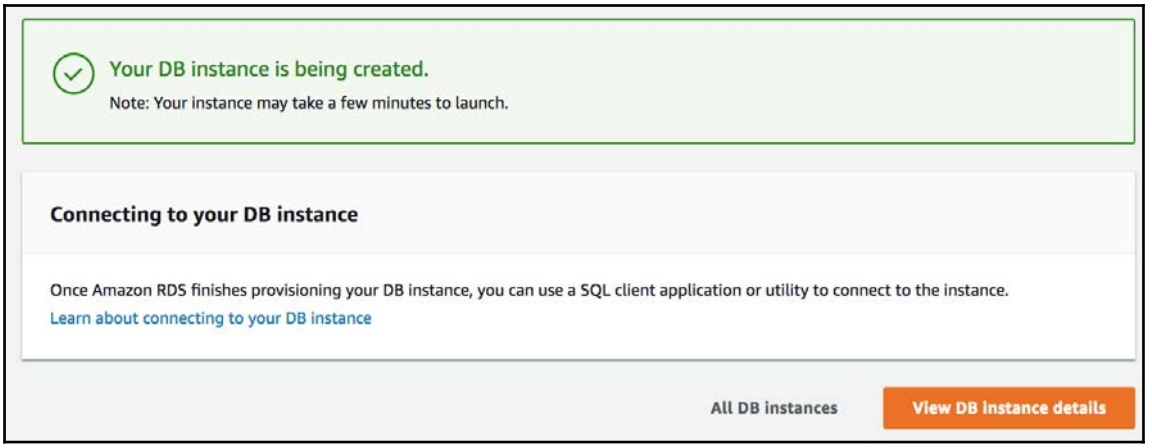
10. We select **Disable enhanced monitoring** for our development database:



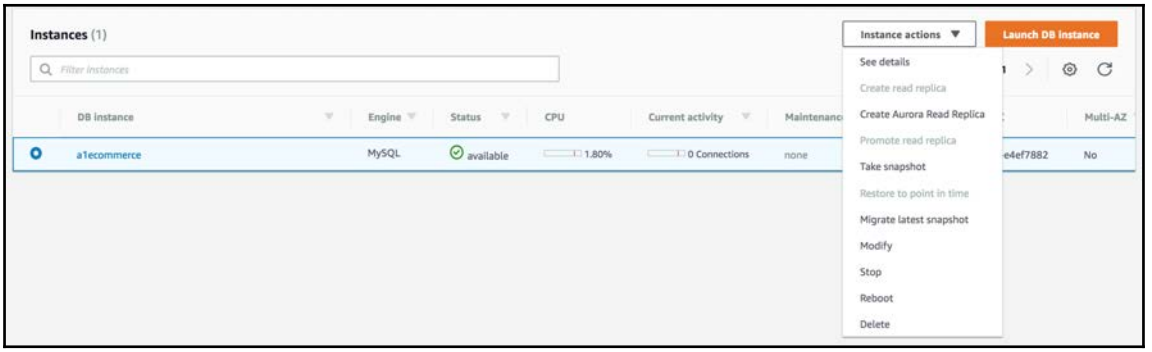
11. Next, we specify the **Maintenance** options.
    1. **Auto minor version upgrade**: Amazon RDS will automatically update the DB instance only for minor updates. Select the **Enable auto minor version upgrade** option.
    2. **Maintenance window**: Any modifications to the DB instance like changing the DB instance class, storage size, password, multi availability zone deployment. These changes take place during the maintenance window period. Again this is useful for production instances. The maintenance window can be overridden during the time of modification of the DB instance. Select the **No preference** option:



12. Click on the **Launch DB instance** button. This will create a DB instance and launch it:



13. From the RDS dashboard, click on **Instances** in the navigation pane and then on **a1ecommerce** to view the details of the DB instance. If you note there is no IP address associated with the DB instance, the only way you can access this DB instance is via the endpoint:



14. Here, we display the **Connect settings** section of the details screen as we will be using some of them in the subsequent steps for installing and verifying the software stack:



Installing and verifying the software stack

The next step is to remote log into the EC2 instance, and install Apache Tomcat and the MySQL client libraries. We will use the private key file created and downloaded under *Creating EC2 Instance Key Pair*s in an earlier step.

- **Assign rights to the private key**: Copy the private key file to the `.ssh` folder in your `home` directory, if for some reason it does not exists then create it and make sure that it has read/write/executable rights assigned for the owner (`drwx------`). To log in from your command line, type in the following to assign correct rights to the private key file downloaded earlier. This assigns read/write and execution rights only to the file owner. Unless the rights are changed, it will not be possible to login remotely.

```
chmod 700 ~/.ssh/AWSBook2EdKeyPair.pem
```

- **Remote login**: Now, we can login remotely. The default user name for Ubuntu AMI's is `ubuntu`, the IP address to connect to is the EIP of the EC2 instance is `34.215.190.229`, type `Yes` when you get a warning that the authenticity of the host `34.215.190.229` can't be established.

```
Aurobindos-MacBook-Pro-2:.ssh aurobindosarkar$ ssh -i
~/.ssh/AWSBook2EdKeyPair.pem ubuntu@34.215.190.229
The authenticity of host '34.215.190.229 (34.215.190.229)' can't be
```

```
established.
ECDSA key fingerprint is
SHA256:GS+9/SSDhI4rI81hEIi14ujeeyMJpMmt49DjCsu+DDU.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '34.215.190.229' (ECDSA) to the list of
known hosts.
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.4.0-1022-aws x86_64)

* Documentation: https://help.ubuntu.com
* Management:     https://landscape.canonical.com
* Support:       https://ubuntu.com/advantage

 Get cloud support with Ubuntu Advantage Cloud Guest:
    http://www.ubuntu.com/business/services/cloud

0 packages can be updated.
0 updates are security updates.

The programs included with the Ubuntu system are free software; the
exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.
Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted
by applicable law.

To run a command as administrator (user "root"), use "sudo
<command>". See "man sudo_root" for details.

ubuntu@ip-172-31-18-70:~$
```

- **Installing software**: The next step is to install Apache Tomcat and MySQL client libraries on to the EC2 instance. First update the package repositories and then install the packages.

```
sudo apt-get update;
sudo apt-get install tomcat7 mysql-client-5.7;
```

- **Verify Tomcat7 installation**: Open any browser and type in http://34.215.190.229:8080 . You will see a default Apache Tomcat page on the browser.
- **Verify MySQL access from EC2 instance**: From the EC2 instance command line, type the endpoint (URL) from RDS instance dashboard.

```
mysql -u a1dbroot -p -h a1ecommerce.cpyfddnufvjm.us-
west-2.rds.amazonaws.com -P 3306
```

When prompted for the password, type in the password you entered while creating the DB instance. At the end of it, you should see a MySQL command prompt.

Repeat the earlier-mentioned step from your development machine. This verifies the DB instance is accessible from both the EC2 instance and your development machine.

Now, we have the DB instance configured, and the Tomcat web server is up and running, all we need to do next is to deploy the a1ecommerce application on EC2 instance. But this needs a minor modification in the configuration file for the code to point to the correct database instance (now on the AWS cloud instead being in of our development machine).

- **Change the database endpoint**: Check out the latest version from GitHub, and in the data-access.properties file in `src/main/resources/spring` folder change the following properties to:

```
jdbc.url=jdbc:mysql://a1ecommerce.cpyfddnufvjm.us-
west-2.rds.amazonaws.com:3306/a1ecommerceDb

jdbc.username=a1dbroot # username of Amazon DB instance
jdbc.password=a1dbroot #Password for the Amazon DB instance
```

After modifying the data-access.properties file, it is time to build the project and copy the war file to the EC2 instance for deployment. From the root of the project,

```
mvn package
```

This will create `a1ecommerce.war` file in the target folder copy this to the EC2 instance for deployment.

```
scp -i ~/.ssh/AWSBook2EdKeyPair.pem ./target/a1ecommerce.war
ubuntu@34.215.190.229:~/
```

This copies the `a1ecommerce.war` file from the target folder to the home folder of `ubuntu` in the EC2 instance.

```
sudo cp a1ecommerce.war /var/lib/tomcat7/webapps
```

Next, we ssh into the EC2 instance and copy the file to the Tomcat webapps folder. This will deploy the war file to the Apache Tomcat webserver.

At this stage, you have successfully completed deploying a web application on the Amazon cloud. To verify, from the browser, type in the `http://34.215.190.229:8080/a1ecommerce`, and you should see the A1Electronics ecommerce site up and running.

Summary

In this chapter, we described the main AWS services that are most commonly used for AWS cloud applications development. These included compute, storage and content delivery, databases, networking, application, administration, analytics, machine learning and deployment services. Next, we described some strategies to lower your cloud infrastructure bills. We also explained the purpose and characteristics of environments that are typically provisioned for cloud development. Finally, we walked you through the process of provisioning the AWS development infrastructure for our sample application.

In the next chapter, we will focus our attention on how you can design and implement application scalability on AWS cloud. We will describe some design patterns to achieving application scalability. Next, we will describe the AWS autoscaling feature and how to select the best set of rules for configuring it. Finally, we will implement some of these design patterns in our sample application and implement the autoscaling rules.

4

Designing for and Implementing Scalability

In this chapter, we will introduce key design principles and approaches to achieving scalability in applications deployed on the AWS cloud. As an enterprise or a start-up at its inflection point, you never want your customers to be greeted with a 503 message (that is, Temporarily Unavailable). The approaches in this chapter will ensure your web and mobile applications scale effectively to meet your demand patterns, growth in business, and spikes in traffic. We will also show you how to set up auto scaling in order to automate the scalability in the sample application.

In this chapter, you shall learn about:

- Defining scalability objectives
- Designing scalable application architectures
- Leveraging AWS infrastructure services for scalability
- Setting up auto scaling for your deployed application

Defining scalability objectives

Achieving scalability requires the underlying application architecture to be scalable in order to fully leverage the highly scalable infrastructure services provided by AWS cloud.

Setting scalability objectives for an application will depend on many factors, such as the nature of the application, the number of users (peak and average), the growth rate, the business model (subscription based, free, paid, or freemium model), SLAs, the nature of customers (businesses or the general public), and so on. It is very important to set some scalability objectives, however, as operating on the cloud doesn't require you to be absolutely accurate about hard-to-estimate parameters. On the cloud, you can always respond quickly to your rapidly evolving business. So, while it is good to have sufficient information to guide you initially, you don't need to spend excessive time and effort trying to arrive at very accurate estimates.

At a high level, your application should respond proportionally to the increase in the resources consumed, and be operationally efficient and cost-effective. For example, executing a "lift-and-shift" strategy for migrating your on-premises applications to the cloud, followed by a traditional approach of increasing the sizes of your instances to meet increasing loads, will likely become very expensive. And your application's increasing resource requirements will necessitate another move to even larger instances. In most cases, you will obtain the best results by splitting your application into smaller components, and then optimizing them using AWS infrastructural features and best practices.

In all cases, an underlying objective is to always try and squeeze as much performance out of each service/component as possible before scaling and spending additional dollars.

Designing scalable application architectures

In this section, we present some of the common approaches to designing scalable application architectures. Some of these design principles are not unique to cloud-based applications, however, they become even more important in a cloud context. Let's review a few of these design principles in the following sections.

Using AWS services for out-of-the-box scalability

One of the simplest guidelines to follow is leveraging AWS PaaS services wherever possible to enjoy the benefits of scalability and availability out of the box, without the associated administrative headaches or design complexity. Don't reinvent the wheel when ready-to-use services such as email, queuing, search, databases, monitoring, metrics, logging, and so on, are available to you from Amazon or other third-party vendors. For example, you can leverage the RDS or the DynamoDB services available for scalable relational and NoSQL database services, respectively. Similarly, you can leverage the AWS SQS service, as it offers a multi-AZ, scalability (unlimited messages), and a secure queuing service accessible via simple APIs without having to roll out your own implementation or managing an open-source product deployed on an EC2 cluster.

There are several standardized and highly scalable monitoring, metrics, and logging services available—use them. Additionally, look for opportunities to get rid of server-based computing and use AWS Lambda instead. Lambda can be used to implement event-driven computing functionalities using languages such as JavaScript, Java, and Python.

Using a scale-out approach

Designing an application that can scale horizontally allows you to distribute application components, partition your data, and follow a services-oriented design strategy. This approach will help you better leverage the elasticity of the AWS cloud infrastructure. For example, you can choose the right sizes and number of EC2 instances, automatically and on demand, to meet your varying demands.

Implementing loosely-coupled components

Loosely-coupled applications are typically implemented using message-oriented architecture. The more loosely coupled the application components are, the better they will scale. Design your application to comprise of independent components. Design everything as a black box and decouple interactions to the extent possible. You can use the AWS SQS service for this purpose. SQS queues are commonly introduced between application components to buffer messages. This ensures that the application is functional under high concurrency, unpredictable loads, and/or load spikes.

Loosely-coupled components enable you to differentially scale out your architecture by deploying more instances of any given component or by provisioning more powerful instances for the components that require it. You can also provision specialized EC2 instances to meet the specific requirements of your components or use cases, for example, computing optimized, memory optimized, and/or storage optimized instances.

In addition, you should try to design your application components to be stateless services, as much as possible. This will help you distribute your components more effectively. With a **Service-Oriented Architecture** (**SOA**) you can move services into their own tiers, treat them separately, and scale them independently. This also offers greater flexibility and understanding of each component. In situations where you need to store session states, it is important that you do so outside of your component, so that it is accessible from any instance serving your users' requests. This is especially important in the auto scaling context, where the number of instances are varying in response to demand.

Loose coupling plus SOA is a winning architectural combination whether on-premises or on the cloud, so spend the time and effort to architect your applications according to their key guiding principles.

Implementing asynchronous processing

Implementing asynchronous processing in your application can improve scalability. This is typically done using AWS SQS queues. However, ensure that you implement a dead letter queue for queue requests that fail after several retries (usually between three and five times). You can use the AWS SNS service for notifying components when a message's request processing has been completed. You can also create asynchronous pipelines for data flows within your application using AWS Kinesis data streams, and AWS SQS queues where you can route your data to different queues to be processed differently.

Leveraging AWS infrastructure services for scalability

In this section, we will shift our focus to the strategies you can use to leverage the AWS cloud infrastructure to scale your applications.

Using AWS CloudFront to distribute content

Try to offload as much content as possible to the AWS CloudFront CDN service for distribution to Amazon edge locations. This can include both static and dynamic content. For example, static content or files would include CSS, HTML, images, and so on, that are stored in Amazon S3 (and not on your web server instance). This can reduce load on your web servers and improve the efficiency of maintaining content (by storing at one S3 location) while reducing latency for your end users and overall cost (by reducing the size or the number of EC2 instances required for your web servers).

In the case of dynamic content, for example, the repeated queries from many different users resulting in the same content response from your servers are cached and served up from the edge locations. This results in deriving similar benefits as in the case of static content distribution. This approach can be especially useful in speeding up responses to mobile apps.

Using AWS ELB to scale without service interruptions

Configure an AWS ELB in your deployment architecture even if you are using a single EC2 instance behind it. This will ensure you are ready to scale up or down without interrupting your services. ELB ensures the CNAME application access point remains the same, as you auto scale the number of servers or even replace a fleet of servers behind it. This can also help you systematically roll out new versions of your application behind the ELB without service interruptions for your customers.

ELBs work within an AWS region only, however, they can work across multiple AZs within a region. Typically, for very large installations where you might want to distribute traffic across multiple AWS regions, you would use the Route53 service.

Using Amazon CloudWatch for Auto Scaling

Amazon CloudWatch is a web service that enables you to monitor and manage various metrics, and configure alarm actions based on the metrics. A metric is a variable that you want to monitor, for example, CPU usage or incoming network traffic. A CloudWatch alarm is an object that monitors a single metric over a specific period. The alarm changes its state when the value of the metric breaches a defined range and maintains the change for a specified number of periods.

Amazon CloudWatch can aggregate metrics across pre-defined dimensions, for example, aggregating the CPU utilization of all EC2 instances in an Auto Scaling group. The alarm connected to an Auto Scaling policy triggers a scaling event, and the number of instances are increased or decreased as per the defined policy. For these simple scaling policies, the reaction is always the same independent of the size of the breach. Such policies lock the Auto Scaling group while a scaling action is running, and no further alarms can be raised during this time. Additionally, it evaluates the metric only when no scaling action is ongoing. However, such policies do not let you control how aggressively you want to react to the breaches of your metrics.

We can also define Step Scaling policies in which we can define multiple steps in the same policy. The appropriate scaling step is selected based on the value of the metric that triggered the alarm, based on the magnitude of the breach. In these policies, the metrics are continuously evaluated and it does not *lock* the Auto Scaling group while the action is evaluated. You can also use additional metrics and define multiple thresholds to specify the best strategy for your use cases. Overall, Step Scaling policies are the better and more flexible choice.

Aside from the scaling policies, you can also leverage the Instance Lifecycle hooks for finer grained control for launching and terminating instances. These hooks are useful in supporting common use cases including assigning EIP address on launch, registering new instances with DNS, gathering log files before instances are terminated, investigating issues with an instance before terminating it, and so on.

In situations of rolling deployments across Auto Scaling groups, you can terminate the instances one by one, followed by relaunching the instances (the new instances will launch with the new configurations enabled).

Scaling data services

There are several options for data services optimized for specific use cases available from AWS. Choose the most appropriate one for your application needs. For example, you can choose the RDS service for using MySQL databases, and create read replicas for use in your reporting applications. Read replicas not only serve your application needs efficiently but also help you reduce the size and number of RDS instances required. Similarly, you can exploit AWS ElastiCache to further offload requests that need to be served by your master RDS instance. In many applications, a vast majority of database requests (as high as 80 to 90%) can be serviced from ElastiCache.

Remember to monitor the utilization of your RDS using AWS CloudWatch to tune your instance sizes. In chatty applications, it can also help to offload some of your data from RDS to low-latency AWS DynamoDB with ElastiCache to further reduce the cost of RDS usage.

Scaling proactively

You can proactively scale your applications in response to known traffic patterns or special events. For example, if you have cyclical patterns (daily, weekly, or monthly) in the usage of your application, then you can leverage that information to scale up or down the number of instances at the appropriate time to handle the increase or decrease in demand, respectively. You can also rapidly scale up just minutes in advance of special events, such as flash sales or in response to some major breaking news, to handle a huge surge in traffic.

Using the EC2 container service

For building and running distributed applications, we need think in terms of using a different primitive task (that you can move around) and not servers or machines. Currently, you may be thinking about a pool of resources or EC2 instances. However, you need to change that to thinking about where you can take a task and drop it in so that it will run appropriately. ECS helps in distributing applications/microservices over a cluster with managed task life cycle management. It has tight integration with other AWS services (IAM, ELB) and can run multiple schedulers with prioritization (for example, jobs from different teams with different priorities). It supports Auto Scaling groups where we can scale tasks by running more copies of them to take on increased traffic. They can be behind an ELB and appropriate metrics can be used to scale your fleet. Service updates are a lot easier to manage as well.



Refer to the extensive documentation available from Amazon for architectural blueprints, technical blogs, white papers, and videos containing in-depth guidance on effective scalability strategies to follow for each of the AWS services.

Evolving architecture against increasing loads

Auto scaling is not the single thing that fixes everything. In real life, you will probably evolve your architecture against an increasing number of users or load. In this section, we will suggest actions you can take at each stage of your growth starting from a handful of users right up to tens of millions of users for a typical web stack.

Scaling from one to half a million users

In the beginning, you can get started with a single EC2 instance that hosts your web service and the DB on the same instance. You can provision an EIP and use Route53 for DNS services. This should be sufficient to handle a typical website or service for a new business.

As your number of users increases to several hundreds or thousands, there are several easy-to-upgrade options available to scale your infrastructure. The simplest first step is to get a bigger EC2 instance (scale vertically). You can also leverage instance types (high I/O, memory, compute, storage intensive) based on your specific workloads. Additionally, increasing PIOP settings can give you the results you need to handle an increasing number of users. However, be aware that you will hit an upper limit in terms of how far you can scale this way. Additionally, having no redundancy or failover mechanisms as the number of users increases will become a major cause for worry.

The first change to the architecture is to separate the DB from the web tier. This step can instantly improve overall scalability, as it enables the web and data tiers to be independently scaled. Web and data tiers don't have the same requirements, so having the ability to scale them separately gives you more flexibility to align the architecture more closely to your workloads.

Typically, you would start with a relational database because you are probably most familiar and comfortable with it. It is a well established and well-worn technology; there is lots of existing sample code available, active communities, many reference books, and tools to help you with anything you are likely to face. For most workloads, you aren't going to break RDBMS down to several million users. There are well-documented patterns for scaling relational databases.

However, there may be exceptions where relational databases are not suitable for your first year of operations, and deploying a NoSQL database makes more sense. These use cases could be due to massive data volumes (>5 TB) expected in the first year itself. Other reasons for deploying a NoSQL solution could be due to requirements such as super low-latency use cases, highly non-relational data, or unstructured data requiring schema-less data constructs, rapid ingestion of streaming data (thousands of records per second), and so on. In many applications, the functional and non-functional requirements will most likely be best served by having both SQL and NoSQL databases in your application.

At this stage, you will also need to decide between the self-managed database (on Amazon EC2) or a fully managed AWS database service (such as RDS, DynamoDB, Redshift, and Aurora). Though it is likely that you have sufficient familiarity with setting up and managing a database like MySQL, you may want to consider a MySQL-compatible RDS service such as Aurora instead to avoid managing your own database servers. Aurora automatically scales storage up to 64 TB and can have 15 read replicas currently. Additionally, the service comes with continuous (incremental) backups to Amazon S3 and six-way replication across three AZs. It is an economical, performant, and scalable alternative.

As the number of users goes over a thousand, you will want to consider a more distributed architecture for higher scalability and High Availability. For example, consider distributing the web tier across two AZs, and a multi-AZ DB deployment to enable replicated DBs across the two AZs. You can also include a classic elastic load balancer to distribute the load evenly and get HA out of the box. The load balancing service scales automatically and performs health checks on the instances registered with it. The ELB won't send traffic to unhealthy instances. You can also consider content-based routing with an application load balancer. Using these basic approaches can get us pretty far, especially with the ELB helping us to scale horizontally and upgrading instance types to scale vertically.

As we move to handling traffic for tens of thousands of users, we can consider additional load balancing for various services and instances across multiple AZs. We can now add read replicas to reduce load on the master DB and allow more scalability in the data tier.

As you scale, don't lose sight of performance, efficiency, and costs involved. Consider lightening the load on your origin servers by leveraging CloudFront and S3—take the static content and put it in S3 buckets and front them with CloudFront. Amazon S3 object-based storage is an economical storage option that is great for static assets, is infinitely scalable, and can store objects up to 5 TB in size with optional encryption. CloudFront can cache content for faster delivery, lowers load on the origin, and can deal with both static and dynamic content. It also supports custom SSL certificates and low TTLs (as short as zero seconds, optimizing connection to the origin).

The service is optimized for AWS. With CloudFront, the ability to scale improves, reduces load on the origin, and improves response times significantly. These steps can make the web tier a lot more lightweight.

We can introduce DynamoDB at this stage. DynamoDB is a managed NoSQL database that is fully distributed and fault tolerant and supports provisioned throughput. You can provision the reads separately from the writes, and achieve fast predictable performance. You can also use DynamoDB to store session date data to create a stateless application. If the state information is removed from the web tier, it greatly increases your ability to scale it while maintaining a good customer experience.

We should also start using ElastiCache at this stage. ElastiCache is a managed Memcached or Redis service that can scale from one to many nodes. It is a self-healing (replaces dead instances) and performant (single digit ms speeds) caching service. Note that the Memcached option is local to a single AZ, however, multi-AZ configurations are possible with Redis. For high access queries, you should not keep going back to the data tier. You can store the results in the cache instead for better user experience and scale the data tier in a more reasonable manner.

Scaling from half a million to a million users

As we cross half a million users, we definitely need to implement auto scaling for the automatic resizing of compute clusters. At this stage, we need to know our requirements in terms of defining min/max pool sizes, the CloudWatch metrics to drive scaling, and using on demand and Spot instances in our auto scaling groups. Over provisioning resources for peak periods will get expensive at these volumes. You can create an auto scaling group for web servers in the web tier across the AZs (if you have been following the suggestions in the sections, the rest of the architectural components are managed services, and scaling them is largely Amazon's responsibility).

At the most basic level, we need to be clear on three parameters for auto scaling: the minimum, maximum, and the desired number of instances. We should always keep the minimum number of instances running, and launch or terminate new instances to meet desired capacity. As a practice, we never start more than a maximum number of instances, and as much as possible we keep the instances balanced across the AZs.

We will need to define launch configurations to determine what is going to be launched (EC2 instance type and size) and the AMI to be used along with the security groups, SSH keys, IAM instance profile, and users' data (essentially any arbitrary data). Bootstrapping the infrastructure requires the installation and setup to be fully automated. We use an AMI with all the required software and configurations specified. These configurations can be specified via user data, or using tools like Chef/Puppet/Ansible and AWS CodeDeploy.

If we are terminating an instance due to reduced traffic then we also need to de-register the instance from the ELB, select a target instance, and then terminate it. Which instance is terminated can be configured using termination policies, and the service will try to terminate the instance in a manner that balances the capacities across the AZs. Termination policies determine which instances are terminated first. There are several options such as the longest running, the one having the oldest launch configuration, and the closest to the full billing hour (for better cost control, as it gets the most value out of the terminating instance).

Scaling plans determine when an Auto Scaling group scales in or scales out. If the desired capacity is greater than the current capacity then we launch new instances, and if the desired capacity is less than the current capacity then we terminate instances.

There are different types of scaling plans. The default plan ensures that the current capacity of healthy instances remains within boundaries (never less than the minimum). We can also modify the desired capacity (via the API, console, or CLI) to trigger a scaling event. This type of *manual* scaling is helpful for testing, and can be set and changed through scripts based on requirements at any given point. A scheduled scale in/scale out plan is based on timed events, for example, in order to align with extended business hours, we can have a plan that scales up at 8 A.M. and scales down at 8 P.M. We can also do dynamic scaling that scales based on Amazon CloudWatch metrics and thresholds.

You should also configure the Auto Scaling groups to work with one or more ELB. This integration will enable the registering of new instances and deregistering instances on termination automatically. Furthermore, we can use ELB metrics when defining our auto scaling policies.

Scaling policies determine when to change capacity. The capacity can be changed in different ways. For example, setting a fixed capacity as the desired capacity and adding/removing a fixed number of instances or adding/removing a percentage of existing capacity. Similarly, dynamic scaling policies trigger scaling events based on demand. The demand is measured based on metrics and the changes in metrics can be mapped to scaling policies.

Scaling from a million to ten million users

As you approach a million users, you are going to have to implement further along the lines of what we have discussed so far including multi-AZ deployments, elastic load balancing between tiers, auto scaling, SOA, serving content smartly (S3, CloudFront), caching DB data, and moving state information off tiers that auto scale.

As you grow your user base in the range of 5 to 10 million you will need to focus a lot more on the data tier. This would be the time to think about database federation, that is, splitting the DB into multiple DBs based on the function/purpose, sharding (splitting a dataset into multiple parts), and moving some of the functionalities to other types of specialized DBs (such as NoSQL and graph databases).

However, note that database federation will make it harder to do cross-functional queries and it will not help in situations where you have individual functions or tables that are huge. In some ways, database federation is essentially delaying sharding and/or a shift to using NoSQL databases. Sharding brings in horizontal scaling but makes it more complex at the application level. This approach essentially has no practical limit on scalability. Typically, the data is sharded by function or key space; and the strategy can be implemented with both RDBMS and NoSQL databases. However, shifting over to using NoSQL databases solves the scale problem in the longer term. By leveraging managed services like DynamoDB, the move to NoSQL databases can be a lot simpler than managing your own NoSQL database clusters.

Moving to beyond ten million users requires you to deeply analyze your entire stack, and possibly look at custom solutions to solve your scalability issues. You should explore more fine-tuning of your application, move to services architecture for most of the features/functionalities, move from multi-AZ to a multi-region deployment, and look for opportunities to leverage AWS ECS and AWS Lambda services wherever you can.

Event handling at scale

For real-time streaming applications, supporting hundreds of thousands of concurrent users in a reliable, secure, and auditable manner that is also cost effective requires different design considerations and additional AWS services to be implemented in a scalable manner. In such applications, you may need to plan for growth from one thousand events currently to a future of 15 million events per second. In the following sections, we will explore typical AWS services used for implementing a large-scale API-based architecture and analyzing streaming data in real time with Amazon Kinesis Analytics.

Implementing a large-scale API-based architecture with AWS services

In the upcoming sections, we will consider AWS services that can be leveraged for implementing a large-scale API-based architecture. These services include Amazon API Gateway, AWS Lambda, Amazon S3, Elasticsearch, RDS, DynamoDB, CloudWatch, and others. As we have covered services such as Amazon S3, RDS, and DynamoDB previously, we will focus more on some of the other services used.

Using Amazon API Gateway

Amazon API Gateway is a fully managed service used for hosting HTTPS APIs on top of AWS. This service helps in creating, publishing, maintaining, monitoring, and securing APIs. It supports standard HTTP methods, and you can authenticate and authorize requests using services such as IAM and Conginito. Amazon API Gateway provides highly scalable parallel processing, DDoS protection, and features for throttling, metering, and capping usage for backend systems.

Its benefits include the ability to create a unified API frontend to multiple microservices and supporting multiple versions of the APIs (as we iterate through dev, test, and release of APIs). CloudFront distribution can created for the API at no extra charge, and the API gateway can be set up to cache responses. The backend can be implemented as a Lambda function or a HTTP endpoint, and it can be integrated with CloudWatch for monitoring.

Compared to implementing code for HTTP/HTTPS yourself, the API Gateway is a highly integrated service that scales automatically and can handle thousands of concurrent calls, while providing support for authorization, access control, monitoring, and API version management.

Using AWS Lambda

AWS Lambda is a serverless, event-driven compute service that runs your function code without you having to manage or scale servers. It provides an API to trigger the execution of your functions and ensures that the function are executed in parallel, regardless of scale. It provides additional capabilities for your functions such as logging and monitoring. Lambda functions are essentially stateless, trigger-based code execution. For example, the streaming events could be landing on S3 or DynamoDB, or they could be an API Gateway call, or even a scheduled job. These functions can access services inside or outside your VPC.

Compared to provisioning and managing EC2 instances yourself, using the Lambda service means no servers or instances to manage and it comes with built-in scaling and a fixed cost model. However, as AWS Lambda is a relatively new service, you will have limited experience with it, and you will be limited to using CloudWatch for monitoring (at this time). Additionally, there is a 6 MB data limit on Lambda functions, and debugging the logs can be a very time-consuming task.

Using Kinesis Streams

Kinesis Streams is a fully managed service for real-time processing of high-volume streaming data. It processes data in real time, and enables highly scalable parallel processing. There are source libraries for sending data to and reading data from a stream. It synchronously replicates your data across three facilities, and is integrated with many AWS and third-party services. It also supports SSL and automatic encryption of data once it is uploaded.

AWS Kinesis makes large-scale data ingestion a lot simpler. It is easy to administer. You have to essentially create a new stream and set the desired capacity in terms of shards. You can scale your capacity to match your data throughput rates and expected volumes. It is typically used to build real-time cloud-based applications, for example, the continuous processing of streaming log data. It provides a cost-efficient mechanism for streaming workloads of any scale.

With Kinesis you can process very high volumes of streaming data (currently up to 15 million records at peak load and growing). As a managed service you also don't have to predict storage and volume requirements as you would have to do with Apache Kafka and ZooKeeper clusters. It also comes with cross AZ replication. However, remember to configure the max data retention settings (from 24 hours to 7 days) or you risk losing your records.

Using Elasticsearch

Elasticsearch is a powerful real-time, distributed, open source search and analytics engine built on top of Apache Lucene, a schema-free and developer-friendly RESTful API. Amazon Elasticsearch Service is a managed service that makes it easy to set up, operate, and scale Elasticsearch clusters in the cloud. It comes with built-in Kibana and Logstash plugins.

You can modify clusters with no downtime, and it can be integrated with many AWS services including CloudWatch, Lambda, DynamoDB, and so on. It supports the Elasticsearch API and is a drop-in replacement for your existing Elasticsearch clusters. In addition, you only pay for what you use. It provides simple interfaces for cluster creation and configuration management. These clusters are self-healing clusters, HA (clusters are AZ aware and can spread to multiple AZs), and have high data durability.

Compared with Lucene and Elasticsearch company's offering, Amazon Elasticsearch is a managed service that provides out of the box integration with Amazon Kinesis and S3. However, note that Amazon's version is typically a release behind the official Elasticsearch release, so it may not have the latest features available.

Analyzing streaming data in real time with Amazon Kinesis Analytics

Most data is produced continuously. For example, mobile apps, logs, IoT sensors, and so on generate data at a furious pace. Recent data is considered highly valuable, if you act on it in time. These insights can diminish in value or perish with the passage of time. A different set of tools for collecting and analyzing real-time data is required for implementing such applications. The fast pace and variable rates (bursts) of incoming data need to be stored durably and processed correctly, in a continuous, fast, and reliable manner. Typical use cases for such processing include time series analytics, feeding real-time metrics, and generating real-time alarms and notifications.

Amazon Kinesis Stream makes it easy to work with real-time streaming data. You can reliably ingest and durably store streaming data at a low cost. Additionally, you can build custom real-time applications to process streaming data. It also provides the ability to scale using a configurable number of shards. Shards give you a certain amount of throughput, for example, a single shard gives you 1,000 writes per second or 1 MB of total ingestion. Increasing the number of shards to 10 shards will give you approximately 10,000 writes per second.

Using Amazon Kinesis Firehose

Kinesis Firehose lets you load massive volumes of streaming data. It provides a means to reliably ingest and deliver batched, compressed, and encrypted data to S3, Amazon S3, Amazon Redshift, and Amazon Elasticsearch Service (Amazon ES) destinations. A UI driven point-and-click setup with zero administration makes it simpler for the developers to leverage the seamless elasticity of Kinesis Streams under the hood.

Using Amazon Kinesis Analytics

Amazon Kinesis Analytics enables the analysis of data streams using standard SQL queries. It lets you interact with streaming data in real time using SQL and to build fully managed and elastic stream processing applications that process data for real-time visualizations and alarms.

Building real-time applications with Amazon Kinesis Analytics

Typically, it is a three step process to build real-time applications:

1. **Connect to the streaming source**: Streaming data sources include Amazon Kinesis Firehose or Amazon Kinesis Streams. Input formats supported include JSON, CSV, variable column, or unstructured text. Each input has a schema that can be automatically inferred but you can also manually edit the schema. Ensure you carefully review and test inferred input schema. You might need to manually update the schema to handle nested JSON with greater than two levels of depth.
2. **Writing the SQL code**: Build streaming applications with SQL statements. It provides robust SQL support and advanced analytic functions out of the box. Additionally, it provides extensions to the SQL standard that work seamlessly with streaming data. It has built-in support for at-least-once processing semantics. Best practices include avoiding time-based windows of greater than one hour and using smaller SQL queries, with multiple in-application streams, rather than a single, large query.
3. **Continuously deliver SQL results**: You can send processed data to multiple destinations: S3, Redshift, AWS ES (through Firehose), and Streams (with AWS Lambda integration for custom destinations). It gives you end-to-end processing speeds that are in the sub-second range (depending on the query).

Additionally, you can also reference data sources (S3) for data enrichment purposes. As a practice, you should limit the number of applications reading from same source to avoid exceeding the provisioned throughput. For example, for an Amazon Kinesis Streams source, limit the total number of applications to two applications, and for Amazon Kinesis Firehose, limit it to a single application.

You can set up CloudWatch alarms to track how far behind the application is from the source and raise alarms accordingly. You can also increase input parallelism to improve the performance. For example, if the application is not keeping up with the input stream, then consider increasing input parallelism to create multiple source in-application streams.

Setting up Auto Scaling

This section introduces you to dynamic scaling for your deployed application. As explained, the application will scale out; more EC2 instances will be added or scaled in. That is, the running of EC2 instances will be removed, based on some measurable metric. We will select the metric from a defined set and apply rules so that our Auto Scaling can scale in or out based on these rules.

AWS Auto Scaling construction

To implement AWS Auto Scaling, we will create an Elastic Load Balancer (ELB), a base AMI which will be an EC2 instance running our e-commerce application, a launch configuration (the base AMI to launch in an EC2 instance), CloudWatch alarms to add/remove instances that apply to an Auto Scaling group, and finally an Auto Scaling group.

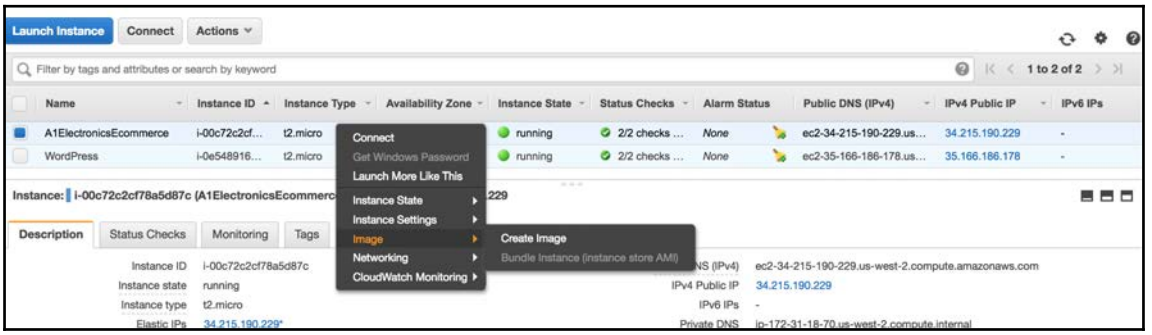
We will perform the steps in the following sections to implement auto scaling for our sample application.

Creating an AMI

An **Amazon Machine Image** (**AMI**) is a master image used for creating virtual servers on the Amazon cloud. An AMI contains instructions to launch an EC2 instance, and includes information pertaining to an operating system, a machine architecture of 32 bit or 64 bit, the software stack of your applications, launch permissions, disk sizes, and so on. Typically, you will start with a basic AMI provided by Amazon, the user community, or the marketplace, and then customize it as per your requirements. You can also create an AMI from a pre-existing EC2 instance.

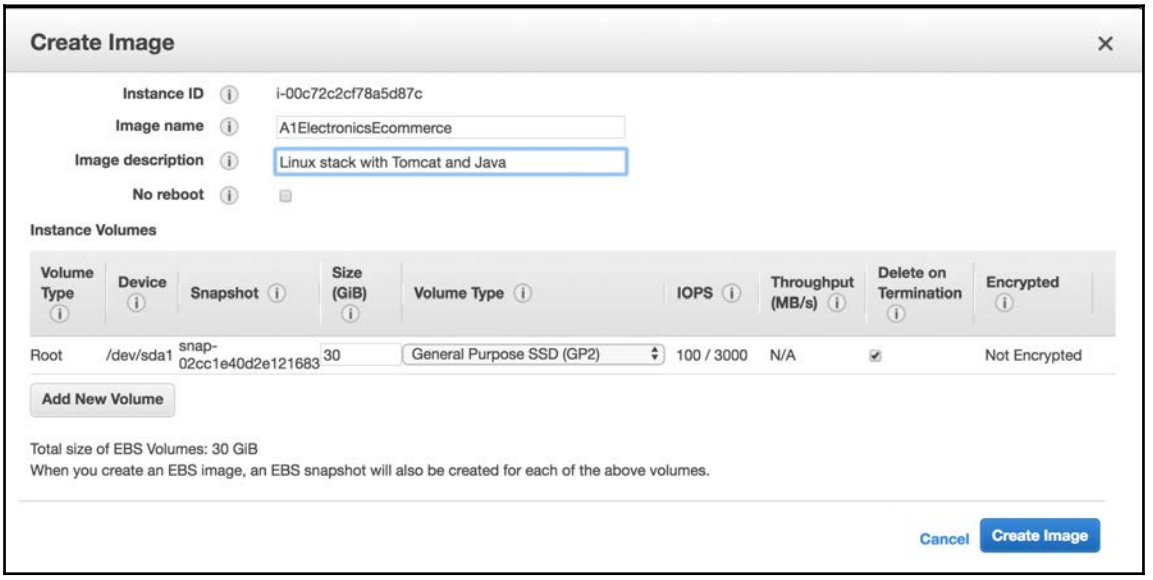
An AMI is a prerequisite for creating and using an auto scaling group. The way it works is that whenever a scale out is required, auto scaling uses the AMI to create an EC2 instance and adds it to the group. We will use the **A1ElectronicsEcommerce** instance that we created in `Chapter 3`, *AWS Components, Cost Models, and Application Development Environments*, to create the AMI:

1. From the EC2 navigation pane, click on **Instances** to view all your EC2 instances. Select the **A1ElectronicsEcommerce** instance and then right-click to view all the actions you can perform on the selected instance. Select **Image** and then click on the **Create Image** option from the menu to create an AMI:

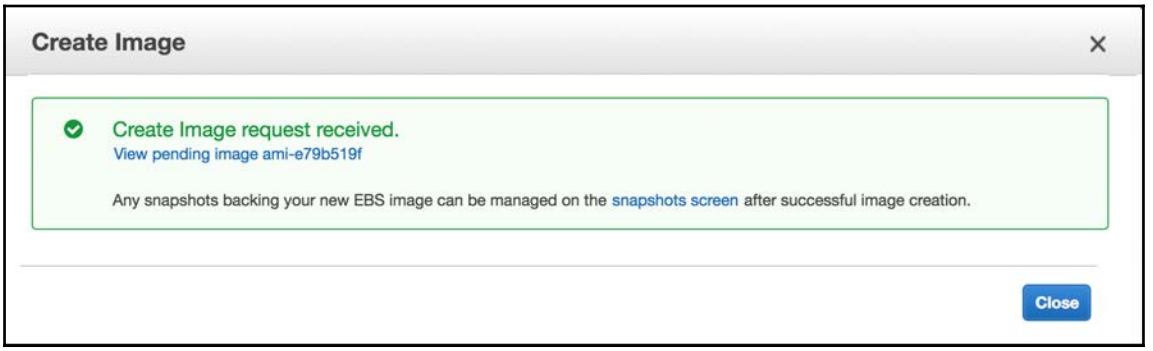


2. The next step is to name the AMI and allocate some disk space to it. On this screen, you only need to be aware of the following configuration parameters:
    - **No reboot:** By default, Amazon EC2 shuts down the instance and takes a snapshot of attached volumes and then creates and registers the AMI. If this option is checked then the EC2 instance will not shut down, and the integrity of the filesystem cannot be guaranteed while creating the AMI.
    - **Delete on Termination:** During Auto Scaling, the EC2 instance will be created and terminated depending on the configured metrics. During the launch of an EC2 instance, the EBS volumes will be created and referenced by the AMI (in our case, it is the **Root** volume), and when the EC2 instance is terminated the associated volume is not deleted, so over a period of time you could have many EBS volumes for which you are unnecessarily paying. As our application is stateless and does not store any user data on the EBS volume, we can safely delete the EBS volume at the same time as the instance.

Now, click on the **Create Image** button:



3. You should see the following message. Click on the **Close** button:



You should see the AMI listed with the **Status** as **available**, which you can see in the following screenshot:



In the next step, we will create the ELB for our application.

Creating the Elastic Load Balancer

An ELB distributes the incoming requests from the internet/intranet to the EC2 instances registered with it. The elastic load balancer can distribute the requests to the instances in a round-robin manner. If you need more complex routing algorithms then either use the Amazon Route53 DNS service, use Nginx as a reverse proxy, or use HAProxy. Amazon ELB is designed to handle an unlimited number of concurrent requests per second with increasing loads. However, it is not designed to handle sudden spikes in the number of requests that typically occur during special promotional sales, online exams, or online trading, when you might experience sudden surges. If your use case falls into this category then you can request that the Amazon Web Service support team pre-warm the ELBs to handle the sudden load increases.

The ELB consists of two parts :

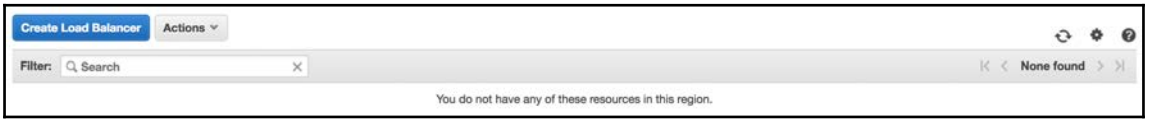
- **Load balancer**: Monitors and handles the requests coming in through the internet/intranet, and distributes them across the EC2 instances registered with it.
- **Control service**: It automatically scales the handling capacity in response to incoming traffic by adding or removing load balancers, as needed, and it also performs the health checks.

Before creating an ELB, we modify the EC2 security group created in `Chapter 3`, *AWS Components, Cost Models, and Application Development Environments*, to allow HTTP traffic on port 80 by adding an inbound rule, as shown in the following screenshot:

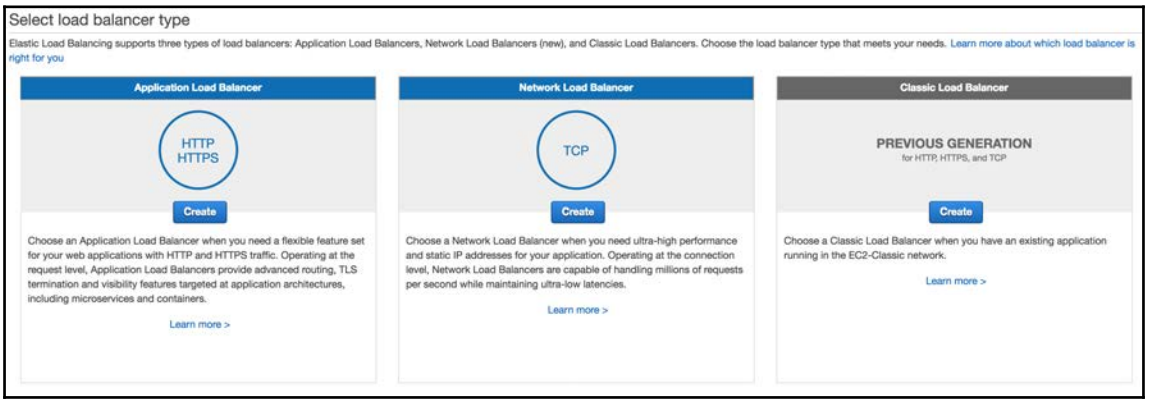


Follow the steps below to create the ELB:

1. From the EC2 navigation pane, click on **Load Balancers** listed under **Network & Security**, and then on the **Create Load Balancer** button:



2. The first step is to select the type of load balancer we want. At this stage, for simplicity, we will create a **Classic load balancer**. Click on the **Create** button:



We name the load balancer and configure the protocols it will service:

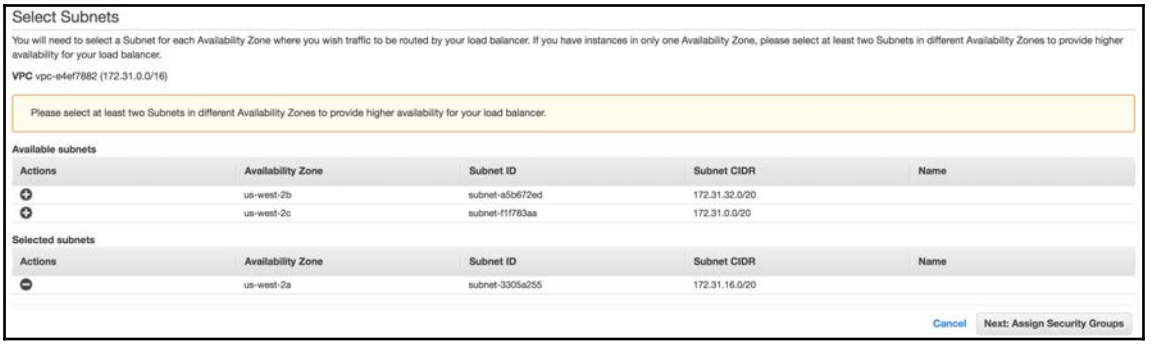
- **Load Balancer name**: A name which uniquely identifies the load balancer. This name will be a part of the public DNS name of your load balancer.
- **Create LB Inside**: Allows you to select the VPC where this load balancer will be deployed. Since we have only one VPC, select the **My Default VPC (172.31.0.0/16)**.
- **Create an internal load balancer**: A load balancer can be created to serve the internet traffic in a public subnet or for intranet traffic, that is, between your internal servers, such as between the web servers and application servers. Check this only when you want to create an internal load balancer.

- **Enable advanced VPC configuration**: The advanced VPC configuration option allows you to specify your own subnets. Select this option if you have created your own subnets and want to use them instead of the default subnets. This allows your load balancer to route traffic to EC2 instances in other Availability Zones.
- **Listener Configuration**: A listener is a process which listens for incoming requests on a specific port from the client end and relays it to an EC2 instance configured for the protocol and the port. It supports protocols both at transport layer (TCP/SSL) and application layer (HTTP/HTTPS). Our Apache Tomcat server listens on port 8080; we enter port 80 for the Load Balancer Port and 8080 for the Instance Port. The acceptable ports for both HTTPS/SSL and HTTP/TCP connections are `25`, `80`, `443`, `465`, `587`, and `1024-65535`. Select **HTTP** as the protocol both on the **Load Balancer Protocol** and **Instance Protocol**.

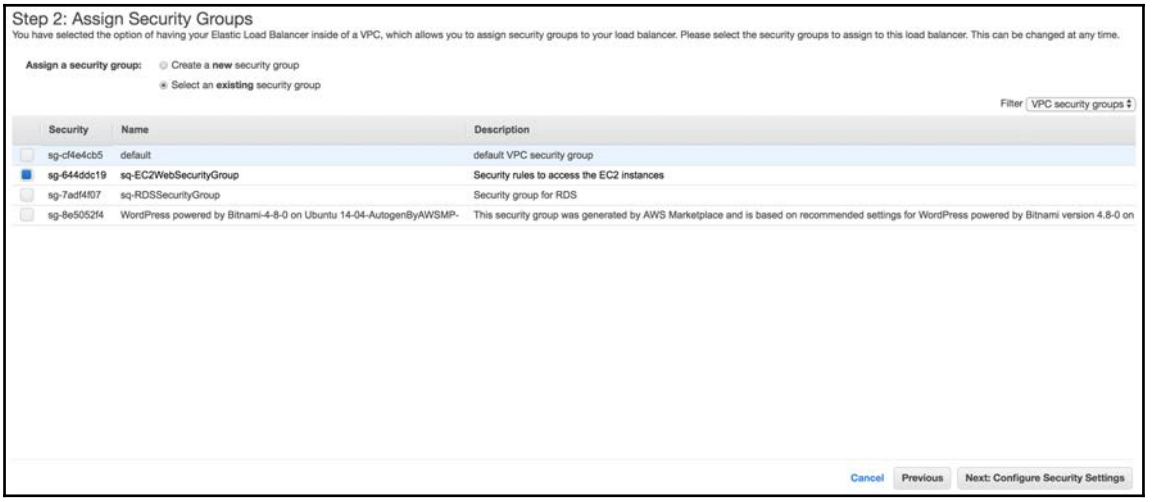
  The protocol for the load balancer and the instance should be at the same layer for a listener configuration. For example, if your load balancer protocol is using the TCP or SSL protocol then your instance protocol can either be TCP or SSL and not HTTP or HTTPS:



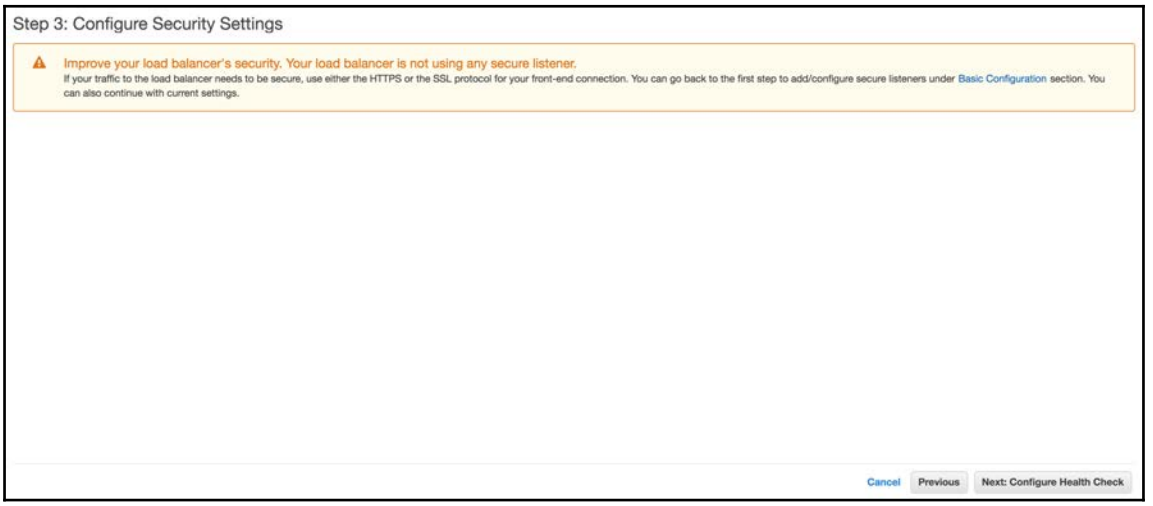
3. The next step is to configure the ELB to route the incoming traffic to the subnets in which the application is running. The VPC is configured so that each unique subnet is associated with an Availability Zone. The purpose of this is to make your application resistant to failure; if an Availability Zone goes down then all the application instances in that Availability Zone will not respond and will be detected via the ELB's health check. The ELB will start routing the data to your healthy application instances running in the other Availability Zones within the same region. Since our application is deployed in the **us-west-2a** Availability Zone, we add that to our **Selected subnets** by selecting it from the **Available subnets** list:



4. Now we assign a security group to our ELB. We have already created the **sq-EC2WebSecurity Group** security group in `Chapter 3`, *AWS Components, Cost Models, and Application Development Environments* (and updated it in this section). Next, click on the **Configure Security Settings** button:



5. You should see the following screen. Now, click on the **Next: Configure Health Check** button:



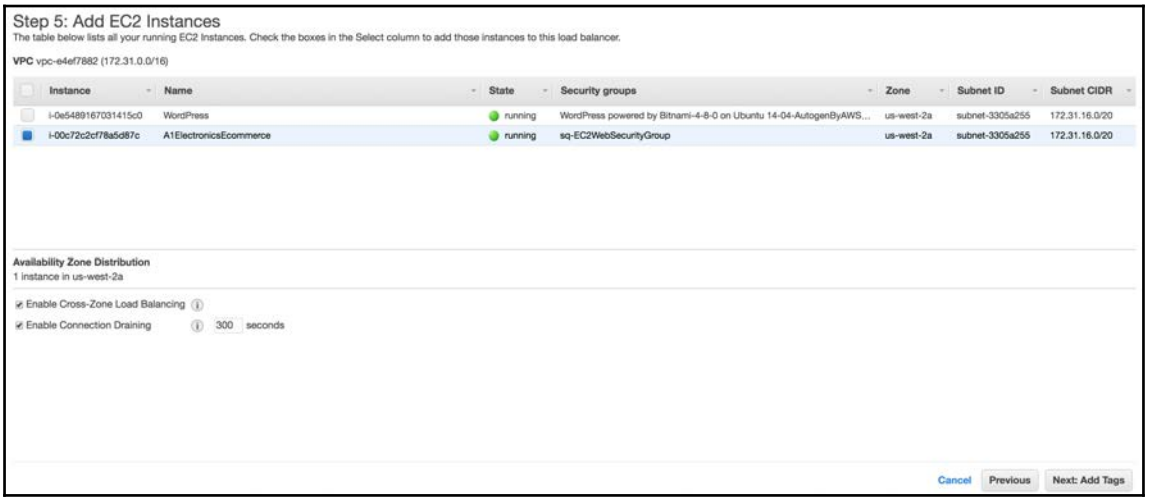
6. Next, we configure the health check. ELB periodically sends requests to test the availability of the EC2 instances. These tests are called health checks. EC2 instances that are healthy at the time of the health check are marked as **InService** and the instances that are unhealthy at the time of the health check are marked as **OutOfService**. The ELB performs health checks on all registered instances, regardless of whether the instance is in a healthy or unhealthy state. However, the ELB will route requests only to **InService** instances. In your web application, you define a URL which the ELB can call for a health check. To reduce network traffic we suggest you have a REST endpoint or a static HTML page which returns no data, only a 200 OK HTTP response code:
    - **Ping Protocol**: The protocol to connect to on the EC2 instance. It can be TCP, SSL or HTTP/HTTPS. Select **HTTP** from the dropdown.
    - **Ping Port**: The port to connect to with the instance. Enter 8080, which is the default port of our Apache Tomcat server.
    - **Ping Path**: The HTTP/HTTPS destination for the health request. A HTTP/HTTPS GET request is issued to the instance on the ping port and the ping path. If the ELB receives any response other than 200 OK within the response timeout period, the instance is considered unhealthy.

- **Response Timeout**: Time to wait when receiving a response from the health check. If the instance does not respond within the set time period it is considered unhealthy. Use the default value of `5` seconds.
- **Health Check Interval**: The amount of time in-between the health checks. If you have a low value, then you will increase the network traffic, but a healthy/unhealthy EC2 instance can be detected quickly and vice versa. Use the default value of `30` seconds.
- **Unhealthy Threshold**: Number of consecutive health check failures before declaring an EC2 instance unhealthy or **OutOfService**. An **OutOfService** EC2 instance will only be detected after a time period (in seconds) of HealthCheck Interval*Unhealthy Threshold. Use the default value, `2`.
- **Healthy Threshold**: Number of consecutive health check successes before declaring an EC2 instance healthy or **InService**. An InService EC2 instance will only be detected after a time period of HealthCheck Interval*Healthy Threshold seconds. Use the default value, `10`:



7. Next, we add the running instances to ELB. As we are creating this ELB for an Auto Scaling group we can skip this step. The Auto Scaling group when activated will add the defined EC2 instance to the ELB on the fly:
    - **Enable Cross-Zone Load Balancing**: This option allows the ELB route traffic across Availability Zones.

- **Enable Connection Draining**: This feature only works when it used in conjunction with auto scaling. In auto scaling, the instances are dynamically added or removed based on the defined policies. Auto scaling should not de-register an instance from the ELB when it is in the middle of processing a request. Enabling the connection draining option performs two functions: one, it does not de-register the instance immediately (if it is processing a request), instead it delays the termination by a predefined time period, and two, it does not route any new traffic to the instance. After the elapsed time period the ELB will de-register the instance and hopefully by that time the instance would have processed all the pending requests. The default time period is 300 seconds; you can change it as per your application needs:



8. Next, we add a tag to the ELB; enter the **Key** field as `Name` and the **Value** field as `A1ElectonicsEcommerce-ELB-us-west-2`. Now, click on the **Review and Create** button:



9. The final step is to review all the configuration details and change, if required (by clicking on the **Previous** button). Now, click on the **Create** button:

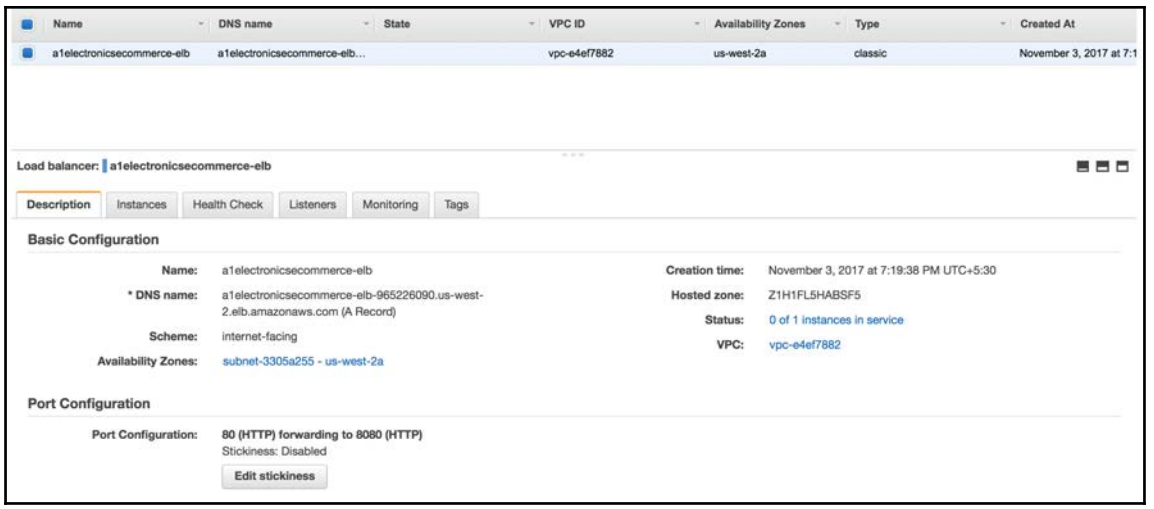


10. You should see the following message. Click on the **Close** button:

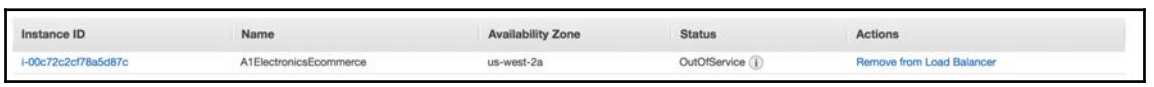


After the ELB has been created it will be assigned a DNS name by which you can access it over the internet. This DNS name is assigned by AWS and cannot be changed, moreover it is not a user-friendly name. You would rather use `www.a1eletronics.com` than `a1electronicsecommerce-elb-965226090.us-west-2.elb.amazonaws.com`. In a production environment, you would need to associate your custom domain name with you ELB domain name by registering a CNAME with your DNS provider.

11. Click on the **Description** tab and check the **Status** row. Initially, you might see the status as **0 of 1 instances in service**:



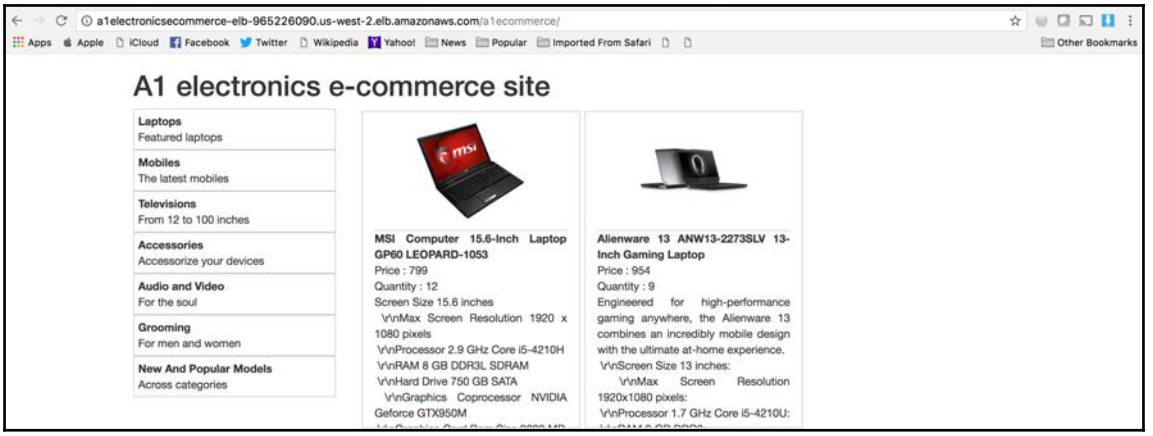
12. Click on the **Instances** tab. Initially, you might see the **Status** as **OutOfService**:



Shortly thereafter, the **Status** will change to **InService** as shown:



13. At this stage, you can key in the DNS name from the **Description** tab into a browser, and you should see the homepage of your site (add /a1ecommerce at the end of the DNS name):



In the next section, we will create a launch configuration.

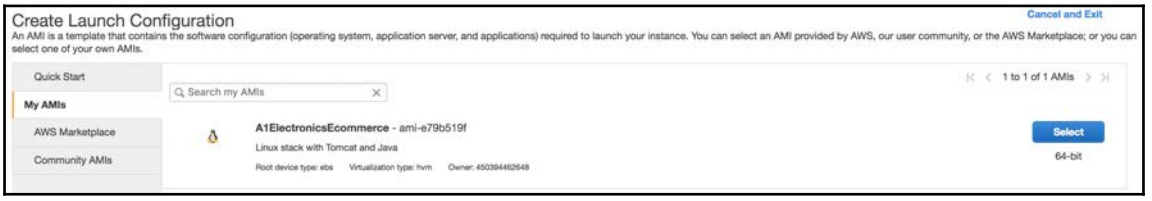
Creating launch configuration

A **launch configuration** is a template which is used by the auto scaling group to select and configure the EC2 instances. This includes configuring the IAM role, configuring the IP address, disk size, security group, and public-private key pair to access the instances. You cannot modify the launch configuration after you have created it:

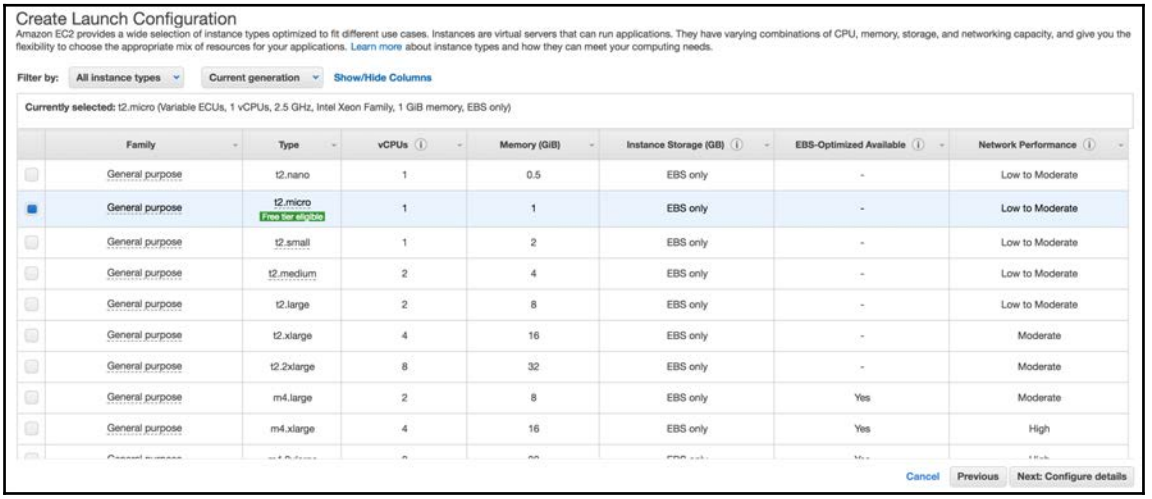
1. From the EC2 dashboard navigation pane, click on **Launch Configurations**, then on **Create Auto Scaling group**, and on the **Create launch configuration** button to start the launch configuration creation process:



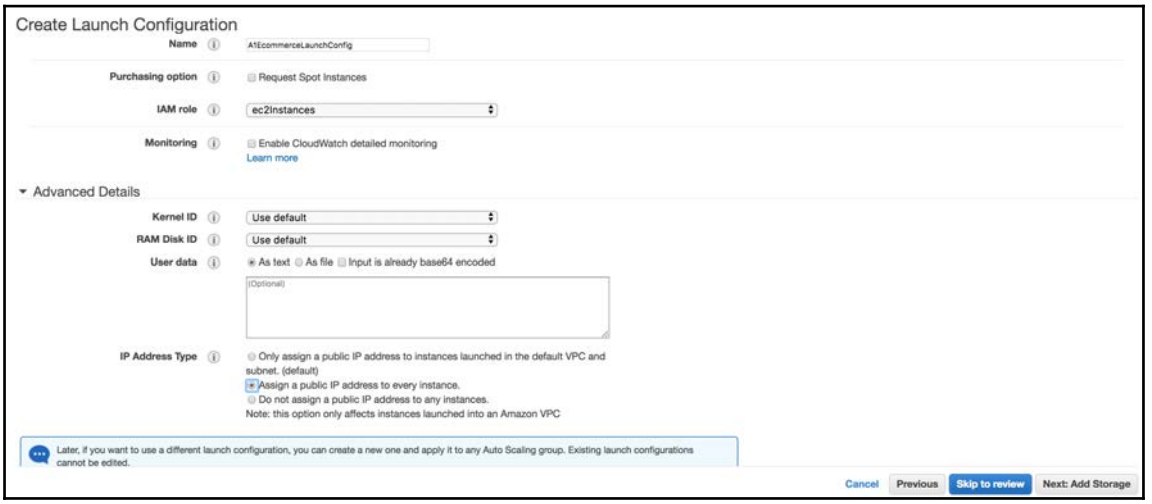
2. The first step is to select the AMI; as we have already created an AMI earlier, select it from **My AMIs** in the navigation pane:



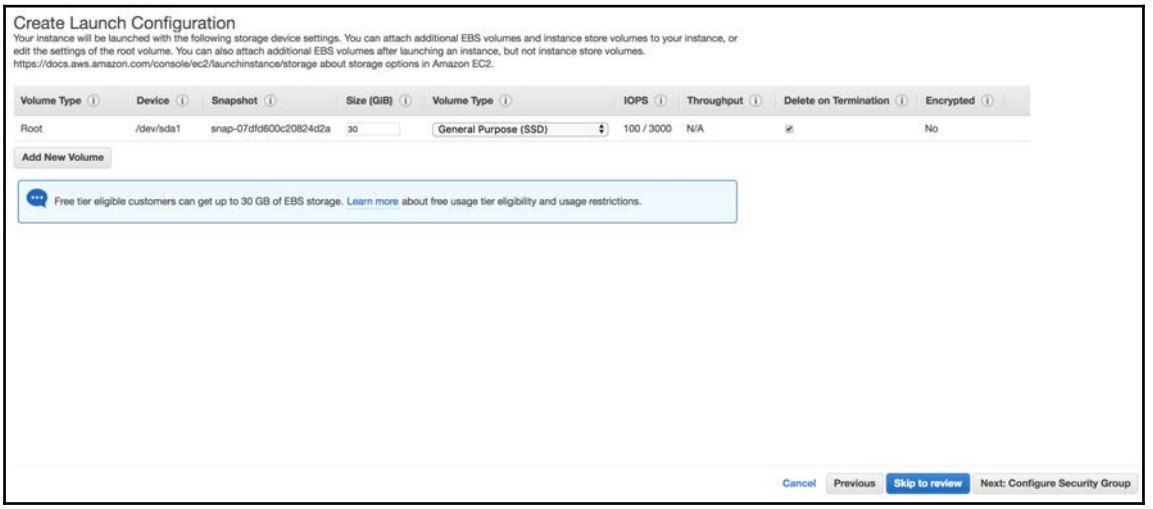
3. The next step select the instance type. We select **t2.micro** instance. Now, click on the **Next: Configure details** button:



4. In this step, we will configure the AMI. Apart from filling in the usual values, such as the name, most of the other parameters are already discussed in `Chapter 3`, *AWS Components, Cost Models, and Application Development Environments*, under the *Creating an EC2 Instance* section. Click on the **Advanced Details** hyperlink to see all the fields. We select the option to **Assign a public IP address to every instance** so that we can SSH into it. As a good security practice, for production servers select the option of **Do not Assign a public IP address** for any of the instances. If you want to access the instances then you can create an EC2 instance which can only be connected to from your static IP address, and from there you can access any instance. This is sometimes called a bastion host, or jump host. Now, click on the **Next: Add Storage** button:



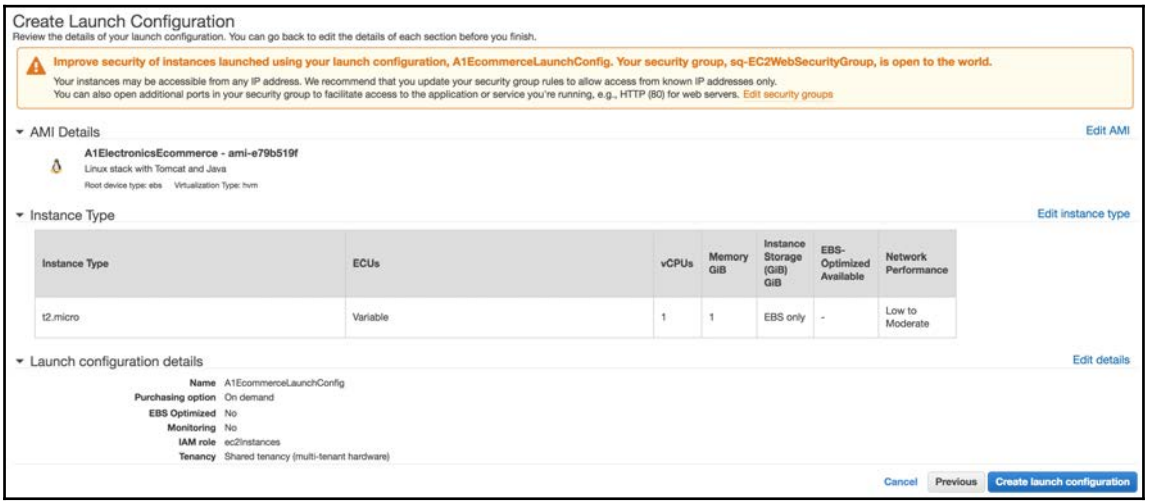
5. Next we add storage to the AMI; use the defaults unless your requirement is for high disk bandwidth. Now, click on the **Next: Configure Security Group** button.



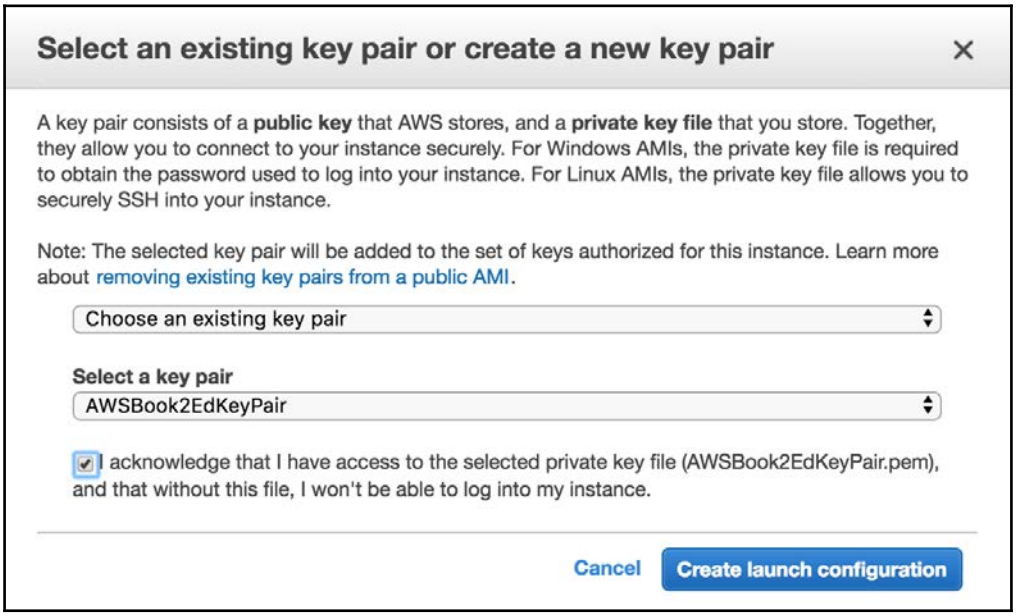
6. Next, we configure the security group; use the one which was created in `Chapter 3`, *AWS Components, Cost Models, and Application Development Environments*, under the *Creating Security Groups* section (and modified earlier in this section):



7. Next we review the launch configuration and make changes if required. Now, click on the **Review** button:



8. Click on the **Create launch configuration** button. Before the launch configuration is created you need to provide the public private key pair to SSH into the instance. Select the public private key created in `Chapter 3`, *AWS Components, Cost Models, and Application Development Environments,* under the *Creating EC2 Instance Key Pairs* section, the ec2AccessKey. Now, click on the **Create launch configuration** button:



You should see the following screen. Click on **Create an Auto Scaling group using this launch configuration** button:

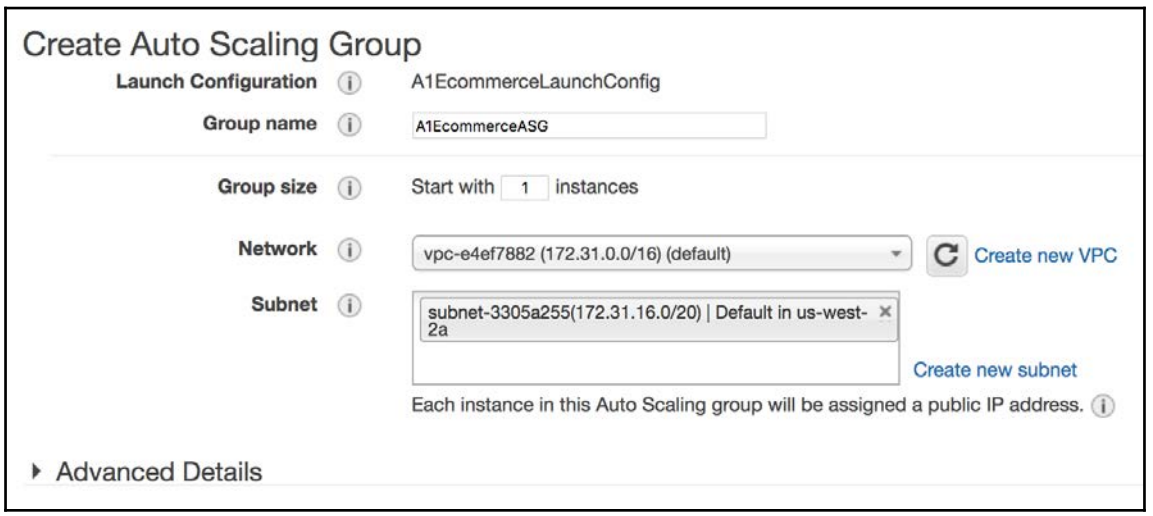


Creating an Auto Scaling group

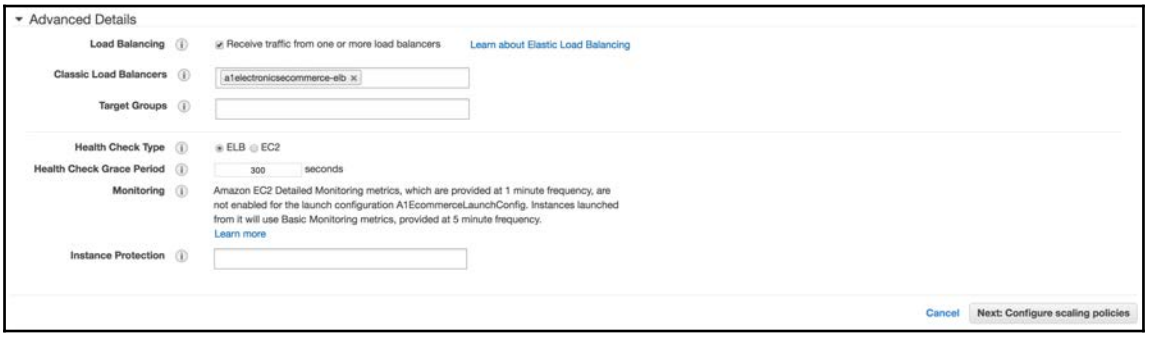
After the launch configuration is created you are directly taken to the creation of Auto Scaling groups:

1. The first step is to specify the auto scaling group details:
    - **Launch Configuration**: The launch configuration for this auto scaling group. This is pre-filled in our case as `A1EcommerceLaunchConfig`.
    - **Group name**: The name of the group for this Auto Scaling group.
    - **Group size**: The size here refers to the minimum number of instances that run inside the auto scaling group. This number typically depends on the load the application is expecting and how many requests a single instance can serve without any latencies. Since ours is a demo with the aim to keep the costs to a bare minimum, we start with one instance.
    - **Network**: Select the VPC where the auto scaling group will launch. Use the default VPC already created in Chapter 3.
    - **Subnet**: Select the default subnet for the Auto Scaling group within the selected VPC.

    Click on the **Advanced Details** link to expand it:



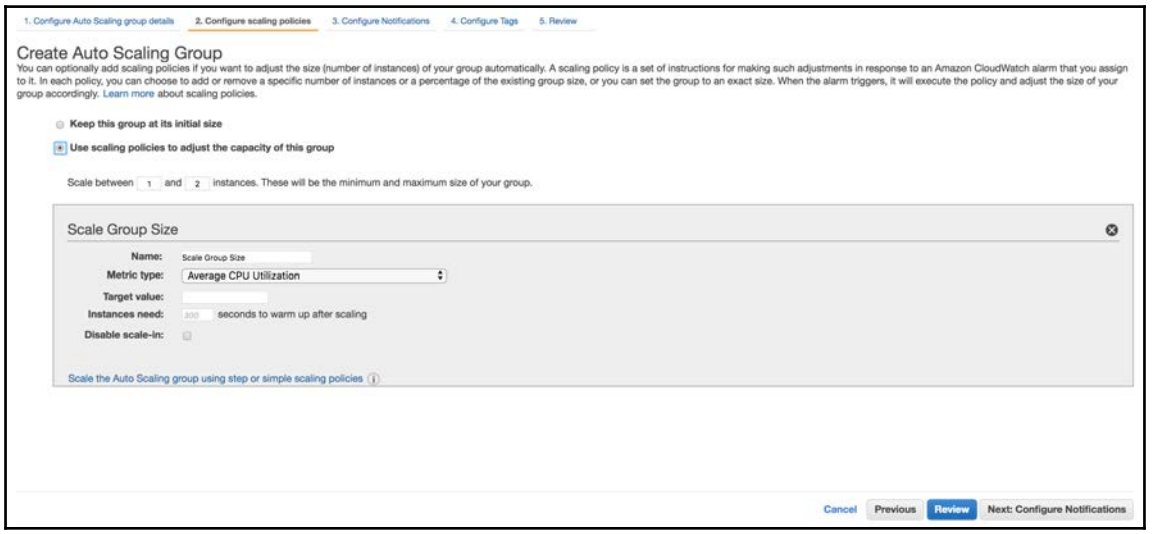
- **Load Balancing**: An Auto Scaling group can be associated with an elastic load balancer. Select the elastic load balancer we created earlier in this chapter in the *Creating Elastic Load Balancer* section, a1electronicsecommerce-elb. If you are using other means of load balancing for your Auto Scaling group then uncheck this option.
- **Health Check Type**: The Auto Scaling group performs health checks on the instances in the group and replaces the failed instances with new ones. It can either use the results of the elastic load balancer or monitor the state of the EC2 instance to detect a failed instance. Select the ELB option since we have configured our auto scaling group to use an ELB.
- **Health Check Grace Period**: When an instance comes up it might take some time before it starts responding to the health checks. The time should always be greater than the expected boot time of the instance plus the startup time of the application. Choose the default value of 300 seconds. Now, click on the **Next: Configure scaling policies** button:



2. Next we configure the scaling policies for the Auto Scaling group. A policy is a rule which defines how to scale in our response to varying conditions. The two options are:
    - **Keep this group at its initial size**: This is a static option. The auto scaling group does not scale in or out the number of instances specified for group size (as defined earlier). The Auto Scaling group will monitor and replace any failed instances.

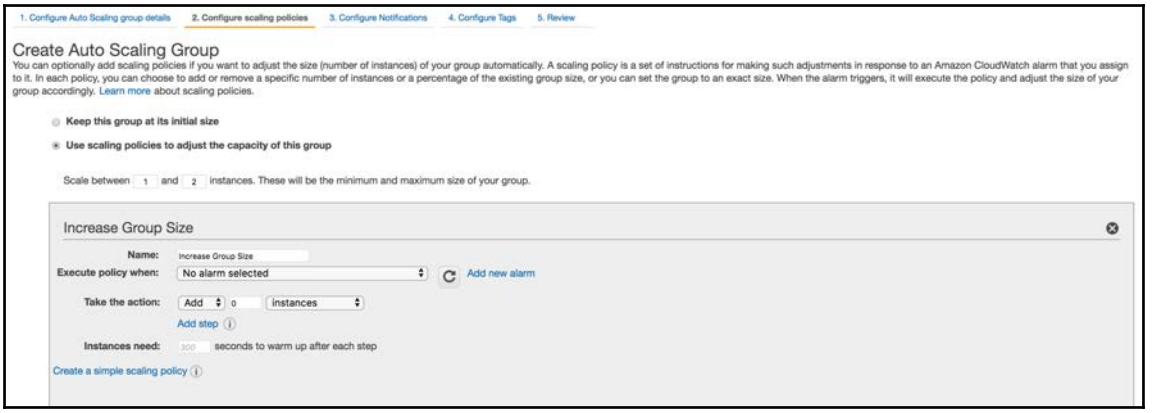
- **Use scaling policies to adjust the capacity of this group**: This option allows the auto scaling group to scale in or out dynamically depending the conditions. These conditions are alarms set in CloudWatch which monitor a pre-selected metric of an AWS resource. Whenever an alarm goes off, breaching the metric limit, CloudWatch sends a message to the scale-in or the scale-out policy which in turn invokes the scaling activity. There are two policies to be defined, one for scaling in and the other for scaling out.

Set the minimum and maximum instance in the Auto Scaling group. This depends on your application and the server's instance type. For our purpose, set this minimum to `1` and maximum `2`. Next, we define the policy which will increase the number of instance in an Auto Scaling group when an alarm associated with it goes off. Now, click on the **Scale the Auto Scaling group using step or simple scaling policies** link:

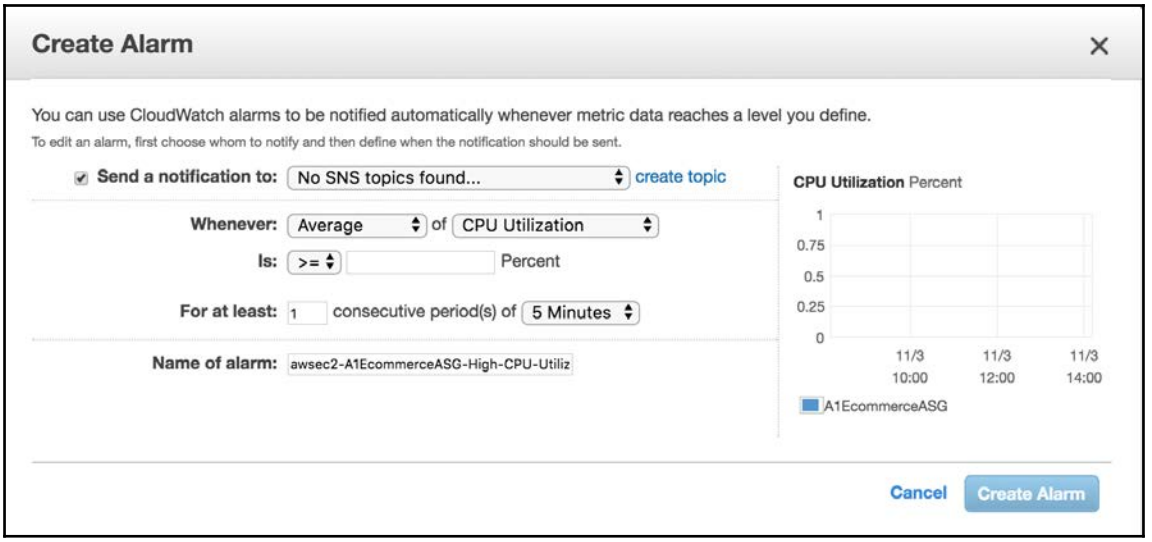


3. You should see the following screen:
    - **Name**: Name of the increase group. Keep the default value.
    - **Execute policy when**: This is where we create an alarm which will be triggered by CloudWatch when it is breached. Since there are no alarms set, click on **Add new alarm**. See step 5 for more details.
    - **Take the action**: When the alarm triggers, we can either add or remove an instance in terms of either percentage of total instances or by a fixed number of instances. Since it is a scale-out situation, we add an instance to the auto scaling group. Adding instances can be done either in terms of percentage of the total machines running in the auto scaling group or by a fixed number of instances. Since the maximum instances in our auto scaling group is two, increasing by a percentage does not make sense.

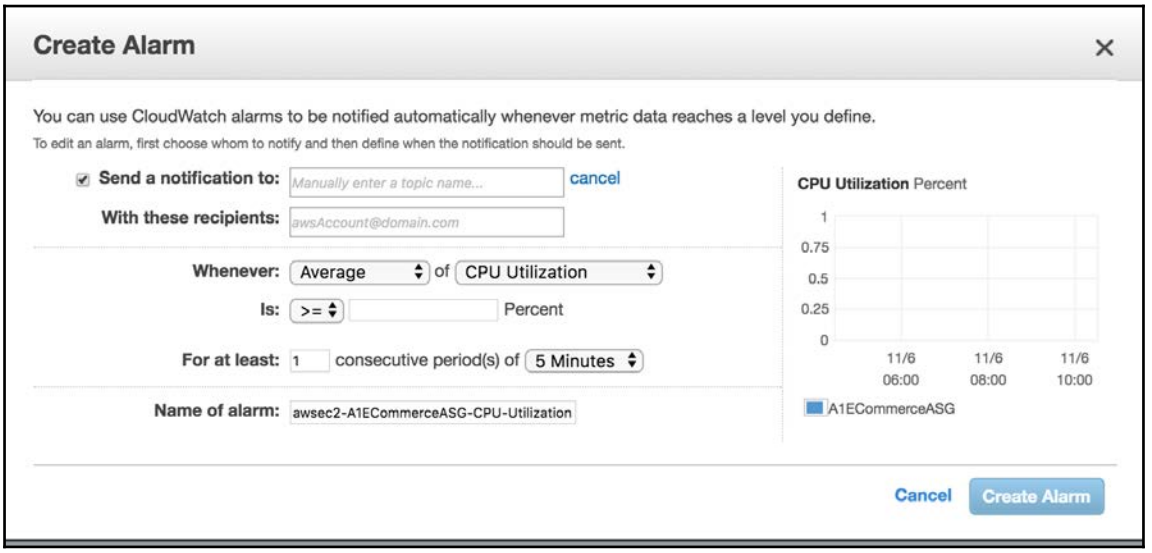
    Now, click on the **Add new alarm** link:



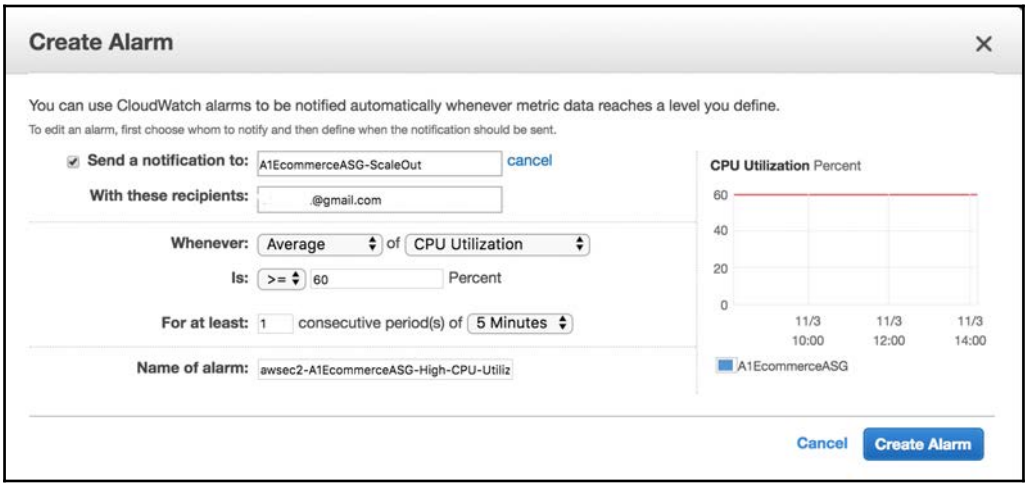
4. You should see the following screen. Now, click on the **create topic** hyperlink:



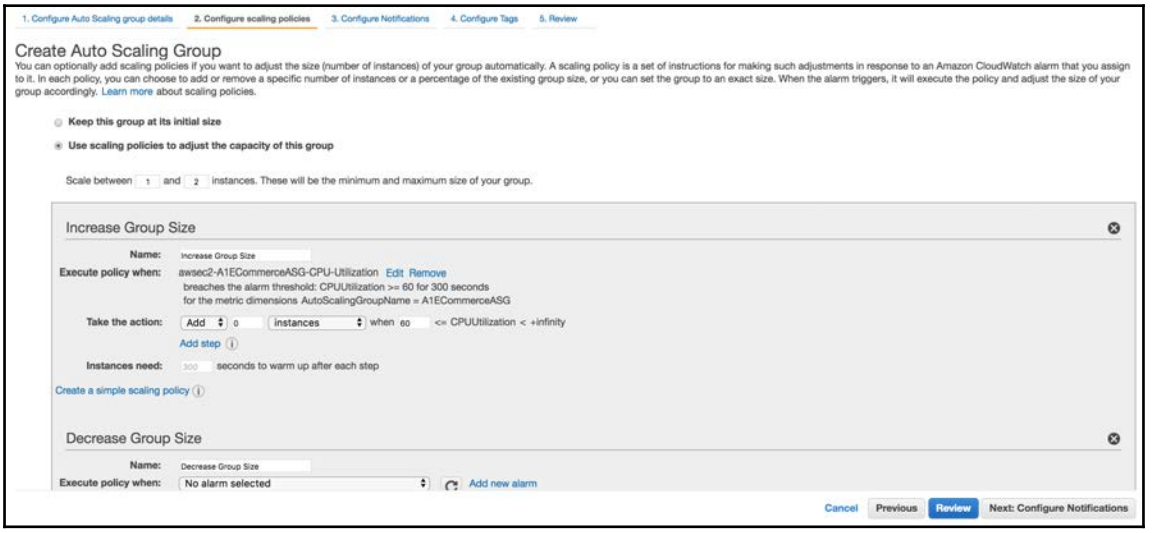
5. Your should see the following screen. Fill in the details for the topic name and enter an email address in the recipients field.



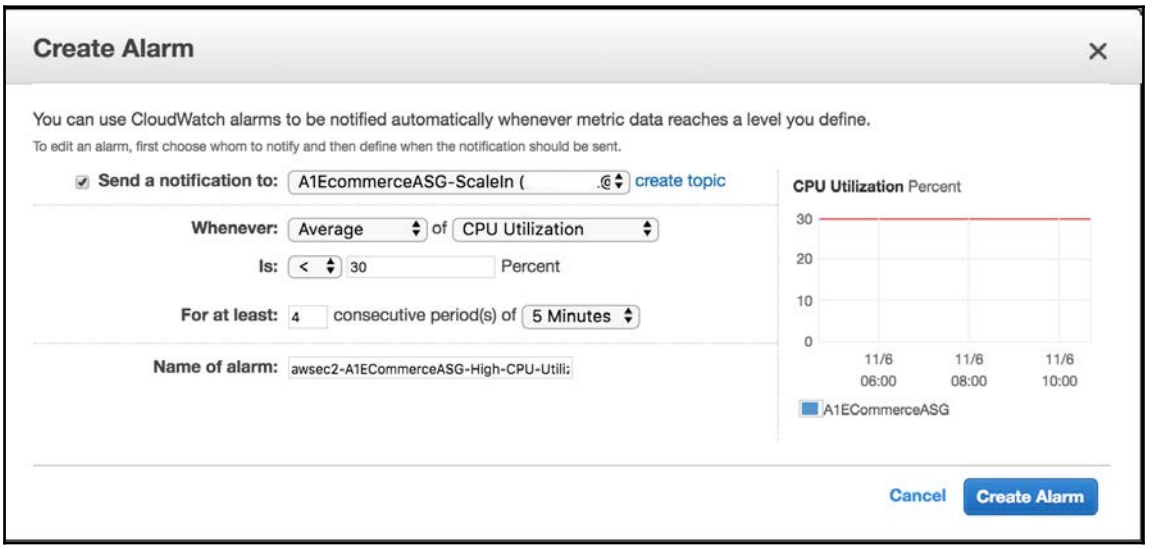
When creating a CloudWatch alarm for scaling out, it is very important to choose the right set of parameter values for the pre-conditions (of the scale-out). The idea is to trigger the alarm 5 to 10 minutes before the 75% of the target threshold is reached. You need to take into account the bootup time of the instance as well as whether the application before the instance is ready to serve the requests; hence we trigger early and catch up with the future demand. The metric we are using to trigger the alarm is CPU utilization. The other useful metrics which can be used are network utilization and memory utilization. The alarm we want to set is **trigger the alarm when the CPU utilization of the EC2 instance is greater than 60% for at least 5 minutes**. The fully filled out screen details are displayed as follows. Now, click on the **Create Alarm** button:



You should see the following screen:

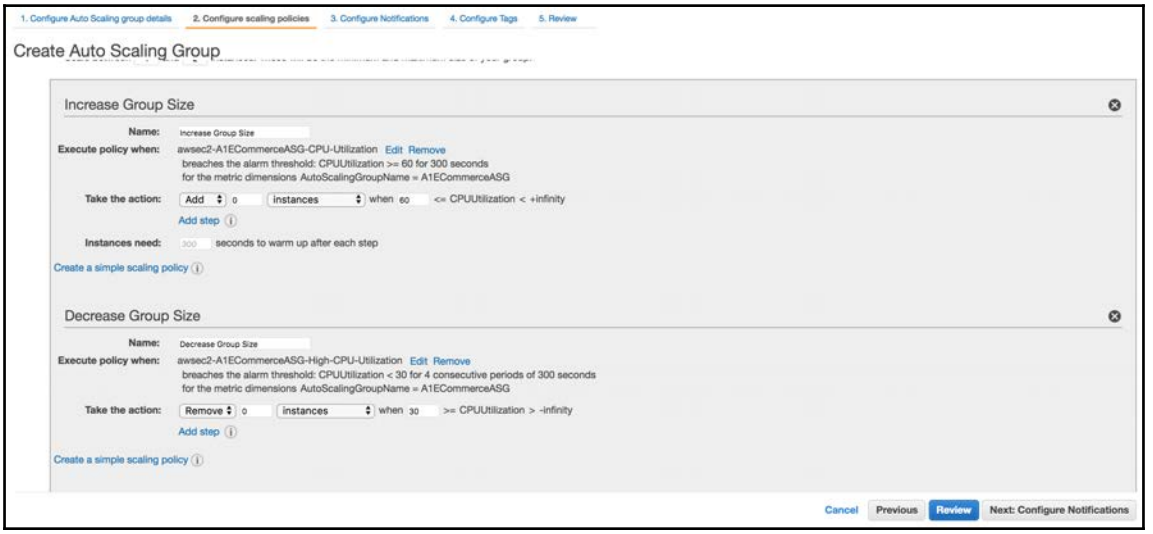


6. Next, we will configure the policy for decreasing the group size. As done before, we create a separate topic for the scale in part. The filled out details are as shown. The alarm we want to set is **trigger the alarm when the CPU utilization of the EC2 instance is less than 30% for at least 20 minutes**. Click on the **Create Alarm** button:

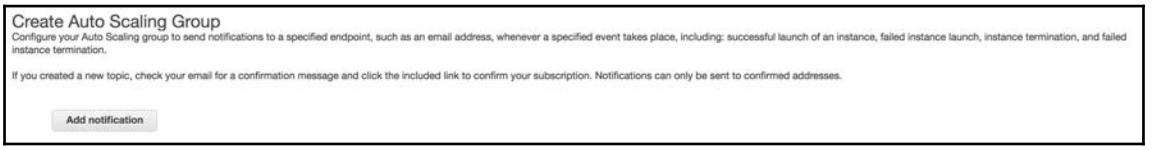


7. You should see the following screen:
    - **Decrease Group Size**: Next, we define the policy which will decrease the instance in an auto scaling group when an alarm associated with it goes off.
    - **Name**: Name of the decrease group. Keep the default value.
    - **Execute policy when**: This is where we create an alarm which will be triggered by CloudWatch when it is breached. Since there are no alarms set, click on **Add new alarm**.
    - **Take the action**: When the alarm triggers we can either add or remove an instance in terms of either the percentage of total instances or by a fixed number of instances. Since it is a scale-in situation, we remove an instance from the Auto Scaling group. Adding instances can be done either in terms of the percentage of total machines running in the Auto Scaling group or by a fixed number of instances. Since the minimum instances in our Auto Scaling group is one, decreasing it by a percentage does not make sense.

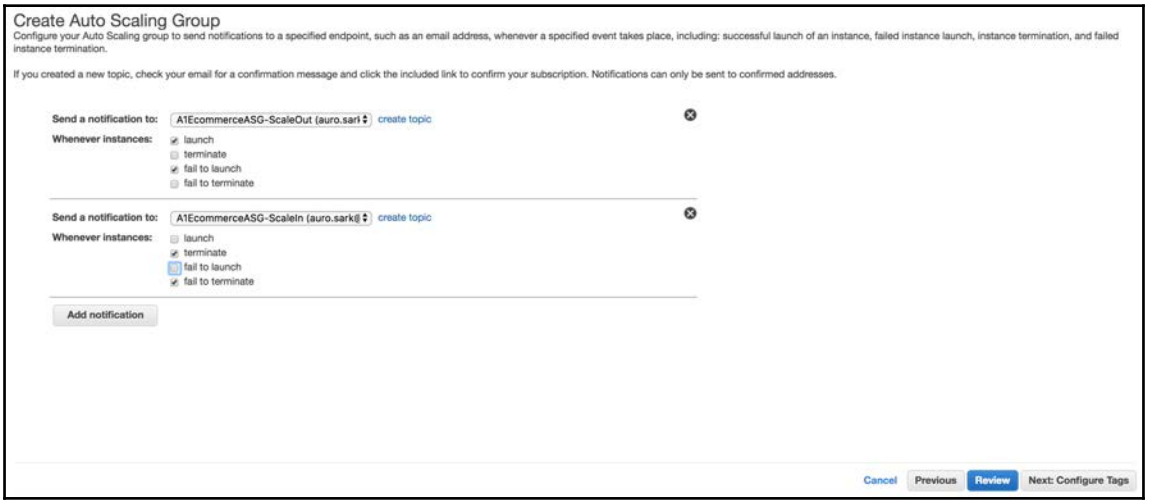
    Change the number of instances to one for the **Take the Action** field for both the groups (**Add** one for **Increase Group Size** and **Remove** one for **Decrease Group Size**). Click on the **Next: Configure Notification** button:



8. Next, we configure the Auto Scaling group to send the notifications to the Amazon SNS topic whenever a scaling event takes place. Currently, we are only interested in fail to launch and fail to terminate scaling events, but in production it is strongly recommended to send all the scaling events. Now, click on the **Add notification** button:



9. You should see the following screen. For the **Scale Out** topic, select the options for **launch** and **fail to launch**, and for the **Scale In** topic select the **terminate** and **fail to terminate** options. Now, click on the **Next: Configure Tags** button:



10. Next, we configure tags; these tags will be applied to all the instances managed by the Auto Scaling group. A maximum of 10 tags can be configured. This is useful when there are many auto scaling groups for different layers. It becomes easier to identify the EC2 instances in the dashboard. Make sure you to select the **Tag New Instances** option. Enter the **Key** field as `Name` and the **Value** field as `A1EcommerceASG`. Now, click on the **Review** button:

>



11. The next step is to review and create the Auto Scaling group. From the navigation pane of the EC2 dashboard, click on **Auto Scaling Groups**; this will list all the auto scaling groups. When the Auto Scaling group is created it also creates the alarms in CloudWatch and topics in SNS. When the Auto Scaling group starts, it starts with the minimum number of instances. Now, click on the **Create Auto Scaling group** button:



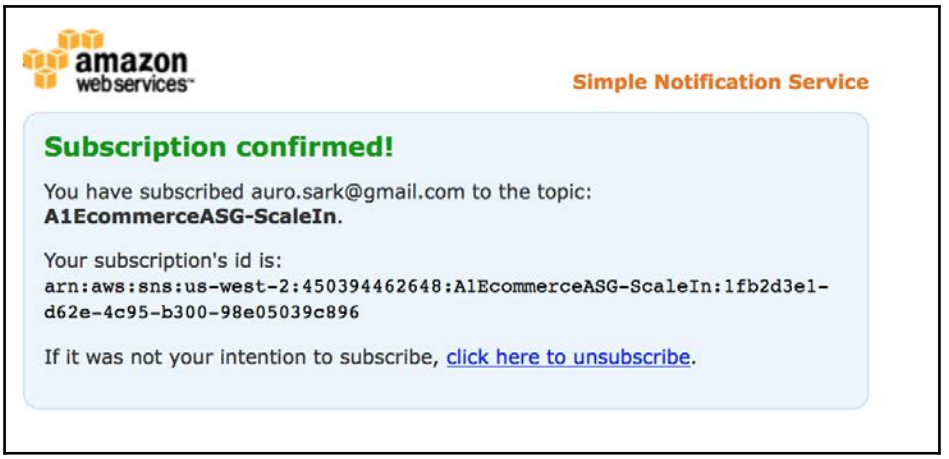
12. You should see the following message. Click on the **Close** button:



13. At this stage, check your emails. You should see two emails: one for the Scale Out and the other for the Scale In topic. In the following screenshot, we show you the Scale In message only. Click on the **Confirm subscription** hyperlink:



14. You should see the following screen confirming the subscription:



You have now completed the Auto Scaling group creation process. In the next section, we will test the Auto Scaling group to ensure it is functioning as expected.

Testing Auto Scaling groups

The next step is to test the Auto Scaling group; it should add an instance when the CPU utilization is greater than 60% for 5 minutes and remove an EC2 instance if the CPU utilization falls to less than 30% for 20 minutes. The easiest way to test this is to load the CPU for more than 5 minutes and check if an EC2 instance is added. As we have configured the launch configuration to assign a public IP address to an instance, the public IP address of the instance is available from Instances in the EC2 dashboard navigation pane.

Log in to the instance via `ssh`:

```
ssh -i ~/.ssh/AWSBook2EdKeyPair.pem ubuntu@34.215.190.229
```

To tax the CPU, use `bc`, which is precision calculator language to compute a value that will take a lot of time. For example, computing 2 to the power of 10 billion here, will push the CPU to 100%:

```
echo 2^1234567890 | bc
```

Now, we have set the stage for the auto scaling group to add a new instance whenever the average load is higher than 60% for more than 5 minutes. There are multiples ways to verify that an instance has been added when the scale out alarm is breached.

As we have set a notification with the scale out alarm, an email will be sent to the configured email address with all the alarm details.

A new EC2 instance is added to the Instances view in the EC2 dashboard.

The Auto Scaling Group view in the EC2 dashboard has a tab for Scaling History which displays all the scaling events.

In the same way, we can verify the scale in by the Auto Scaling group, that is, the removal of an EC2 instance when the average CPU utilization of the EC2 instances falls below 30% for a period of 20 minutes. This can be achieved by ending the `bc` task. The average CPU utilization of the instances will fall below 30% and the scaling in alarm will be breached after a period of 20 minutes.

Summary

In this chapter, we reviewed some of the strategies you can follow for achieving scalability for your cloud applications. We emphasized the importance of both designing your application architecture for scalability and using AWS infrastructural services to get the best results. We followed this up with sections on implementing API-driven applications and streaming applications using AWS services. Finally, we implemented auto scaling for our sample application.

In the next chapter, we will shift our focus to strategies for achieving high availability for your cloud-based applications. We will review some application architectural principles and AWS infrastructural features to implement high availability. We will also include a hands-on section that will walk you through the process of implementing high availability in the sample application.

5

Designing for and Implementing High Availability

In this chapter, we will introduce some key design principles and approaches to achieving high availability in your applications deployed on the AWS cloud. As an enterprise or a start-up, you want to ensure that your mission critical applications are always available to serve your customers. The approaches in this chapter will address availability across the layers of your application architecture including availability aspects of key infrastructural components to ensure that there are no single points of failure.

In order to address availability requirements, we will use AWS infrastructure (Availability Zones and Regions), AWS Foundation Services (EC2 instances, Storage, Security and Access Control, and Networking), and AWS PaaS services (DynamoDB, RDS, CloudFormation, and so on). In addition to availability, we will describe several approaches used for **disaster recovery** (**DR**). We will also show you how to implement high availability for our sample application.

In this chapter, you shall learn the following topics:

- Defining availability objectives
- Nature of failures
- Setting up VPC for high availability
- Using ELB and route 53 for high availability
- Setting up high availability for application and data layers
- Implementing high availability in the application
- Using AWS for disaster recovery
- Testing the disaster recovery strategy

Defining availability objectives

It is easy to get confused between HA and cost optimization objectives because product owners will often push for cost optimization while ignoring their availability requirements until something fails. As a standard practice, remember to always prioritize availability goals and only then look at ways to optimize your costs.

Achieving high availability can be costly. Therefore, it is important to ensure that you align your application's availability requirements with your business objectives. There are several options to achieve the level of availability that is right for your application. Hence, it is essential to start with a clearly defined set of availability objectives and then make the most prudent design choices to achieve those objectives at a reasonable cost. Typically, all systems' functionality and services do not need to achieve the highest levels of availability possible, but at the same time, ensure that you do not introduce a single point of failure in your architecture through dependencies between your components. For example, a mobile taxi ordering service needs its ordering-related service to be highly available; however, a specific customer's travel history need not be addressed at the same level of availability.

The best way to approach high availability design is to assume that anything can fail, at any time, and then consciously design against it.

> *"Everything fails, all the time." - Werner Vogels, CTO, Amazon.com*

In other words, think in terms of availability for each and every component in your application and its environment because any given component, at any given time, can turn into a single point of failure for your entire application. Availability is something you should consider early in your application design process, as it can be hard to retrofit the same later. In addition, it is important to understand that availability objectives can influence and or impact your design, development, test, and running of your system on the cloud.

Finally, ensure that you proactively test all your design assumptions and reduce uncertainty by injecting or forcing failures instead of waiting for random failures to occur.

Nature of failures

There are many types of failures that can happen at any time. These could be a result of disk failures, power outages, natural disasters, software errors, and human errors. In addition, there are several points of failure in any given cloud application. These could include DNS or domain services, load balancers, web and application servers, database servers, application services-related failures, data center-related failures, and so on. You will need to ensure that you have a mitigation strategy for each of these types or points of failure.

It is highly recommended that you automate your recovery strategy and thoroughly test as many of these processes as possible.

Currently, the AWS Cloud operates 44 Availability Zones (AZs) within 16 geographic Regions around the world, with announced plans for 17 more Availability Zones and six more regions. The high-speed connections between AZs allow to architect highly available applications across different physical locations. In the next few sections, we will discuss various strategies to achieve high availability for your application. Specifically, we will discuss the use of AWS features and services as follows:

- VPC
- Amazon Route 53
- Elastic Load Balancing (ELB)
- Auto Scaling
- Redundancy
- Multi-AZ and multi-region deployments.

Setting up VPC for high availability

In this section, we describe a common VPC setup for some of the high-availability approaches discussed later in this chapter.

Before setting up your VPC, you will need to carefully select your primary site and a DR site. Leverage AWS's global presence to select the best regions and availability zones to match your overall business objectives. The choice of a primary site is usually the closest region to the location of a majority of your customers, and the DR site could be in the next closest region, or a significantly distant one depending on your objectives. Next, we need to set up the network topology, which essentially includes setting up the VPC and the appropriate subnets. The public-facing servers are configured in a public subnet, whereas the database servers and other application servers hosting services like the directory services will normally reside in the private subnets.

Ensure that you choose different sets of IP addresses across the different regions for the multiregion deployment, for example, `10.0.0.0/16` for the primary region and `192.168.0.0/16` for the secondary region to avoid any IP addressing conflicts when these regions are connected via a VPN tunnel. Appropriate routing tables and ACLs will also need to be defined to ensure that traffic can traverse between them. Cross-VPC connectivity is required so that data transfer can happen between the VPCs (say, from the private subnets in one region to the other region). The secure VPN tunnels are basically IPSec tunnels powered by VPN appliances—a primary and a secondary tunnel should be defined (in case, the primary IPSec tunnel fails).

ELB is configured in the primary region to route traffic across multiple availability zones. However, you need not necessarily commission ELB for your secondary site at this time. Even though the ELB is not particularly expensive, this will help you avoid unnecessary costs for the ELB in your DR or secondary site. Gateway servers and NAT will need to be configured as they act as gatekeepers for all inbound and outbound internet access. Gateway servers are defined in the public subnet with appropriate licenses and keys to access your servers in the private subnet for server administration purposes. NAT is required for servers located in the private subnet to access the internet and is typically used for automatic patch updates.

Elastic load balancing and Amazon Route 53 are critical infrastructure components for scalable and highly available applications; we discuss these services in the next section.

Using ELB and Route 53 for high availability

In this section, we describe different levels of availability, and the role ELBs and Route 53 play from an availability perspective.

Instance availability

The guideline here is to never run a single instance in a production environment. The simplest approach to improving availability is to spin up multiple EC2 instances and stick an ELB in front of them. The incoming request load is shared by all the instances behind the load balancer.

ELBs use a least connections algorithm to spread requests across healthy instances. Least connections target instances with the fewest outstanding requests and adjust to the request response times of an instance. For example, slower response times from an instance will result in that machine receiving fewer requests.

Even though it is not recommended to have different instance sizes, within a specific tier, between or within the AZs, the ELB will adjust for the number of requests it sends to smaller or larger instances based on response times. In addition, ELBs use cross-zone load balancing to distribute traffic across all healthy instances regardless of AZs. Hence, ELBs help balance the request load even if there are unequal number of instances in different AZs at any given time (perhaps due to a failed instance in one of the AZs). Note that there is no bandwidth charge for cross-zone traffic (if you are using ELB).

Auto Scaling for increased availability and reliability

If the AZ goes down, then new instances are spun up in a different AZ, if necessary. As soon as the first AZ comes back to life, auto scaling will try to launch the instances there and try rebalancing the load appropriately. This is the only time the number of instances can go above max capacity specified (for a short amount of time). As soon as a certain amount of capacity is available in AZ 1, it will start terminating the instances in AZ 2.

Instances that fail can be seamlessly replaced using auto-scaling while other instances continue to operate. Although auto-replacement of instances works really well, storing application state or caching locally on your instances can lead to problems hard to detect.

Health checks are performed periodically and the instances are marked as unhealthy or healthy. Unhealthy instances are terminated and replaced (if new number of instances < minimum or < desired capacity). The EC2 instance status is unhealthy when the instance state is not "running" or the system health check is "impaired". Instances are determined to be unhealthy when ELB health check results in "OutOfService" (or the EC2 health check fails). You can also define your own health checks to mark individual instances as "unhealthy". You can also integrate instance statuses with an external monitoring systems.

TCP- and / or HTTP-based heartbeats can be created for this purpose. However, it is important to consider the depth and accuracy of your health checks because deep-in-the-stack health checks can lead to situations where you can get excessive number of false positives and false negatives in your results. It is worthwhile to approach the implementation of health checks, iteratively, to arrive at the right set that meet your goals.

Zonal Availability or Availability Zone Redundancy

Availability Zones are distinct geographical locations engineered to be insulated from failures in other zones. It is critically important from a HA perspective to run your application stack in more than one zone. In addition, avoid dependencies across the zones as far as possible. For sites with very high request loads, a 3-zone configuration may be the preferred configuration to handle zone-level failures. In this situation, if one zone goes down, then other two AZs can ensure continuing high availability and uninterrupted customer experience.

In the event of a zone failure in a multi-AZ configuration, there are several challenges. The DNS record will contain multiple IP addresses and DNS round robin can be used to balance traffic between the availability zones. Using multiple AZs can result in traffic imbalances between AZs due to clients caching DNS records. However, ELBs can help reduce the impact of this caching.

Region availability or regional redundancy

Elastic Load Balancing and Amazon Route 53 have been integrated to support a single application across multiple regions. Route 53 is AWS's highly available scalable DNS and health checking service. Route 53 supports high availability architectures by health checking load balancer nodes and re-routing traffic to avoid the failed nodes as they support implementation of multi-region deployments. In addition, Route 53 uses Latency Based Routing to route your customers to the end-point that has the least latency. If multiple primary sites are implemented with appropriate health checks configured then failures result in traffic being shifted away from that site to an alternate region.

Region failures can present several challenges as a result of rapidly shifting traffic (similar to the case of zone failures). These can include Auto Scaling, time required for instance startup, and cache fill time (as we may need to fault to our data sources, initially). Another difficulty usually arises from the lack of information or clarity on what constitutes the minimal or critical stack required to keep the site functioning as normally as possible. Any or all services will need to be considered as critical in these circumstances.

The health checks are essentially automated requests sent over the internet to your application to verify that your application is reachable, available, and functional. This can include both your EC2 instances and your application. As answers are returned only for the resources that are healthy and reachable from the outside world the end users can be routed away from a failed application. Amazon Route 53 health checks are conducted from within each AWS region to check whether your application is reachable from that location.

DNS failover is designed to be entirely automatic. After you have set up your DNS records and health checks, no further manual intervention is required for failover. Typically, it takes about 2-3 minutes from the time of the failure to the point where traffic is routed to an alternate location. Compare this to the traditional long drawn out process where an operator receives an alarm, manually configures the DNS update, and waits for the DNS changes to propagate.

Depending on the availability objectives there is an additional strategy (using Route 53) that you may want to consider for your application. For example, you can create a backup static site to maintain a "presence" for your end customers while your primary (dynamic) site is down. In the normal course, Route 53 will point to your dynamic site and maintain health checks for it. You will also need to configure Route 53 to point to S3 storage where your static site is residing. If your primary site goes down then traffic can be diverted to the static site (while you work to restore your primary site). You can also combine this static backup site strategy with a multiple region deployment.

Setting up high availability for application and data layers

In this section, we will discuss approaches for implementing high availability in the application and data tiers in the application architecture.

In the application layer, we can do a cold start from pre-configured images or a warm start from scaled down instances for the web servers and application servers in a secondary region. By leveraging auto-scaling we can quickly ramp up these servers to handle full production loads. In this configuration you would deploy the web servers and application servers across multiple AZs (in the primary region) and the standby servers in a separate, secondary region. The standby servers need not be launched till you actually need them. However, you should keep the preconfigured AMIs for these servers ready-to-launch in your secondary region.

The data layer can comprise of SQL databases, NoSQL databases, caches, and so on. These can be AWS managed services such as RDS, DynamoDB, and S3, or your own SQL and NoSQL databases like Oracle, SQL Server, or MongoDB running on EC2 instances. AWS services come with HA built-in while using self-managed database products running on EC2 instances offers a do-it-yourself option. It can be advantageous to use AWS services, if you want to avoid taking on database administration responsibilities. For example, with the increasing sizes of your databases you might choose to shard your databases may be easy for you to do. However, re-sharding your databases while accepting live traffic can be a complex undertaking and present availability risks. Choosing to use AWS DynamoDB service in such situations, offloads the work to AWS thereby resulting in higher availability out-of-the-box.

Next, we discuss some data replication options.

DynamoDB automatically replicates your data across several AZs to provide higher levels of data durability and availability. In addition, you can use data pipelines to copy your data from one region to another. DynamoDB streams functionality that can be leveraged to replicate to another DynamoDB in a different region. For very high volumes, low-latency Kinesis services can also be used for this replication across multiple regions distributed all over the world.

You can also enable the multi-AZ setting for the AWS RDS service to ensure that AWS replicates your data to a different AZ within the same region. In the case of Amazon S3, the S3 bucket contents can be copied to a different bucket and the failover managed on the client side. Depending on the volume of data, always think in terms of multiple machines, multiple threads, and multiple parts to significantly reduce the time it takes to upload data to S3 buckets.

While using your own database (running on EC2 instances), use your database-specific high-availability features for within and cross-region database deployments. For example, if you can leverage your databases' replication features for synchronous and asynchronous replication across the nodes. If the volume of data is high, then you can also upload your data to Amazon S3 and restore the data to your database instance on AWS. You can also replicate your non-RDS databases (on-premise or on AWS) to AWS RDS databases. You will typically define two nodes in the primary region with synchronous replication and a third node in the secondary region with asynchronous replication. NoSQL databases, such as MongoDB and Cassandra, have their own asynchronous replication features that can be leveraged for replication to a different region.

In addition, you can create Read Replicas for your databases in other AZs and regions. In this case, if your master database fails followed by a failure of your secondary database, then one of the read replicas can be promoted to being the master. In hybrid architectures where you need to replicate between on-premise and AWS data sources, you can do so through a VPN connection between your data center and AWS. In case of any connectivity issues, you can also temporarily store pending data updates in SQS and process them when the connectivity is restored.

Usually, data is actively replicated to the secondary region while all other servers like the web servers and application servers are maintained in a cold state to control costs. However, in cases of high availability for web-scale or mission critical applications, you can also choose to deploy your servers in active-active configuration across multiple regions.

Implementing high availability in the application

In this section, we will discuss a few design principles to use in your application from a high availability perspective. We briefly discuss using highly available AWS services to implement common features in mobile and **Internet of Things** (**IoT**) applications. Finally, we also cover running packaged applications on AWS cloud.

Designing your application services to be stateless and implementing a microservices-oriented architecture can help the overall availability of your application. In such architectures, if a service fails, then that failure is contained or isolated to that particular service while the rest of your application services continue to serve your customers. This approach can lead to an acceptable degraded experience rather than outright failures or worse. You should also store user or session information in a central location such as the AWS ElastiCache and spread the information across multiple AZs for high availability. Another design principle is to rigorously implement exception handling in your application code, and each of your services, to ensure graceful exit in case of failures.

Most mobile applications share common features including user authentication and authorization, data synchronization across devices, user behavior analytics, retention tracking, storing, sharing, and delivering media globally, sending push notifications, storing shared data, streaming real-time data, and so on. There are a host of highly available AWS services that can be used to implement such mobile application functionality. For example, you can use Amazon Cognito to authenticate users, Amazon Mobile Analytics to analyze user behavior and track retention, Amazon SNS for push notifications, and Amazon Kinesis for streaming real-time data. In addition, other AWS services such as S3, DynamoDB, and IAM can also be effectively used to complete most mobile application scenarios.

Similar to mobile applications, for IoT applications, you can use the same highly available AWS services to implement common functionality, such as device analytics and device messaging / notifications. You can also leverage Amazon Kinesis to ingest data from hundreds of thousands sensors that are continuously generating massive quantities of data.

Aside from your own custom applications, you can also run packaged applications such as SAP on AWS. In such cases, you can leverage some of the same AWS features and approaches discussed in this chapter for high availability. These would typically include replicated standby systems, multi-AZ and multi-region deployments, hybrid architectures spanning your own data center and AWS cloud (connected via VPN or AWS Direct Connect service), and so on. For more details, refer to the specific package guides to achieve high availability on AWS cloud.

Using AWS for disaster recovery

In this section, we discuss how AWS can be leveraged for your on-premise and cloud-based application's disaster recovery. We present several different DR strategies that may be suitable for different types of applications, budgets, and situations.

Disaster recovery scenarios typically include hardware or software failures, network outages, power outages, natural disasters like floods, or other such significant events that directly impact a company's ability to continue with their business. Traditionally, there have been two key metrics driving the implementation of disaster recovery strategies—Recovery Time Objective (RTO) and Recovery Point Objective.



Recovery Time Objective (RTO) is the time it takes to restore the business process (after a disaster) to its service level.

Recovery Point Objective (RPO) is the acceptable amount of data loss measured in time units.

Depending on your RTO and RPO objectives, there are several architectural strategies available to recover from disasters. The main ones in the order of reducing RTO/RPO (but with higher associated costs) are described in the following sections.

Using a backup and restore DR strategy

Backup and restore is the simplest and the most common option used in traditional data centers. The process of taking tape-based backups and doing restores from tapes is familiar to most organizations. However, there are some simpler and often faster options available as AWS services. Amazon S3 is an ideal storage medium for quick backups and restores for your cloud and on-premise applications. Another storage option is to use Amazon Glacier for your longer term backups.

There are several options available for hybrid architectures where your data center extends into the cloud. You can use Direct Connect facility to set up a high throughput and low-latency connection between your data center and AWS cloud for your backups. In case your data volumes are on a terabyte scale, then Amazon also provides a facility where you can ship your data on portable storage media and Amazon will use their high-speed internal network to load it on S3 for you. This is often a more economical option to load your data than upgrading your network connections and transferring massive volumes of data over the internet. Another AWS option is to use the AWS Storage Gateway, an on-premise software appliance, to store your data on AWS S3 or AWS Glacier storage. In cases of disaster, you can choose to launch your workloads within the AWS environment or your own data center environment.

Using a Pilot Light architecture for DR

As the name suggests, in this architecture, you set up data mirroring/replication of your on-premise or cloud databases. In addition, you create images of your critical on-premise or cloud servers. These images would typically include your web servers and application servers. In order to avoid incurring unnecessary running costs, these instances are launched only when they are needed. In the event of a disaster, the web servers and application servers can be quickly launched from the preconfigured images.

Using a warm standby architecture for DR

This option is similar to the Pilot Light architecture; however in this case, we run a scaled-down version of the production environment. In the event of a disaster, we simply divert the traffic to this site and rapidly scale up to the full-blown production environment.

Using a Multi-Site architecture for DR

In a multi-site architecture, there are multiple production environments running in active-active configuration on AWS only or a hybrid of on-premise and AWS cloud infrastructure. In cases of disaster, the traffic is routed to the already running alternate sites. You can use the weighted routing feature of Route 53 for this purpose. You will need to ensure sufficient capacity at each site, or a rapid scale up of capacity as provided by the auto-scaling feature on AWS, to avoid poor customer experience or cascading failures from occurring. The costs associated with this option depend on how much production traffic is handled by AWS during normal operations and at peak loads.

In order to meet the RTO objectives, the infrastructure is fully automated. AWS CloudFormation can be used for this purpose. However, it is recommended to have an engineer closely monitor the recovery process in case rollbacks are required as result of any failures.

Testing disaster recovery strategy

It is imperative to thoroughly test your recovery strategy end to end to ensure that it works and to iron out any kinks in the processes. However, testing for failures is hard, especially for sites or applications with very high request loads or complex functionality. Such web-scale applications usually comprise massive and rapidly changing datasets, complex interactions and request patterns, asynchronous requests, and a high degree of concurrency. Simulating complete and partial failures in such scenarios is a big challenge. At the same time, it is even more critical to regularly inject failures into such applications, under well-controlled circumstances, to ensure high availability for your end customers.

It is also important to be able to simulate increased latency or failed service calls. One of the biggest challenges in terms of testing services-related failures is that many times the services owners do not know all the dependencies or the overall impact of a particular service failure. In addition, such dependencies are in a constant flux. In these circumstances, it is extremely challenging to test service failures in a live environment at scale. However, a well thought out approach that identifies the critical services in your application takes into consideration prior service outages and having a good understanding of dependency interactions or implementing dynamic tracing of dependencies, can help you execute service failure test cases.

Simulating availability zone and/or region failures need to be executed with care, as you cannot shutdown an entire availability zone or region. However, you can shut down the components in an AZ via the console or use CloudFormation to shut down your resources at a region level. After the shutdown of the resources in the AZs of your primary region, you can launch your instances in the secondary region (from the AMIs) to test the DR site's ability to take over. Another way to simulate region-level failures is to change the load balancer security group settings to block traffic. In this case, the Route 53 health checks will start failing and the failover strategy can be exercised.



For a deeper understanding of testing your deployment against failures, refer to Netflix's site for Chaos Monkey (responsible for randomly terminating instances in production to test resiliency to instance failures).

Setting up high availability

This section introduces configuring AWS infrastructure to support high availability (HA) to our application. Amazon's high-level services are designed for high availability and fault tolerance like the Elastic Load Balancer (ELB),Simple Storage Service (S3), Simple Queue Service (SQS), Simple Notification Service(SNS), Relational Database Service(RDS), Route 53 a dynamic DNS service, and Cloudwatch. The infrastructure services, such as Elastic Cloud Compute (EC2) and Elastic Block Storage (EBS), can leverage Availability Zones, Elastic IP addresses and snapshots to design highly available and fault tolerant applications and deployment environments. Remember that hosting an application on the cloud does not make it fault tolerant or highly available. We need to architect for high availability.

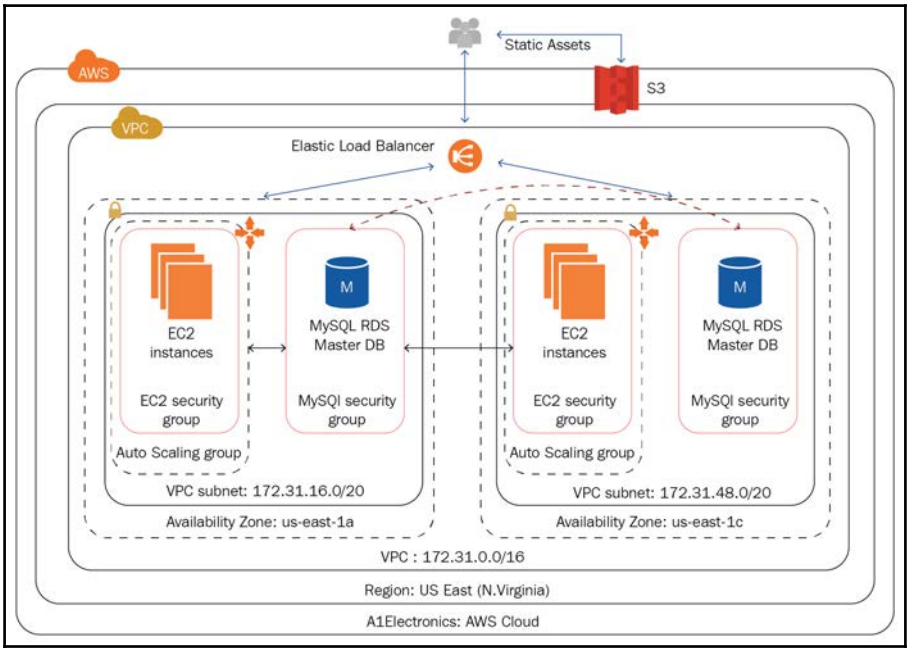
AWS high availability architecture

Let us start by designing a generic high-availability architecture for an Amazon region. High availability can be architected in many different ways depending on the services used and the requirements. Here, we present a very generic architecture. The key to achieving high availability is to avoid single point of failures (SPOF) or application failures resulting from any one of the components or services fails. This implies that we have to account for all the Amazon services that can fail in a region; in our case, these are as follows:

- Availability zone (AZ)
- EC2 instances (EC2)

- Elastic load balancer (ELB)
- Relational Database Service (RDS)

Out of these, ELB and RDS are already designed for high availability and just need to be configured to support high availability as per the architecture. The application is hosted in the US West Region, and the architecture is designed to handle failures within a Region and not across regions.



- **Availability Zones**: An Availability Zone is equivalent to a data center. They are in distinct physical locations and are engineered to isolate failure from each other. The application is hosted in more than one availability zone to isolate them from these failures. The decision on how many Availability Zones to use for an application depends on how critical the application is and the costs involved. Using multiple AZs removes the SPOF for applications using a single AZ.

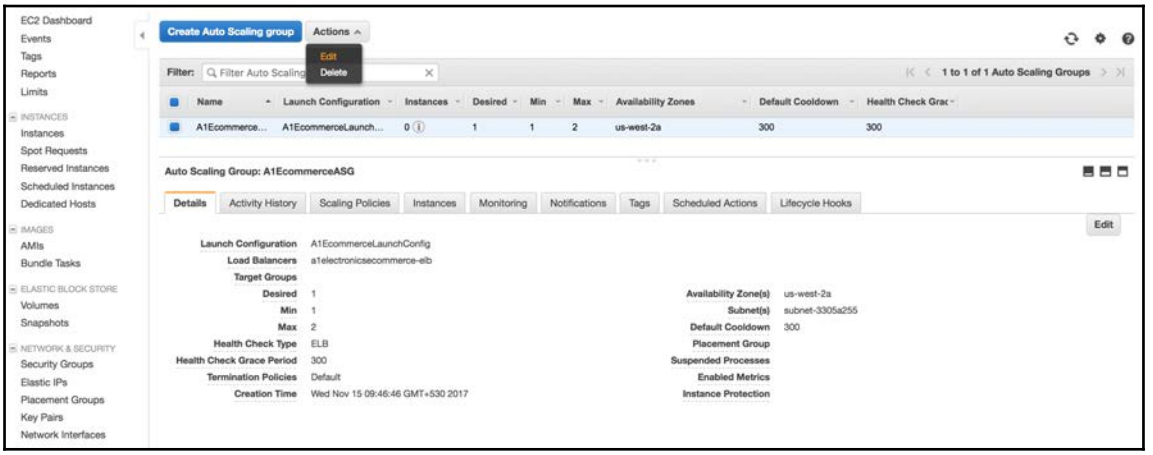
- **Elastic Load Balancer**: All the traffic is routed via the elastic load balancer to the applications. This piece of infrastructure is fault tolerant by design. ELB needs to be configured for routing traffic to the application hosted in different AZs. In addition, the ELB also performs health check on all the EC2 instances registered with it and only routes traffic to healthy EC2 instances. AWS replaces an unhealthy ELB; hence, the SPOF at the load balancer tier is no longer a problem.
- **EC2 Instances**: An EC2 instance is the most vulnerable component in the chain. It is the most complex in terms of the deployment of your software application and the supporting software stack. Any failure in the software stack or the application can potentially render the application to be unavailable. To cover this, an Auto Scaling group (ASG) is used that monitors and launches EC2 instances based on the configuration of alarms and the configured number of maximum, minimum, and desired EC2 instances per availability zone. This addresses the SPOF at web server/application tier.
- **Relation Database Service**: The last in the chain is the database. The RDS service provides high availability and failover using Multi-AZ deployments. The RDS creates a database instance in two AZs with one of them designated as the master and the other a standby replica or slave instance. The master and standby databases are synchronized via synchronous data replication. All the EC2 instances write to the database via a FQDN or an endpoint. Behind the scenes, the RDS service routes the writes to the current master database instance. With Multi-AZ, you can't access the inactive secondary database until the primary fails. If the primary RDS instance fails, the DNS CNAME record is updated to point to the new master. If the standby fails, a new instance is launched and instantiated from the primary as the new standby. Once failover is detected, it takes less than a minute to switch to the slave RDS instance. Multi-AZ deployment is designed for 99.95% availability. In addition, RDS can be configured to snapshot the database at regular intervals that helps during disaster recovery. This addresses the SPOF in the database tier.
- **Simple Storage Service** (**S3**): S3 is a highly available service in order to store static assets. Amazon S3 is designed for 99.99% availability. All the files uploaded to the application should be stored in S3 so that even if the EC2 instances fail, the uploaded file is not lost, and another EC2 instance can process the workload, if required. It is a good practice to store all the static assets such as images and scripts of an application to S3, as it takes the load off your EC2 instances.

- **Virtual Private Cloud (VPC)**: Amazon automatically creates a VPC in a region for all the accounts created after 2013-03-18. By default, subnets are created for all the availability zones in a given region. You can have up to 200 subnets per region that can be increased further upon request. For enabling high availability, the previously configured AWS services setup needs to adjusted to support high availability.
- HA support for Auto Scaling groups.

To launch the EC2 application instances in different availability zones, the subnets within the Auto Scaling group need to be reconfigured (as subnets are associated with specific Availability Zones).

We can follow the steps listed here to launch the EC2 application:

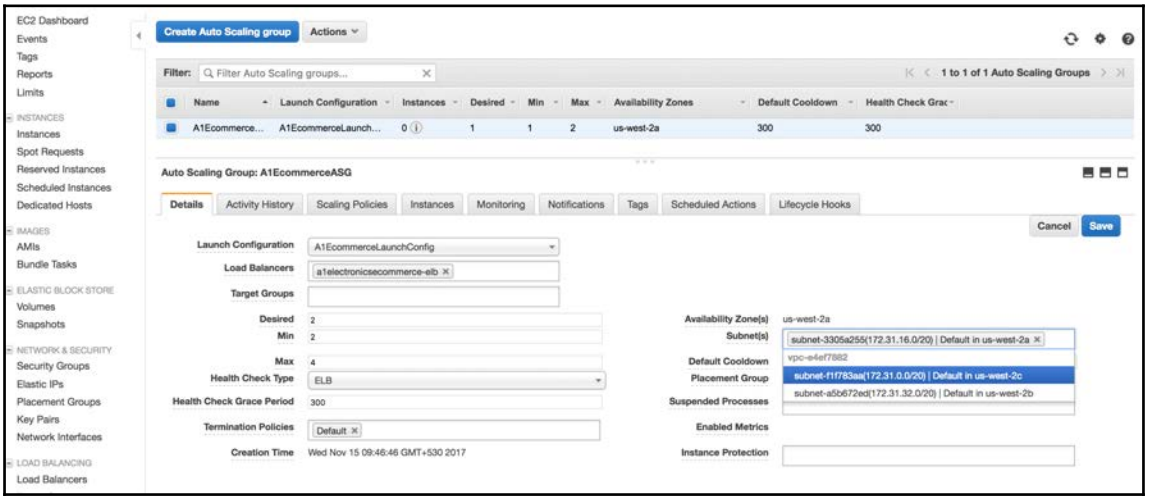
1. From the EC2 dashboard navigation pane, click on **Auto Scaling Groups**. Select the **A1EcommerceASG** Auto Scaling group and then select the **Edit** option in the **Actions** drop-down menu or click on the **Edit** button in the **Details** tab.



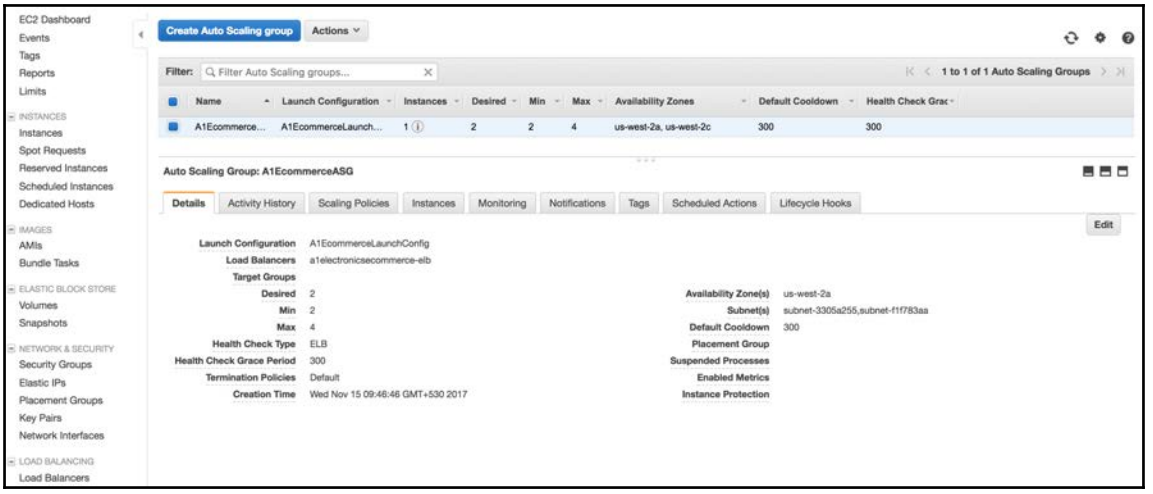
2. The second Availability Zone selected is **us-west-2c** where the new EC2 instances will be launched. Select the subnet associated with the **us-west-2c** Availability Zone from the Subnet(s) dropdown list. This step configures the ASG for high availability.

For high availability, the minimum number of instances running in Auto Scaling group needs to be two, that is, one EC2 instance per availability zone. Modify the **Desired** and the **Min** number of instances to `2` and the **Max** to a number greater or equal to `4`.

Click on the **Save** button to save these changes:



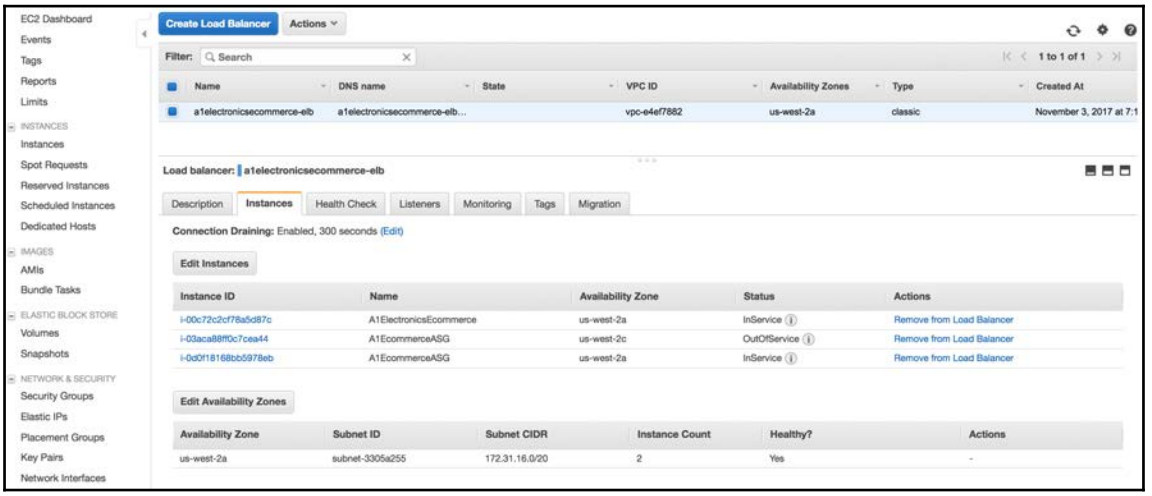
3. As soon as you save the changes, the Auto Scaling group will start a new EC2 instance in the **us-west-2c** Availability Zone as shown:



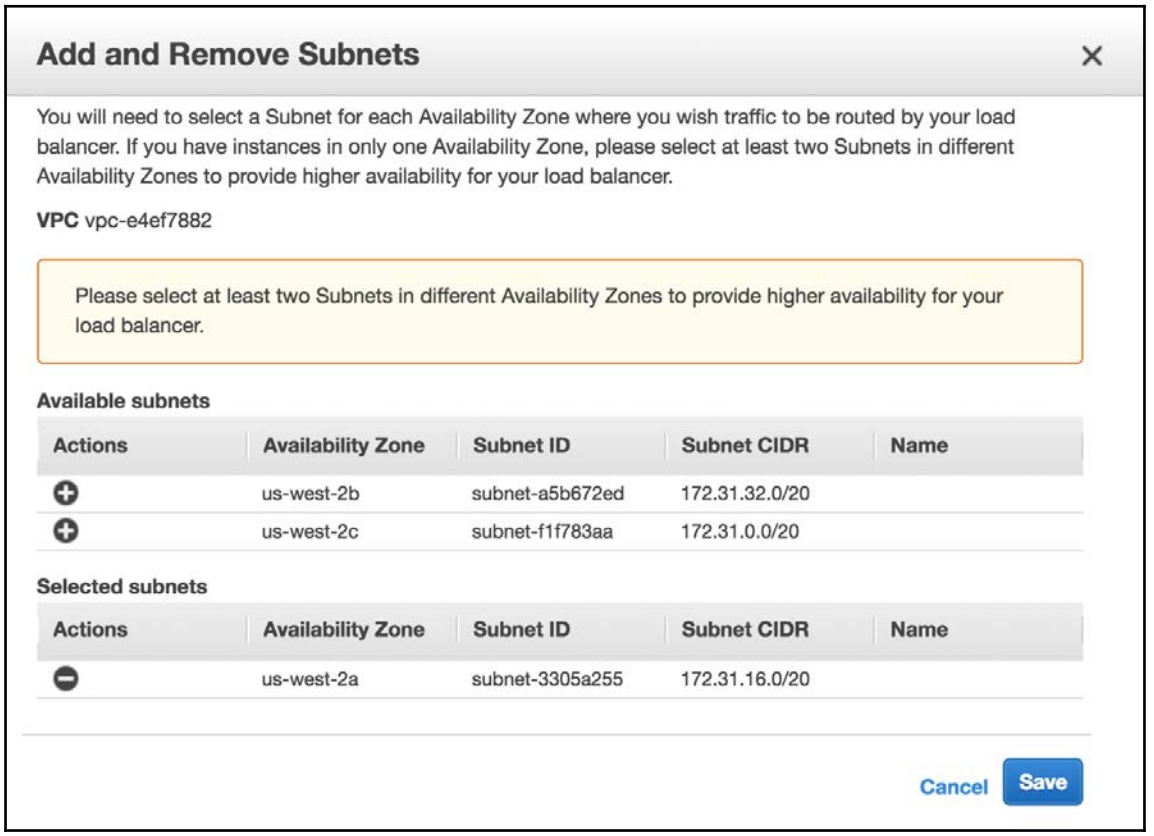
HA support for Elastic Load Balancer

The next step is configuring the ELB to route the traffic to EC2 instances in the added Availability Zone that is **us-west-2c**:

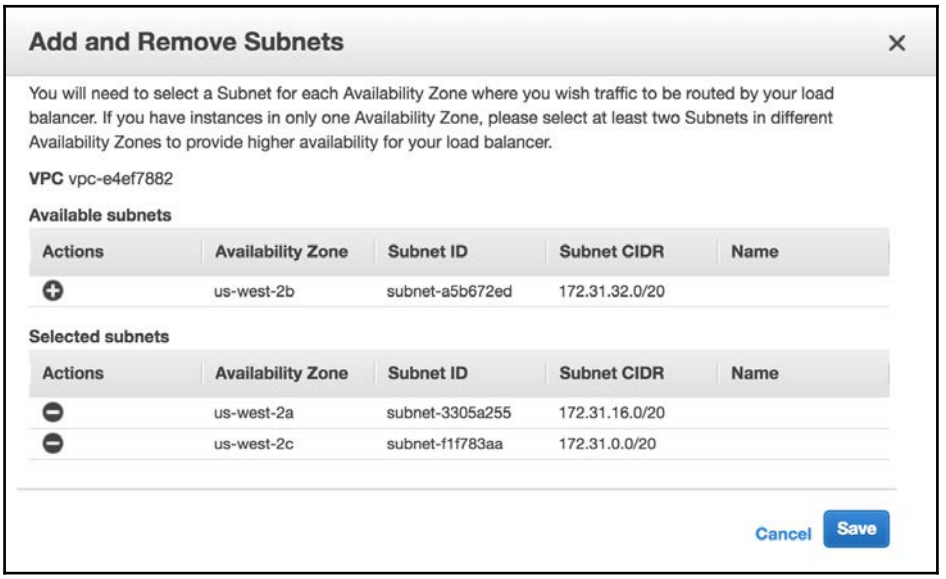
1. From the EC2 dashboard navigation pane, click on **Load Balancers**:



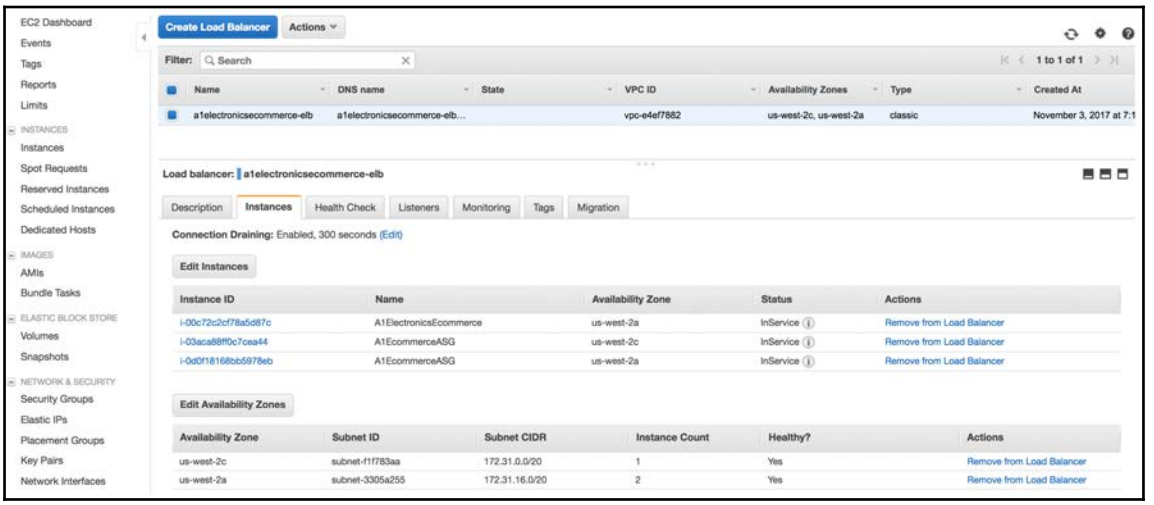
2. Select the **a1electronicsecommerce-elb** load balancer and then click on the **Edit Availability Zones** button in the **Instances** tab:



3. Click on the **us-west-2c** Availability Zone to add it to the ELB and click on the **Save** button:



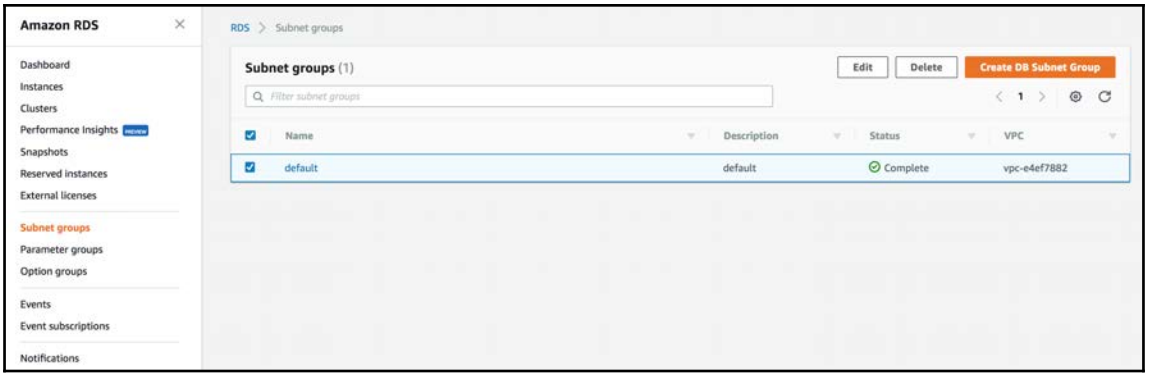
4. You should see the following screen. The ELB starts routing the traffic to the EC2 instances in the **us-west-2c** zone:



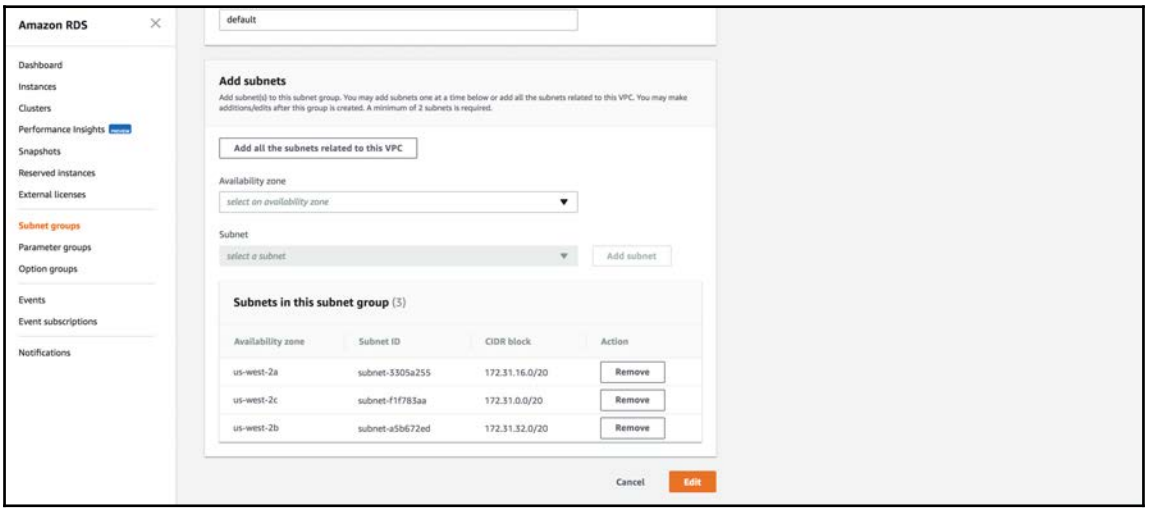
HA support for the Relational Database Service

As explained, RDS provides high availability via Multi-AZ. To set up Multi-AZ, RDS needs to be configured for two things, the availability zone where the slave RDS instance will be launched and instantiated and enable the Multi-AZ option for the current RDS service:

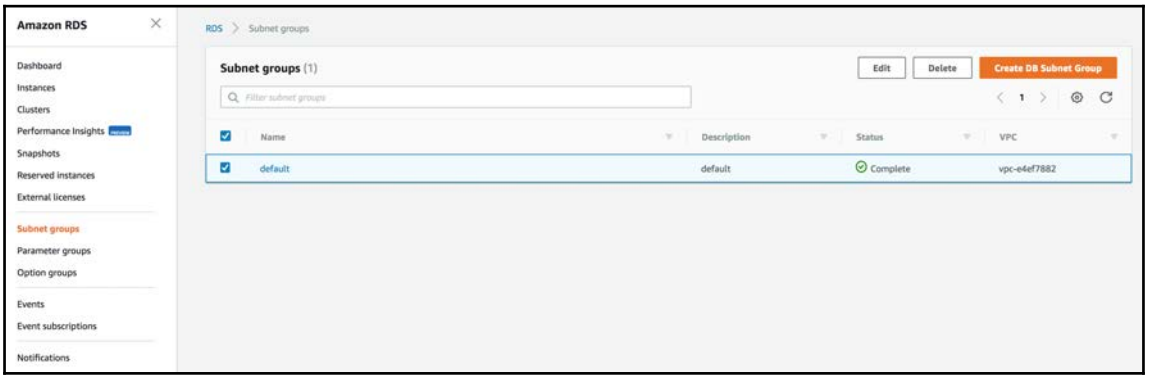
1. From the RDS dashboard navigation pane, click on **Subnet Groups**. Select **default** and click on the **Edit** button:



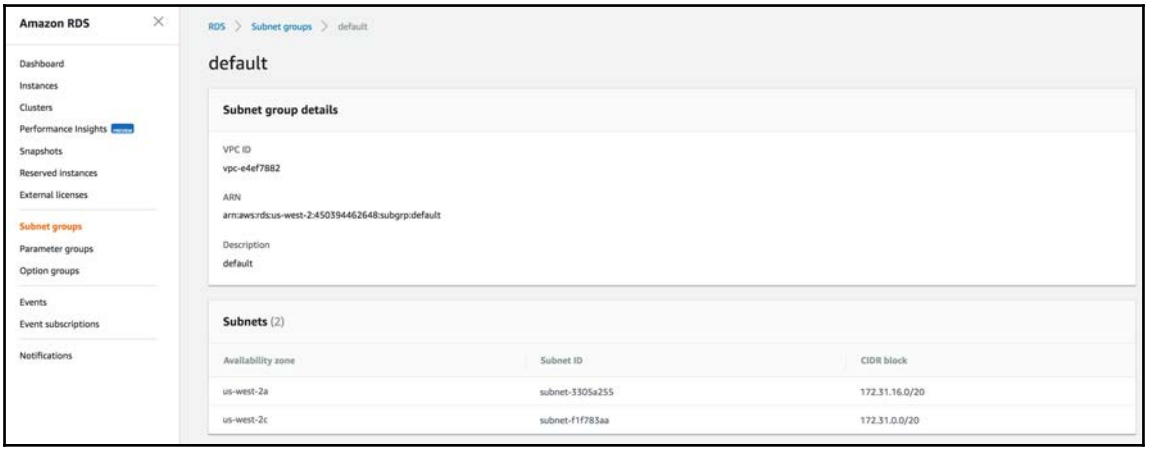
2. The two Availability Zones as per the architecture diagram are **us-west-2a** and **us-west-2c**. Hence, click on the **Remove** button against the **us-west-2b** AZ. Click on the **Edit** button:



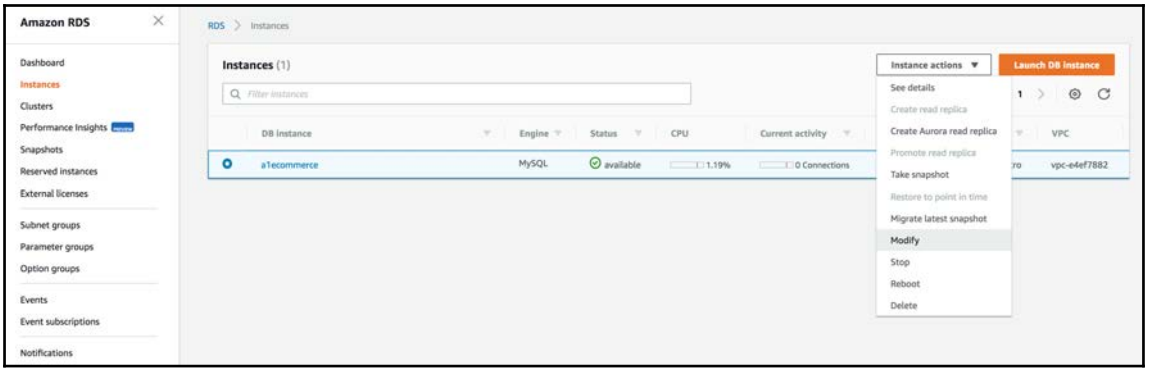
3. You should see the following screen. Click on the **default** hyperlink to the view subnet details:



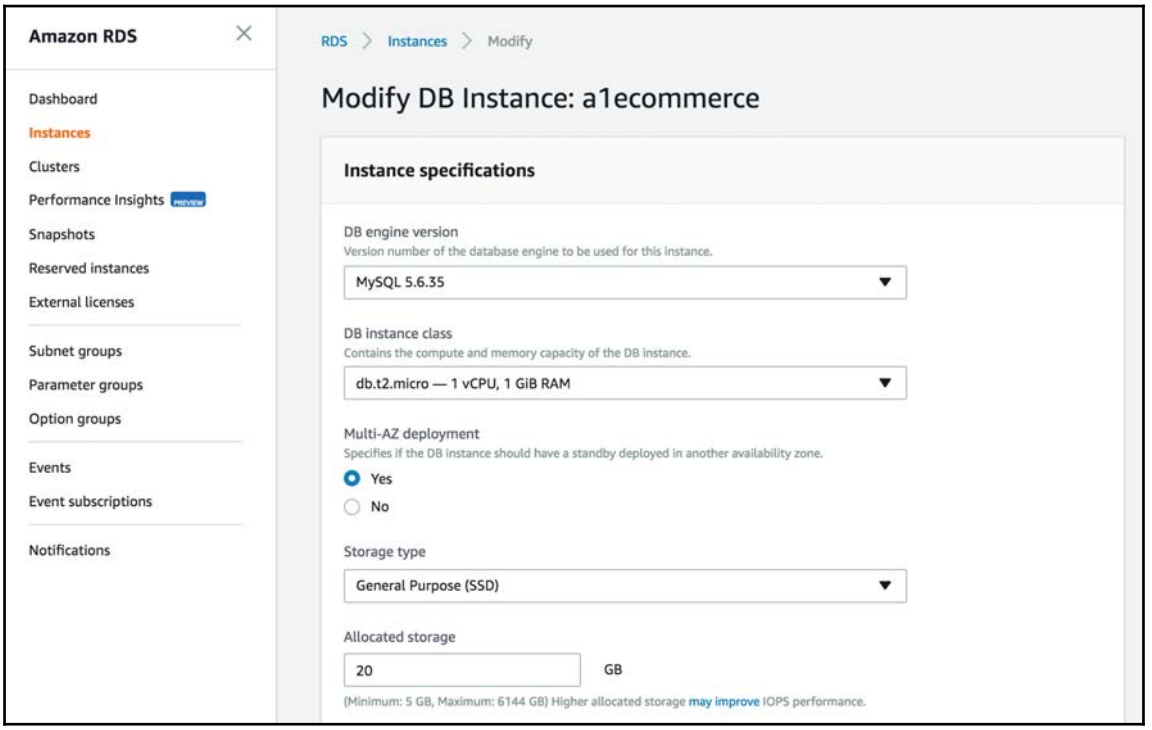
4. Make sure that the two AZs are correctly listed as shown:



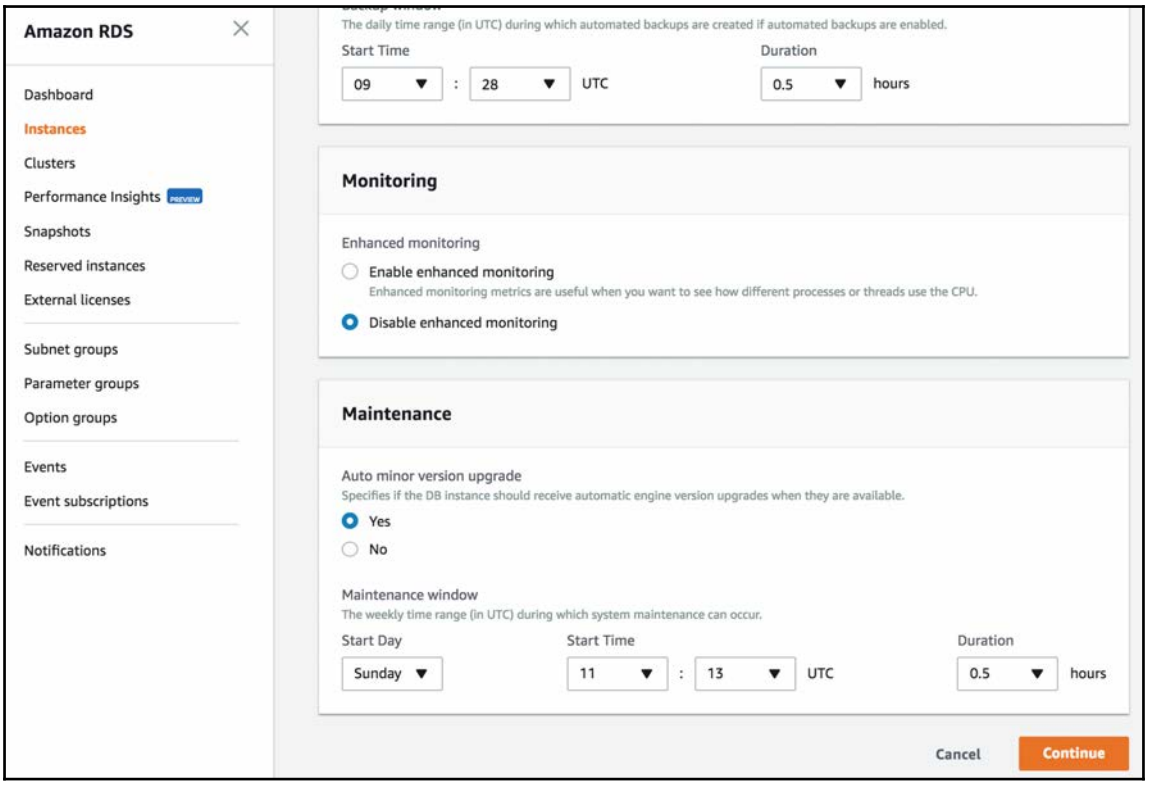
5. From the RDS dashboard navigation pane, click on **Instances**. Select the **a1ecommerce** database instance and click on **Modify** from the **Instance actions** drop-down menu:



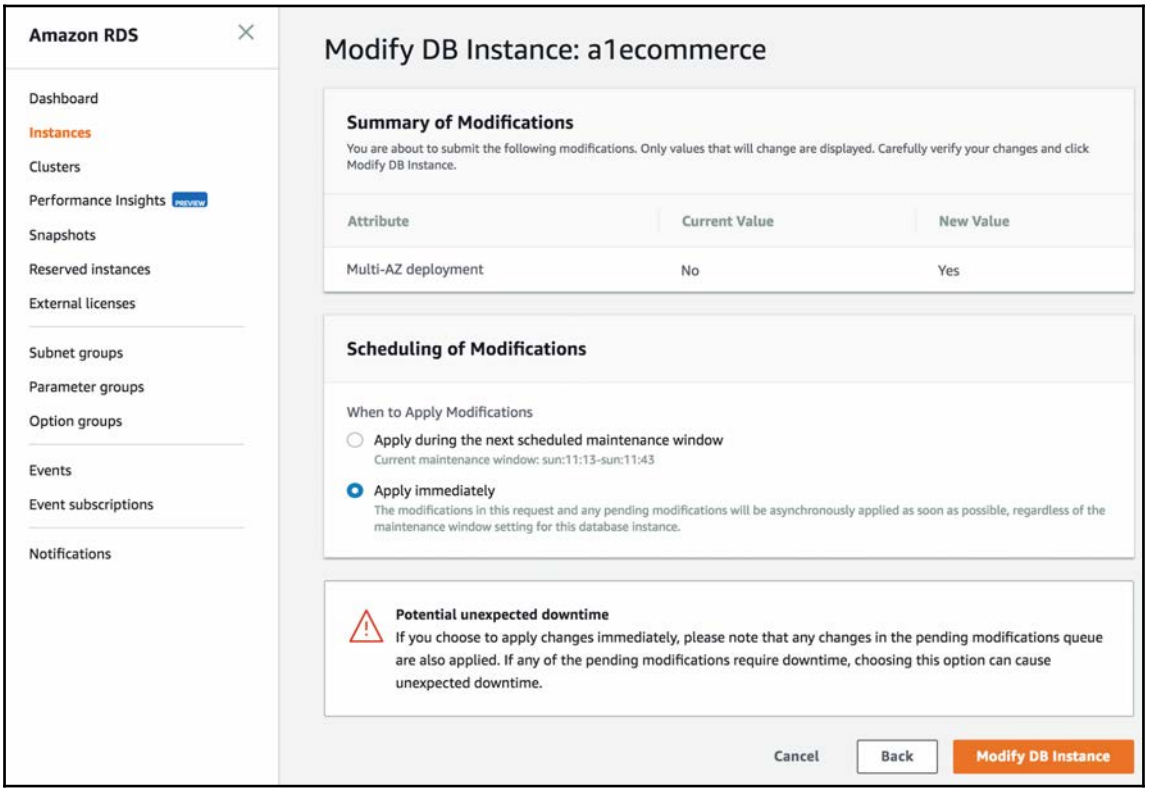
6. Select **Yes** from **Multi A-Z Deployment** drop-down menu. This will instantiate slave database instance in the **us-west-2c** Availability Zone:



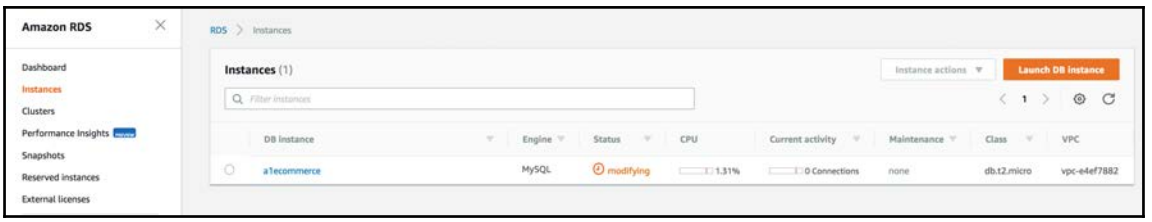
7. Scroll down the page and click on the **Continue** button:



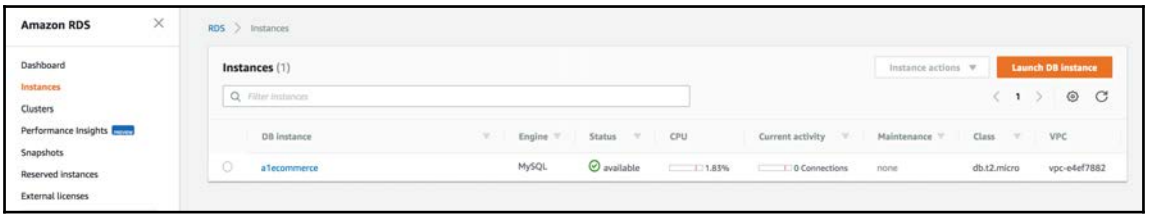
8. Make sure that you check the **Apply Immediately** checkbox; if not, your changes will be scheduled for the next maintenance cycle. Click on the **Modify DB Instance** button:



9. At this stage, you will see the **Status** as **modifying**:



10. After a few minutes, you should see the **status** as **available**:



Summary

In this chapter, we reviewed some of the strategies you can follow in order to achieve high availability in your cloud application. We emphasized the importance both designing your application architecture for availability and using AWS infrastructural services to get the best results. We followed this up with a section on setting up high availability in our sample application.

In the next chapter, we will shift our focus to strategies to design and implement security for your cloud application. We will review some approaches for application security using AWS services. We will also include a hands-on section that will walk you through the process of implementing security in our sample application.

6
Designing for and Implementing Security

In this chapter, we will introduce some key design principles and approaches to achieving security in your applications deployed on the AWS cloud. As an enterprise, or a start-up, you want to ensure your mission critical applications and data are secure while serving your customers. The approaches in this chapter will address security across the layers of your application architecture including security aspects of key infrastructural components. In order to address security requirements, we will use several AWS services, including IAM, CloudTrail, and CloudWatch. Additionally, we will explore AWS Edge services such as CloudFront, Amazon Certificate Manager (ACM), and AWS WAF from a security perspective.

Finally, we will also show you how to implement security for our sample application.

In this chapter, we shall learn about:

- Defining security objectives
- Understanding security responsibilities
- Best practices in implementing AWS security
- Implementing Identity Lifecycle Management
- Tracking AWS API activity using CloudTrail
- Logging for security analysis
- Using third-party security solutions
- Reviewing and auditing security configuration
- Setting up security using IAM roles, the Key Management Service, and configuring SSL
- Securing data at rest, Amazon S3 and RDS

Defining security objectives

Security on the cloud should be a primary area of focus for you because it is a top-of-the-mind issue for your customers. You will need to robustly address these concerns because of the following:

- **Customer trust**: Customers come to our site expecting us to protect their information and keep it safe. We need to live up to that trust.
- **Regulatory compliance**: Increasingly, various regulations and compliance requirements are putting security front and center, especially for cloud-based data storage and applications. And data privacy is a huge part of that.

In order to protect your assets and data on the cloud, you will need to define an **Information Security Management System** (**ISMS**), and implement security policies and processes for your organization. While larger companies may have well-defined security controls already defined for their on-premises environments, start-up organizations may be starting from scratch. However, in all cases, your customers will demand to understand your security model and require strong assurances before they use your cloud-based applications; especially in cases of SaaS or multi-tenanted applications, it can be extremely challenging to collate security-related documentation to meet varying demands, specifications, and the standards of your customers.

There are several information security standards available, for example, the ISO 27000 family of standards can help you define your ISMS. Selecting a control framework can help you cover all bases and measure success against a set of well-defined metrics. Mapping your implementation against the control framework allows you to produce evidence of controls and due diligence to your customers. In addition, you should budget for the expenses and effort required to conduct regular vulnerability assessments and audits. In some cases, be prepared to share these audit reports with your major customers.

In this chapter, we will focus on achieving security objectives for your cloud applications that are also performant, rather than having to choose between being secure and being performant. Additionally, implementation costs can vary widely based on security mechanisms chosen; hence, make your solution choices based on your business needs and risks. As your business evolves, revisit your security plan and make necessary adjustments to better meet your business objectives.

Finally, ensure you build a lot of agility into your processes to keep up with and take advantage of new security-related features and services released frequently by AWS.

Understanding the security responsibilities

AWS security operates on a shared responsibility model comprising of parts to be managed by you and parts managed by AWS. This model consists of three parts—infrastructure security, application security, and services security:

- **Infrastructure security**: AWS has a whole host of industry recognized compliance certifications against various security-centric standards such as Payment Card Industry (PCI), NIST, SSAE, and ISO, as well as PCI DSS 2.0 Level 1, ISO 9001, 27001, 27017, 27018, and so on.
- **Application security**: Services that support security implementation—such as IAM policies, origin protection, ACM integration, keys/certificate rotation, and so on—in applications makes them more secure without sacrificing performance.
- **Services security**: This includes a set of things that Amazon provides by default and what you can do with them to make your applications more secure. For example, the security options and features available on CloudFront across a growing number of edge locations and regions across cities, countries, and continents include a standardized implementation of security features. This ensures that you get a consistent security footprint with CloudFront everywhere. Additionally, AWS provides various features and options such as SSL/TLS options, private content protection mechanisms, origin access identities to protect the origin, a **Web Application Firewall** (**WAF**) to protect against malicious bots, CloudTrail to track the usage of AWS services, and so on.

Essentially, AWS is responsible for managing the security for the virtualization layer, the compute, storage, and network infrastructure, and the global infrastructure (regions, AZs, and endpoints) and physical security. In addition, AWS is responsible for the operating system or the platform layer for EC2 or other infrastructure instances for AWS container services (Amazon RDS, Amazon EMR, and so on). AWS also manages the underlying service components and the operating system for AWS abstracted services (Amazon S3, DynamoDB, SQS, SES, and so on).

Strict controls and procedures are followed by AWS engineers, operations, and others in terms of who can access the edge infrastructure or systems. For maintenance activity, bastion hosts act as a centralized point that the engineers will log in to, and they are the only point from which the edge hosts can be accessed from. Additionally, two-factor authentication is required to access these bastion hosts, and end-to-end encryption is also implemented. Constant testing and probing is done to ensure that the security best practices are being followed.

In subsequent sections, we will present some practical ways to make your content more secure. Hence, we will keep our focus on your responsibilities rather than Amazon's. This includes implementing security controls for users and roles, policies and configuration, applications and data (storage, in transit, and at rest), firewalls, network configuration, and the operating system.

In the next section, we will discuss the basics and best practices of a minimally viable approach—a good starting point to implement some of the security controls that can mature into a comprehensive security strategy over time.

Best practices in implementing AWS security

Typically, you will start with basic security measures in place and then rapidly iterate from there to improve your overall cloud security model and/or implementation. Before designing any of your security solutions, you will need to identify and then classify the assets you need to protect into high/medium/low risk categories. This is often a non-trivial undertaking in large enterprises. Asset data is typically entered manually in most organizations, and it relies heavily on human accuracy. Capturing this data programmatically results in better efficiency and accuracy. Integrate AWS, APIs with your existing enterprise asset management systems, and include your CloudFormation templates or scripts as artifacts in your configuration management database to get a better handle on your cloud assets.

In order to get off the ground faster, take full advantage of everything that is provided out of the box by AWS, whether it is security groups, network ACLs, or the ability to turn CloudTrail on for all your AWS accounts. In addition, we typically implement **Infrastructure as Code** (**IAC**) on AWS Cloud and include security in the whole deployment process. For example, when code is deployed on a new EC2 instance, the OS hardening should happen as a part of the build pipeline.

The AWS **Identity and Access Management** (**IAM**) service is central to implementing security for your applications on AWS cloud. Some of the main activities and best practices for AWS IAM are listed below:

- Use IAM to create users, groups, and roles, and to assign appropriate permissions.

- Manage permissions using groups. You assign permissions to groups and then assign individuals to them. While assigning permissions to groups, always grant the least privilege. AWS provides several policy templates for each of their services. Use these policy templates, as they are a great starting point for setting up the permissions for AWS services. For example, you can quickly set up permissions for a group that has read-only access to S3 buckets.
- In your ISMS, you will need to define a set of roles and responsibilities and assign specific owners to particular security-related tasks and controls. Depending on your choices, these owners may be a combination of people within your organization, or AWS partners, third-party service providers, and vendors. Map each of these owners to appropriate AWS IAM roles.
- Use IAM roles to share access. Never share your credentials for giving access, temporarily or otherwise. Restrict privileged access further by using IAM conditions, and reduce or eliminate the use of root credentials.
- Use IAM roles for getting your access keys to various EC2 instances. This eases rotation of keys as the new set keys can be accessed via a web service call in your application.
- Enable multi-factor authentication for privileged users. For example, users that have permissions to terminate instances.
- Rotate security credentials regularly. Rotate both your passwords and access keys.

There are a few other security-related best practices that are commonly implemented using IAM. For example, you can configure the rules to support your company's password policy. It is advisable to configure a strong password policy for use on the cloud. Other best practices relate to using the AWS Security Token Service to provide just-in-time access for a specific duration to complete a task.



For more details on AWS Security Token Service, refer to: `http://docs.aws.amazon.com/STS/latest/UsingSTS/STSPermission.html`.

Security considerations while using CloudFront

In this section, we will explain the security-related aspects of using CloudFront, as many cloud-based applications use it. In CloudFront, the efficient delivery of dynamic content (not cacheable) is achieved by proxying data to the origin and back over a highly-optimized network. When an end-user makes a request for content, the user is automatically routed to the nearest edge location. A high-quality, consistent connection is maintained between the edge location back to the origin, and the connection is kept alive over the AWS backbone. So, even when you are not caching data, the data going out from your application or coming in from the user can be performant and secure.

CloudFront protects data in transit by delivering content over HTTPS. HTTPS authenticates the viewers to CloudFront and also the origin to CloudFront. The process starts by terminating SSL at the edge location. The communication between the edge location and the user is secured using a SSL certificate. Aiming towards a goal of ensuring secure content delivery, more and more developers are shifting to a complete HTTPS model that covers 100% of your site's contents delivered end to end over HTTPS. CloudFront enables advanced SSL features automatically.

Some of the key security-related CloudFront features include:

- **High security ciphers**: CloudFront uses strong ciphers and cipher suites that are optimized for performance and security.
- **Perfect forward secrecy**: Enables perfect forward secrecy so that even if your certificate is compromised and someone gains access to your private key, they will still not be able to decrypt data collected in the past.
- **Online Certificate Status Protocol (OCSP stapling)**: When the client sends a TLS client connection or "hello" request, CloudFront requests the certificate status from an OCSP responder. The OCSP responder sends the certificate status and CloudFront completes a TLS handshake with the client.
- **TCP fast open**: A TCP cookie returned to the client upon establishing the TCP session. The client sends the cookie the next time it connects to the server, along with the client "hello" request. CloudFront supports this for TLS connections only.
- **Validate origin certificate**: CloudFront validates SSL certificates to the origin, for example, the origin domain name must match the subject name on the certificate; the certificate must be issued by a trusted CA, and the certificate date must be within the expiration window.

- **Session tickets**: We have to do two round trips for the client to establish a TLS connection after the TCP connection (so it is a total of three round trips). Round trips are expensive and add a fair bit of latency. Session tickets allow clients to resume a session. CloudFront sends encrypted session data to the client and the client does an abbreviated SSL handshake.

You can create a site that can deliver both HTTP and HTTPS content; that is, you can use a single CloudFront domain name for both HTTP and HTTPS content. You can choose between enforcing Strict HTTPS, where HTTP requests are failed, or implement an HTTP to HTTPS redirect.

Additionally, CloudFront offers three options for TLS:

- **Default CloudFront SSL domain name**: This option means using a non-human friendly domain name where the CloudFront certificate is shared across customers.
- **SNI-enabled custom SSL**: Allows you to use your own SSL certificate and relies on the SNI extension of the TLS protocol (for the right certificate to be selected for a specific customer). You can have your own domain name and certificate. However, some older browsers/OSs do not support the SNI extension.
- **Dedicated IP custom SSL**: In this option, you can use your own SSL certificate. CloudFront allocates dedicated IP addresses to serve SSL content. It is supported by all browsers/OSs, however, there is an extra charge for using this option.

CloudFront and ACM integration

AWS Certificate Manager (**ACM**) makes it easy to provision, manage, deploy, and renew SSL/TLS certificates on the AWS platform.

ACM makes it easy to procure new certificate (directly from the CloudFront console). It enables extremely fast procurement turnaround times (in minutes), and the certificate is immediately available for use in CloudFront (and ELB). The SNI support of custom certificates generated from ACM comes for free and provides a hassle-free, automatic certificate renewal process.

There are two models for SSL termination that can be implemented:

- **Half bridge termination**: The connection between the edge location and the end user is secured. Connection back to the origin is not secured with HTTPS. It results in better performance as it uses HTTP connections to the origin.

- **Full bridge termination**: The entire access chain is secured. The first connection is terminated at the edge location and a new HTTPS connection established to the origin. This option is usually preferred when collecting data from a user or showing personalized content to the user.

Understanding access control options

There are a variety of access control options available if you want to deliver content only to selected customers, allow access to content only until a certain time, or allow only certain IP addresses to access content. For example, if your site is under development then you can make CloudFront accessible only from your internal IP addresses.

There are two options for controlling access controls to private content:

- **Signed URLs**: Adds the signature to the query string in the URL. Hence, your URL changes. This option is typically used when you want to restrict access to individual files and/or the users are using a client that doesn't support cookies.
- **Signed cookies**: Adds the signature to a cookie. Hence, your URL does not change. This option is typically used when you want to restrict access to multiple files and/or you don't want to change URLs.

Web Application Firewall

Serving unnecessary requests costs money. For example, blocking bad bots dynamically is a typical use case for using AWS WAF. You will need to create an IPSet containing a list of blocked IP addresses and a rule that blocks requests from these IPs. You will define a web ACL which allows requests by default and contains our rule to exclude blocked IPs. Additionally, you will need define a mechanism to detect bad bots and add their IP addresses to IPSet.

You can use `robots.txt` to specify which areas of your site or web app should not be scraped and to ensure there are some links pointing to non-scrapable content. Bad bots (ignoring your `robots.txt`) will request the hidden link and the trigger script will detect the source IP of the request, request a change token, and add the source IP to IPSet blacklist. The web ACL will block subsequent requests from that source.



References to preconfigured rules for blocking IP addresses that exceed request limits, blocking IP addresses that submit bad requests, and so on, are available at: `https://aws.amazon.com/waf/preconfiguredrules/`.

Securing the application

You will need to secure your application and the origin because hackers could bypass CloudFront to access your origin. In this section, we will briefly discuss access control features you can use for restricting access to the origin.

Amazon S3 uses an **Origin Access Identity** (**OAI**) to prevent direct access to your Amazon S3 bucket while ensuring performance benefits for all customers. It works by using a pre-shared secret header and limiting access by whitelisting CloudFront only. Hence, only CloudFront can access the Amazon S3 buckets. However, your origin may not be a S3 bucket, therefore you also need the ability to protect a custom origin. In this case, we whitelist the CloudFront IP range and use a pre-shared secret origin header. You can also configure SNS notifications on any changes made to the IP ranges.

More details on IAM including specific commands for our sample application are presented in a later section of this chapter.

Implementing Identity Lifecycle Management

Typically, establishing robust Identity Lifecycle Management is often considered very late in the development life cycle by organizations offering SaaS applications on a global basis. For example, how do you keep track of active users within your customers' organizations? This can leave you in a situation where an employee having access to your application leaves the customer organization, located in a different time zone. Often it is easiest, from a account management perspective, to have a feature within your SaaS application to create an application administrator role per customer who, in turn, is responsible for managing users within their respective organizations.

AWS Directory Services can help reduce the complexity of managing groups of users. These groups can be mapped to IAM roles for appropriate access to AWS APIs. Organizations can also choose to extend their on-premises directory services to the AWS cloud using Direct Connect.

Tracking AWS API activity using CloudTrail

AWS CloudTrail is a web service for recording API activity (across AWS Console, CLI, or from within SDKs) in your AWS account. It can also record higher-level API calls from AWS services; for example, CloudFormation calls to other services such as EC2. CloudTrail events provide a rich source of information for AWS API calls including the user, timing, nature, resources, and location of an API call. Therefore, CloudTrail logs can be very helpful in incident analysis, tracking changes made to AWS resource configurations, and troubleshooting operational issues.

Logging for security analysis

As a design principle and best practice—log everything. In addition, if you collect all your logs centrally then you can correlate between various log records for more comprehensive threat analysis and mitigation. However, ensure your logging mechanism is scalable and does not unduly impact the performance of your application. For example, you can use SQS with auto-scaling based on queue depth for the logging activity. In addition, you can also use products like Logstash and Kibana to help centralize log collection and visualization. Kibana dashboards are dynamic and support features for drill down, reporting, and so on. In addition, you can automate responses to certain events in your logs using AWS CloudWatch and SNS.

Using third-party security solutions

Familiarize yourself with security offerings available in the AWS Marketplace as there are hundreds of security ISVs and products that can replace what you are doing natively in your application. Partner solution sets can be the answer to your specific situation or application architecture.

In addition, certain enterprise vulnerability scanning software products such as HP Fortify (available as a SaaS service or an on-premises product) or Veracode (SaaS service) can be used to identify vulnerabilities within your application code. These enterprise security tools may be expensive but they are great for the prevention of OWASP top-ten type vulnerabilities in your application, and for promoting secure coding practices in your development teams.

It is important to schedule a penetration test with specialists within your organization and to employ external consultants to ensure your production site is secure. If this is the first time your organization is doing vulnerability scans or getting penetration testing done by specialists, then ensure you allow sufficient time in your project schedule for two or three rounds of testing and remediation work.

Reviewing and auditing security configuration

It is key to regularly review and audit your security controls and implementation using a combination of internal and external audits. They are primarily done to ensure your implementation matches your overall security design and objectives. In addition, these reviews and audits can ensure that your implementation limits the damage in case of any security flaws in your architecture. Overall, these exercises are very useful because they help you remain safe as well as satisfy your customers' security requirements on an ongoing basis.

Typically, these detailed reviews include a review of your network configuration including all your subnets, gateways, ACLs, and security groups. In addition, adherence to IAM best practices, AWS service usage, logging policies, and CloudWatch thresholds, alarms, and responses are also reviewed in-depth.

Your architecture and infrastructure usage will evolve over a period of time. For example, with deployments in new AZs and regions, new roles may get defined, permissions may be created and/or granted, new AWS accounts created, and so on. Verifying changes to your architecture and infrastructure can ensure that you are continuing to meet your security goals.

In the following sections, we will describe the features and walk you through the process of setting up security for our sample application. This will include using IAM roles and the Key Management Service, configuring SSL, and implementing security for data at rest in Amazon S3 and RDS.

Setting up security

This section looks at securing AWS infrastructure and the application. As the AWS security model is a shared one where Amazon is responsible for the security of the infrastructure-like facilities, hardware, network, and some software including virtualization, host operating systems and so on, you as the user are responsible for the security of your software stack, application, updates, data at rest and in transit, data stores, configuration properties, policies, credentials, and the security of the AWS services being used.

Using AWS IAM to secure an infrastructure

AWS Identity and Access Management (**IAM**) is a web service that enables you to manage users and user permissions within the AWS infrastructure. This allows for the central control of users, user access, and security credentials. As there are a plethora of services being offered by AWS, there is a need for authorized users to securely access these services. IAM defines concepts, constructs, and services to achieve this.

IAM solves the following issues:

- **Credential scoping**: Grants access and the required permissions only to the services a user requires. For example, a web application needs write permission to a specific bucket within S3, instead of assigning write permission to all the S3 buckets.
- **Credential distribution**: Facilitates the distribution and rotation of credentials to users, instances, and across applications in a secure manner.
- **Manages access for federated users**: Federated users are users that are managed outside IAM. Typically, these are users in your corporate directory. IAM allows for granting access to the AWS resources to the federated users; this is achieved by granting temporary security credentials to the federated user.

Covering IAM in its entirety is beyond the scope of this book and probably would need a book of its own. In this section, only the pertinent IAM concepts and services are discussed, which cover a general web hosting use case.

Understanding IAM roles

A **role** is a set of permissions that grants access to AWS resources. Roles are not associated with any user or group but instead are assumed by a trusted entity which can be an IAM user, application ,or AWS service such as EC2. The difference between an IAM user and a role is that a role cannot access the AWS resources directly, implying that they do not have any credentials. This property is very useful when the trusted AWS service, such as EC2, assumes a role. There is no need to provide credentials to an EC2 instance. This solves a very important issue—credential distribution and rotation, plus not having the credentials stored as clear text or in an encrypted form.

Since we have already created an IAM role in `Chapter 3`, *AWS Components, Cost Models, and Application Development Environments*, and assigned it to an EC2 instance, we will not go through it again. While assigning permissions to roles, always remember to assign only the required permissions as per the principle of least privileges (`http://en.wikipedia.org/wiki/Principle_of_least_privilege`) .

Let's examine how this works when an application running in an EC2 instance uses an AWS-supplied SDK to access an AWS resource. The SDK API transparently fetches the temporary credentials via the instance metadata service which, in turn, requests the temporary credentials from the AWS Security Token Service. Instance metadata is data about your instance that can be used to configure or manage the running instance. If you are not using an AWS SDK, you can still get the temporary credentials by querying the instance metadata. The instance metadata can be queried from the running EC2 instance from the command line by executing the following command:

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

ami-id
ami-launch-index
ami-manifest-path
block-device-mapping/
hostname
iam/
instance-action
instance-id
instance-type
local-hostname
local-ipv4
mac
metrics/
network/
placement/
profile
```

```
public-hostname
public-ipv4
public-keys/
reservation-id
security-groups
```



For more information on the metadata, please refer to: `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-metadata.html`.

To query the temporary security credentials for a role, execute the following from the running EC2 instance command line:

```
curl
http://169.254.169.254/latest/meta-data/iam/security-credentials/ec2Instances
```

`ec2Instaces` is the name of the role assigned to your EC2 instance. The response will be a temporary security credential which the AWS SDK uses to access the resource:

```
{
  "Code" : "Success",
  "LastUpdated" : "2017-11-29T04:55:57Z",
  "Type" : "AWS-HMAC",
  "AccessKeyId" : "ASIAIGOEGNL3UXEPOELA",
  "SecretAccessKey" : "ZDn5onr2FyIrSyI2AQvq+TDRsHxHQsDHJBXSyROe",
  "Token" :
"FQoDYXdzELb//////////wEaDJOVGyRHYhX2yN8naiK3A+upsj7qMrkM1A3jXM/TyDMNdOEkkp
GGMSnA8IYb4ipwY/9SsWoA71EABjDxskYNbraY55MABvqPzHhforIWnp47Q2XvB7iDLZQh8KXbf
uuy4BlWWKtBbrqlLJZee7cL6mXlsjTYYRnRmasabFGDJ+V/HXA9hsQ4FwybdRVhvXVLlgdIUoUV
13gxTfUr7GbKyLzVYZZD6K7nFUFwttp34WWA2Y6xKB7inCfmlS9jGyB+73XxoeeeDZAflb6NI6w
5JmzOHNqfeBatJUwzJ1ppsP+wGMSkTpsIg/I30DN7TzPTGbbiH76tBJZO7bTfry8IDZVakPVC3M
p4KGJaM6NTDgyqjRnsNsFF9qyXTKiT6GCxRWFzTSis0oDIw95cPZR0OUiXfAurz637ul8FUAtxC
MOsyfxsEwp0hkykMvhEgUp3SOt7tPwWTNkaJj7HlPmC9PXlBXNpRtR1x2ju7C6ZBvyNJsviphZo
fKcSv+MvC1sFHgWK+L+6tEH9NMOqfqvhKwKNxfuZks70Ky59cX3yl+/qgWSEhzov6dREkz5LUbC
4xvmCc8TQKaxMrTMo5bardFcwLpRNgV0orfv40AU=",
  "Expiration" : "2017-11-29T11:28:08Z"
}
```

The temporary credentials are automatically rotated and have an expiry date/time associated with them. The application has to query the instance metadata for the new credentials before the current credential expires. If the AWS SDK is being used within an application, then it manages it transparently and no credential key refresh logic is necessary.

Using the AWS Key Management Service

We all have used encrypted data in some application or other, and the biggest challenge has always been how to effectively hide the encryption key, the key with which the data is encrypted within the application or the OS using different mechanisms. In the end, there will always be a key that will be in clear text, which will unlock other keys or the encrypted data. This is just for a single application. Now imagine if you have dozens of applications running on the cloud. The challenge of key distribution and the effort to keep the key secret multiplies exponentially.

With KMS, the master key is never released, enabling you to encrypt and decrypt data. AWS Key Management Service manages the following issues:

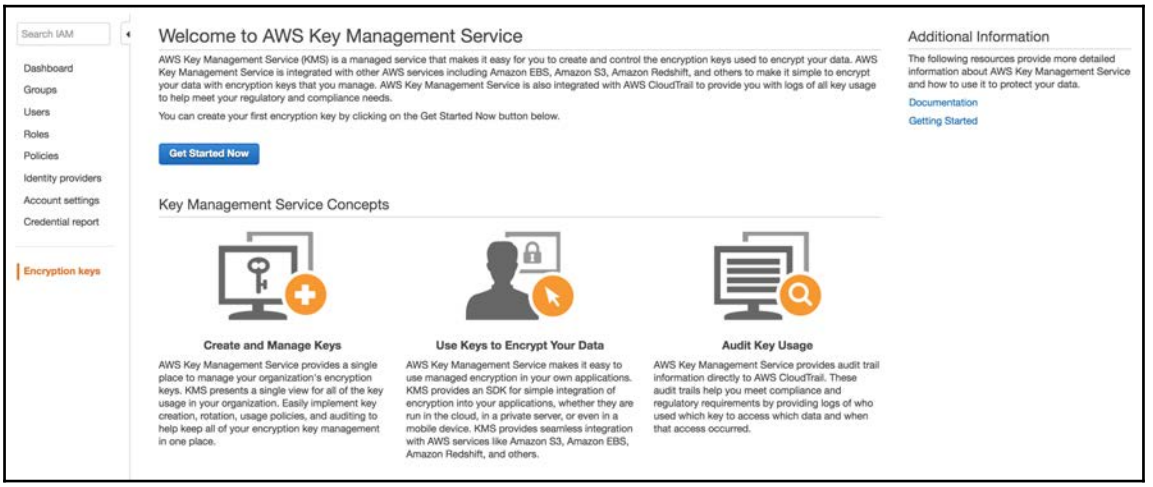
- **Encryption for all your applications**: Manages encryption keys used to encrypt data stored by your applications regardless of where you store it. KMS provides an SDK for the programmatic integration of encryption and key management.
- **Centralized Key Management**: Provides centralized control of your encryption keys, and presents a single view of the keys' usage. Allows for the creation of keys, implements key rotation, creates usage policies, and enables logging.
- **Integrated with AWS services**: Integrated with other AWS services such as S3, Redshift, EBS, and RDS to make it easy to encrypt stored data.
- **Built-in auditing**: Logs all API calls to KMS or AWS CloudTrail. Helps to meet compliance and regulatory requirements by providing details of when keys were accessed and who accessed them. A log file is delivered to your specified S3 bucket.
- **Fully managed**: It is a fully managed service; AWS handles the availability, physical security, and hardware maintenance of the underlying infrastructure.
- **Low cost**: Low costs based on usage.

Let’s get started and create a master key, and then use it to encrypt and decrypt data.

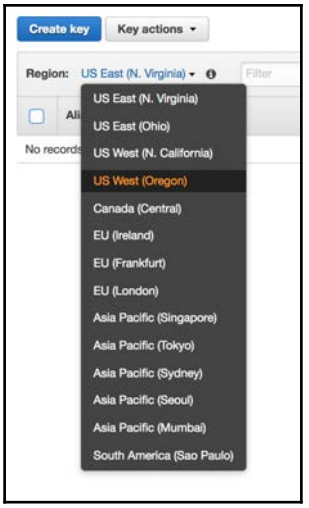
Creating KMS keys

In this section, we will present the steps for creating KMS keys:

1. From the IAM dashboard navigation pane, click on **Encryption Keys** and then on the **Get Started Now** button to create a new master encryption key:



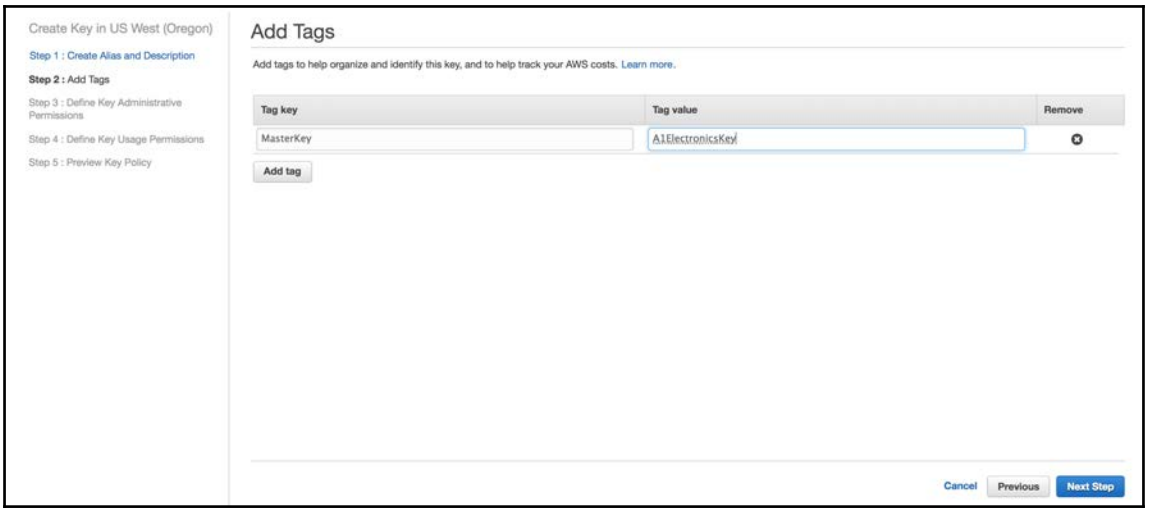
2. Select **US West (Oregon)** from the **Region** drop-down list:



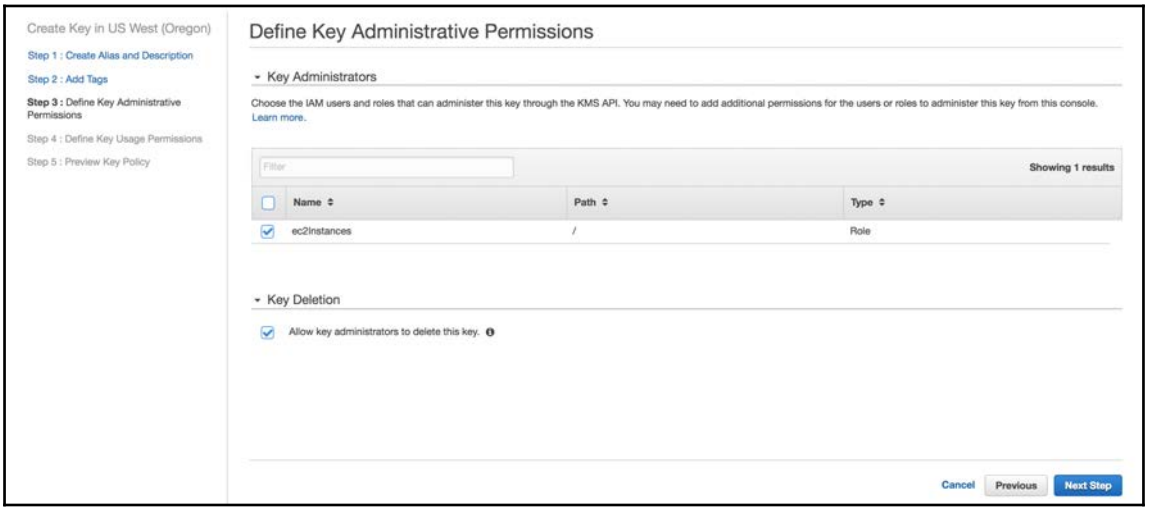
3. In this step, we create an alias:
    - **Alias (required)**: The alias is a display name that is used to easily identify the key. The alias must be between 1 and 32 characters long. An alias must not begin with **aws** as those are reserved by Amazon Web Services to represent AWS-managed keys.
    - **Description**: The description can be up to 256 characters long and should tell users what the key will be used to encrypt.
    - Click on **Next Step**, which will configure the users who administer the key:



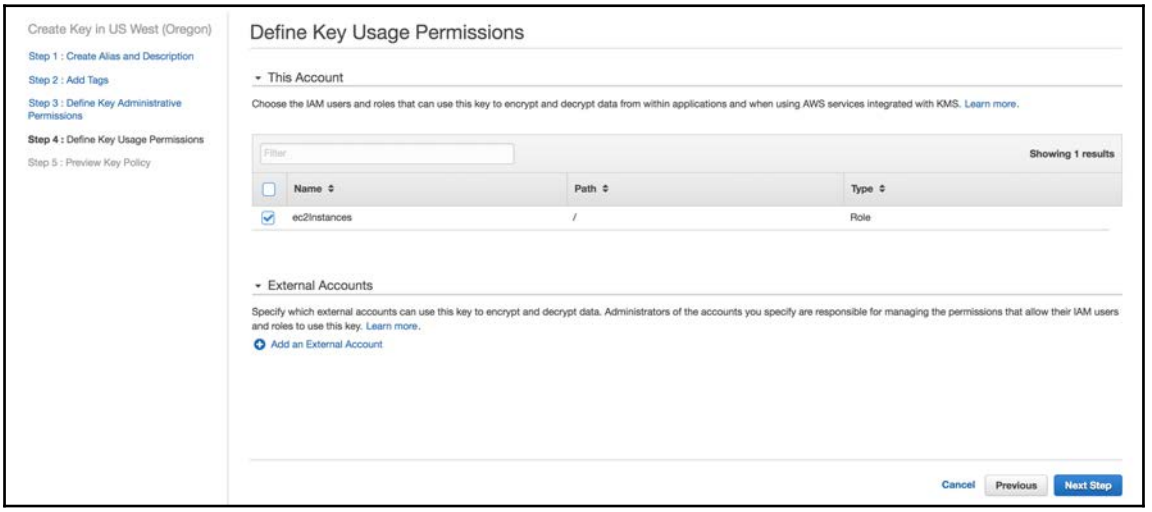
4. Next, we specify a tag:



5. Next, we select the IAM role to define the administrative permissions. In this step, you associate the users/roles who have administration rights to this key. The administration rights are for enabling or disabling a key, the rotation of keys, and adding users/roles who can use the key. In our example, IAM group admin users is selected. Click on **Next Step**, which will configure the users who can use the key to encrypt and decrypt the data:



6. Next, we define the key usage permissions. We assign usage rights to the IAM users/roles. Usage rights in this context means to encrypt and decrypt data using this key. Click on **Next Step** to review the key policy:



7. We review the policy and click on the **Finish** button to complete the process. You can now review the policy before creating it. Click on **Finish** to create the new master key:



8. You should see the following success message:





Note that after a key is created, it cannot be deleted; it can be only enabled or disabled.

Using the KMS key

In the previous step, we created a master key; now we will use this key to encrypt and decrypt data in the application. The use case is in the properties file. The database password needs to be kept in encrypted format.

The following is a Java class used to encrypt and decrypt the data using KMS. Use this class to first encrypt the data and then use the encrypted string in the properties file. Replace the `keyId` in the following code with the ARN of the key you created in the previous section. The ARN of the key can be viewed by double-clicking on the key you want to use from the Encryption Keys screen from the IAM dashboard. Remove the credentials if you are running it within the EC2 instance:

```
public class KMSClient{
    private String keyId = "arn:aws:kms:us-
west-2:450394462648:key/1cd0e2d5-61e1-4a71-a6b2-b9db825c9fce";
    private AWSCredentials credentials;
    private AWSKMSClient kms;
```

```
    public KMSClient(){
        credentials = new BasicAWSCredentials(accessKey, secretKey);
    kms = new AWSKMSClient(credentials);
    kms.setEndpoint("kms.us-west-2.amazonaws.com");
 }

 public String encryptData(String plainText) {
     ByteBuffer plaintext = ByteBuffer.wrap(plainText.getBytes());
     EncryptRequest req = new
EncryptRequest().withKeyId(keyId).withPlaintext(plaintext);
     ByteBuffer ciphertext = kms.encrypt(req).getCiphertextBlob();
     String base64CipherText = "";
     if (ciphertext.hasArray()){
         base64CipherText=Base64.encodeAsString(ciphertext.array());
     }
     return base64CipherText;
 }

 public String decryptData(String cipherText) {
     ByteBuffer cipherTextBlob = null;
     cipherTextBlob = ByteBuffer.wrap(Base64.decode(cipherText));
     DecryptRequest req = new
DecryptRequest().withCiphertextBlob(cipherTextBlob);
     ByteBuffer plainText = kms.decrypt(req).getPlaintext();
     String plainTextString = new String( plainText.array(),
java.nio.charset.StandardCharsets.UTF_8 );
     return plainTextString;
  }
}
```

Application security

Regarding application security, we will explore:

- Securing the data between the endpoints while it is being transported to prevent a man-in-the-middle attack.
- Encrypting and storing the data at rest.
- Encrypting all the critical data, including passwords and keys used by the application. We have already covered this previously in the *Using the AWS Key Management Service* section.

Implementing transport security

Security for transporting data over HTTP is provided by a Secure Sockets Layer (SSL). SSL is widely used on the internet to authenticate a service to a client, and then to provide encryption to the transport channel. Since on AWSone of the endpoints is the user's browser and the other is the **Elastic Load Balancer** (**ELB**), which was configured earlier in `Chapter 4`, *Designing for and Implementing Scalability*, configuring the ELB to accept SSL certificates will secure the transport channel between the user's browser and the ELB. This implies the data is not secured between the ELB and the application running in an EC2 instance, but since it is on a VPC within the AWS infrastructure it is secure.

Digital certificates are issued by **Certification Authorities** (**CAs**) who are trusted third parties that sign certificates for network entities they have authenticated using secure means. Normally, you would create a CSR and have the CSR signed by the CA. Here we will not use a commercial CA to sign a certificate but instead use a self-signed certificate. As a consequence of that the browser will not be able to verify the self-signed digital certificate or the authenticity of the website and will generate an exception. However, it will create a secure transport channel between the browser and the ELB.

Generating self-signed certificates

We will use `openssl` to create the keys and the certificates, so make sure you have it installed on your development machine. From the command line, execute the following command (on an OS X or Linux machine):

```
openssl req -x509 -newkey rsa:2048 -keyout key.pem -out cert.pem -nodes -
days 3650
```

This creates a 2048 bit RSA private key (in the `key.pem` file). The private key is used to sign the certificate (the `cert.pem` file). While generating the signed certificate, ensure you enter the correct information for Common Name (for example, server FQDN or YOUR name). Here, we have used the ELB public DNS name:

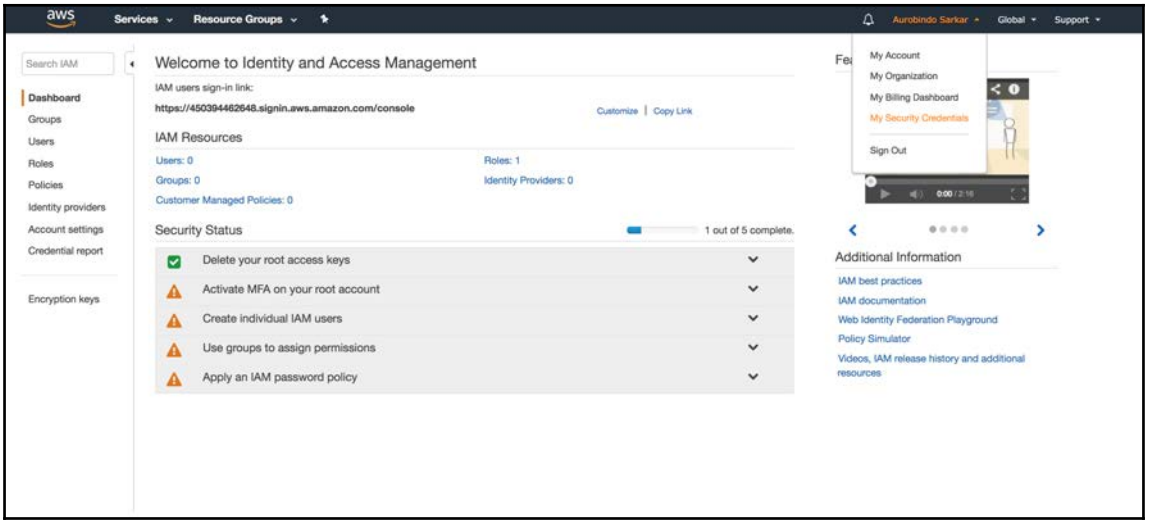
```
Generating a 2048 bit RSA private key
..........+++
.....................................................................
.................................+++
writing new private key to 'key.pem'
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
```

```
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:CA
Locality Name (eg, city) []:Irvine
Organization Name (eg, company) [Internet Widgits Pty Ltd]:A1Electronics
Organizational Unit Name (eg, section) []:Software Engineering
Common Name (e.g. server FQDN or YOUR name) []:a1electronicsecommerce-
elb-965226090.us-west-2.elb.amazonaws.com
Email Address []:admin@a1electronics.com
```

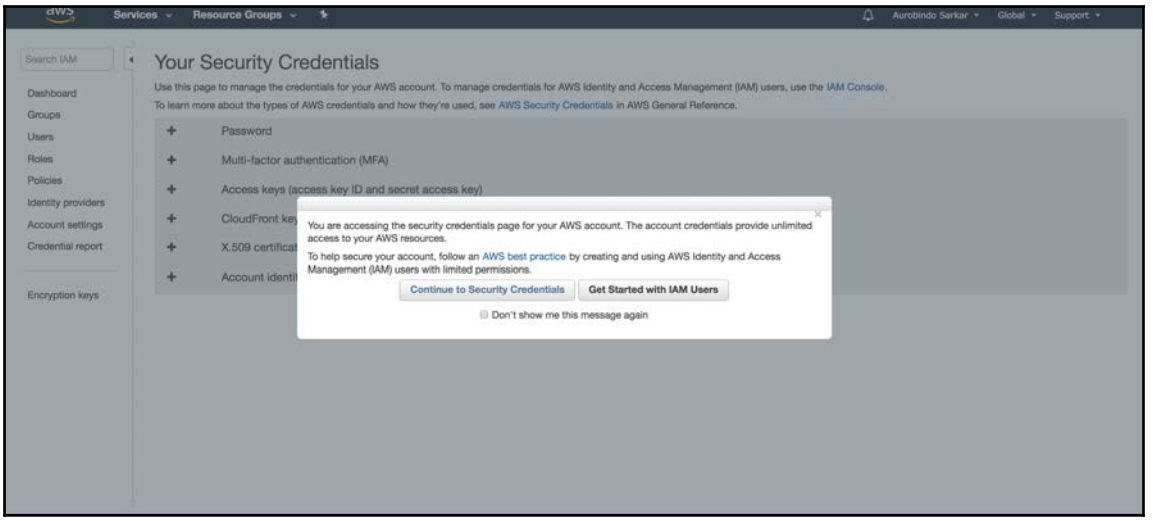
Configuring ELB for SSL

The next step is to configure the ELB to support SSL using AWS CLI (instead of the management console):

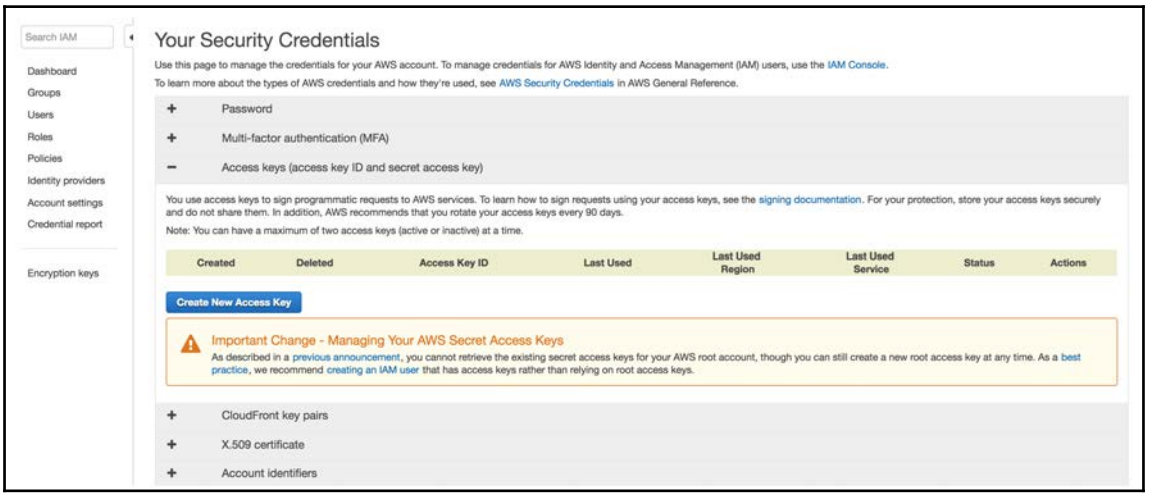
1. First we need to generate security credentials to access AWS services from our development machine. Select **My Security Credentials** from the account drop-down menu:



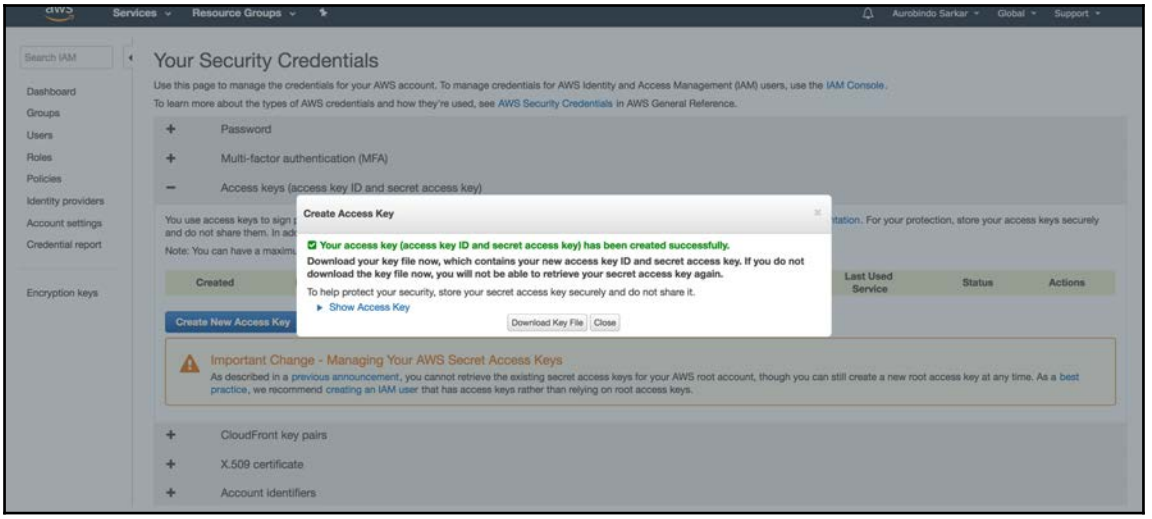
2. Click on the **Continue to Security Credentials** button:



3. Next, click on the **Create New Access Key** button:



4. Click on the **Download Key File** button (it contains your access and secret keys):



5. Install AWS CLI by executing the following command:

```
pip install awscli --upgrade --user
```

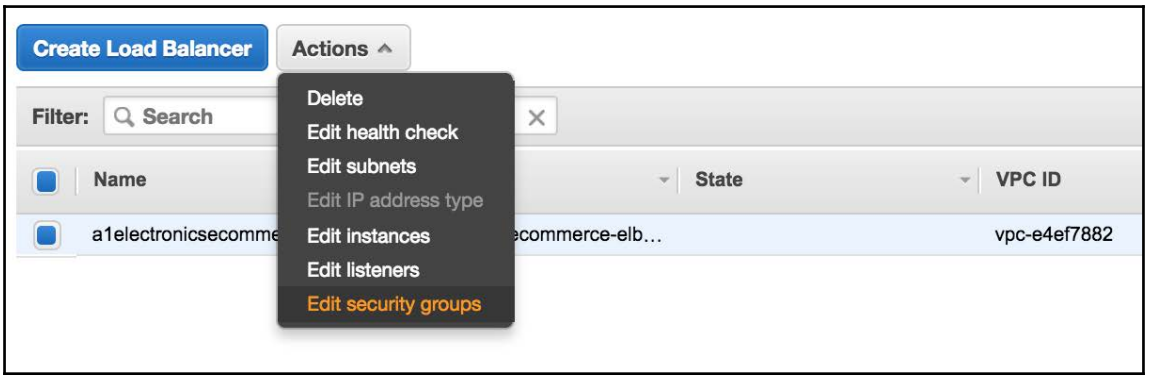
6. Include the library in your path by executing the following command (you can include it in a shell script as well):

```
export PATH=~/.local/bin:$PATH
```

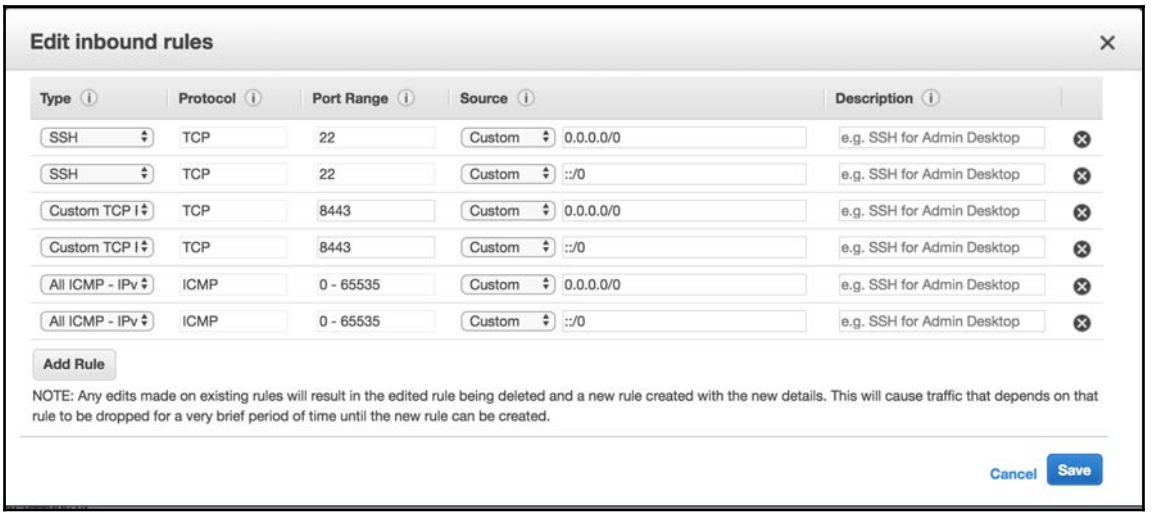
7. Execute the following command using the previously generated certificate and private key files:

```
aws iam upload-server-certificate --server-certificate-name
a1SelfSignedCertificate   --certificate-body
file:///Users/aurobindosarkar/Downloads/cert.pem --private-key
file:///Users/aurobindosarkar/Downloads/key.pem
{
    "ServerCertificateMetadata": {
        "ServerCertificateId": "ASCAI3QWEMZYFDPHV4SDA",
        "ServerCertificateName": "a1SelfSignedCertificate",
        "Expiration": "2027-11-27T07:58:58Z",
        "Path": "/",
        "Arn": "arn:aws:iam::450394462648:server-
certificate/a1SelfSignedCertificate",
        "UploadDate": "2017-11-29T08:34:12.463Z"
    }
}
```

8. Next, configure the security group to add a custom TCP rule to accept data on port `8443`. From the EC2 dashboard, navigate to **Load Balancers** and click on the **Edit security groups** option:

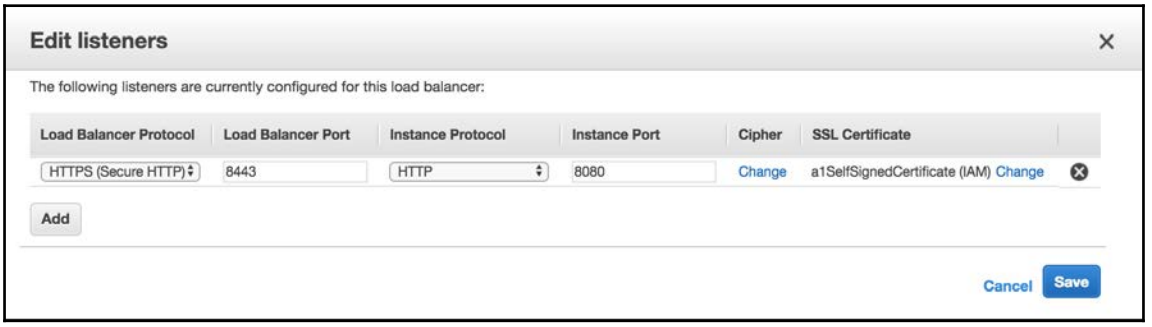


9. In our example, the security group is `sq-EC2WebSecurityGroup`. Click on **Edit** in the **Inbound** tab to add the TCP rule to accept data on port `8443`. Delete the `Custom TCP Rule` on **Port Range** `8080` as it being replaced by the `8443` port:

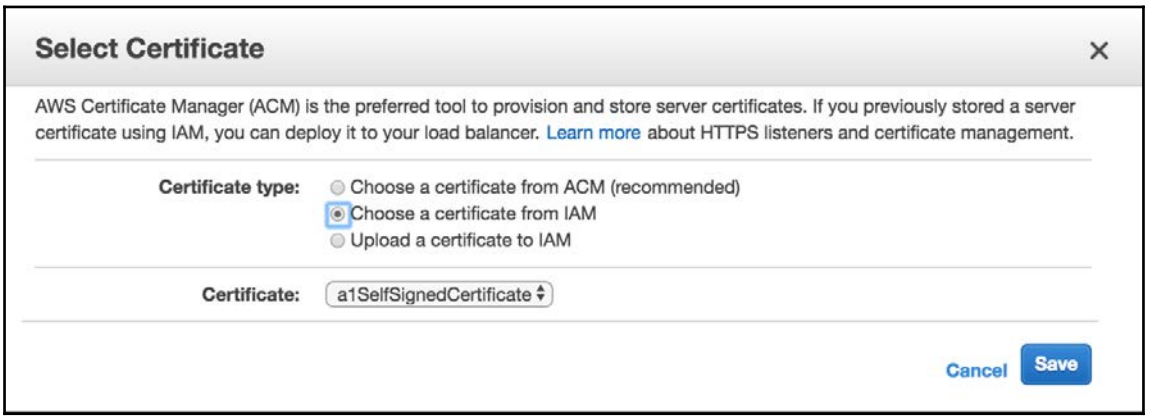


10. The next step is to add/configure the private and the public key on the ELB:
    - From the EC2 dashboard, navigate to **Load Balancers**, click on the **Listeners** tab, and then click on **Edit**.
    - From **Load Balancer Protocol**, select `HTTPS` protocol.

- Set the **Load Balancer Port** to `8443`; this is the port we added to our security group in our previous step.
- From the **Instance Protocol**, select `HTTP`; this is the protocol between the ELB and the EC2 instances.
- Set the **Instance Port** to `8080`; this is port the Tomcat is listening on.
- From the **Load Balancer Protocol**, delete the HTTP protocol as it is not needed anymore.



11. The next step is to associate the SSL certificate with the ELB. Click on **Change** under the **SSL Certificate**:
    - **Certificate type**: Make sure the option for **Choose a certificate from IAM** is selected
    - **Certificate**: Select the name of the certificate uploaded earlier
    - Click on the **Save** button:



This will configure ELB to support the SSL protocol. Test the URL on the browser using the HTTPS protocol.

Securing data at rest

Another key aspect of security is to secure the data stored in physical storage devices such as hard disks, USB drives, SAN devices, and so on. In the AWS cloud world, these would be AWS data storage services such as S3, RDS, Redshift, DynamoDB, and so on. To secure data at rest, symmetric encryption is used; that is, the data is encrypted with an encryption key, and the data is secure as long as the encryption key is secure, so all effort is directed at keeping the encryption key secure.

AWS provides the Key Management Service (KMS) to resolve issues related to the management and storage of encryption keys, as described in the previous section. This service is also used to secure data at rest. The encryption of data at rest is a key component of regulations such as HIPPA, PCI DSS, SOC 1, 2 , 3 and so on. In the upcoming sections we walk you through the process of securing the data at rest for S3 and RDS.

Securing data on S3

To secure the data at rest within S3, broadly there are two options:

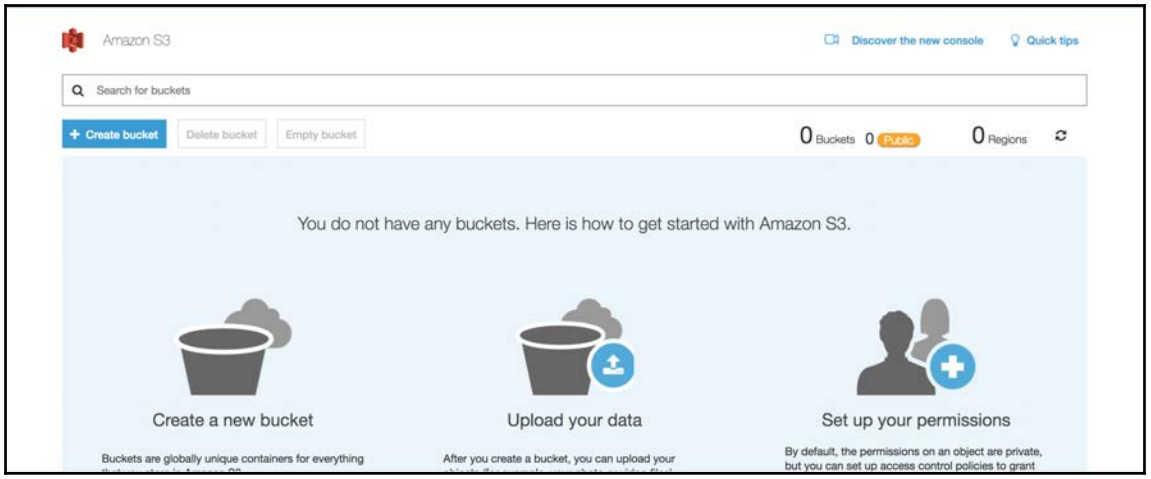
- **Server-side encryption**: Amazon S3 encrypts your object before saving it and decrypts it when you retrieve the objects. The encryption and decryption process is totally transparent and seamless. Amazon S3 can be configured in multiple ways for the encryption keys.
- **Client-side encryption**: The client is responsible for encryption of the object before uploading to Amazon S3, and for decrypting the object after it has been retrieved. The client is responsible for the encryption/decryption process and management of encryption keys.

Using the S3 console for server-side encryption

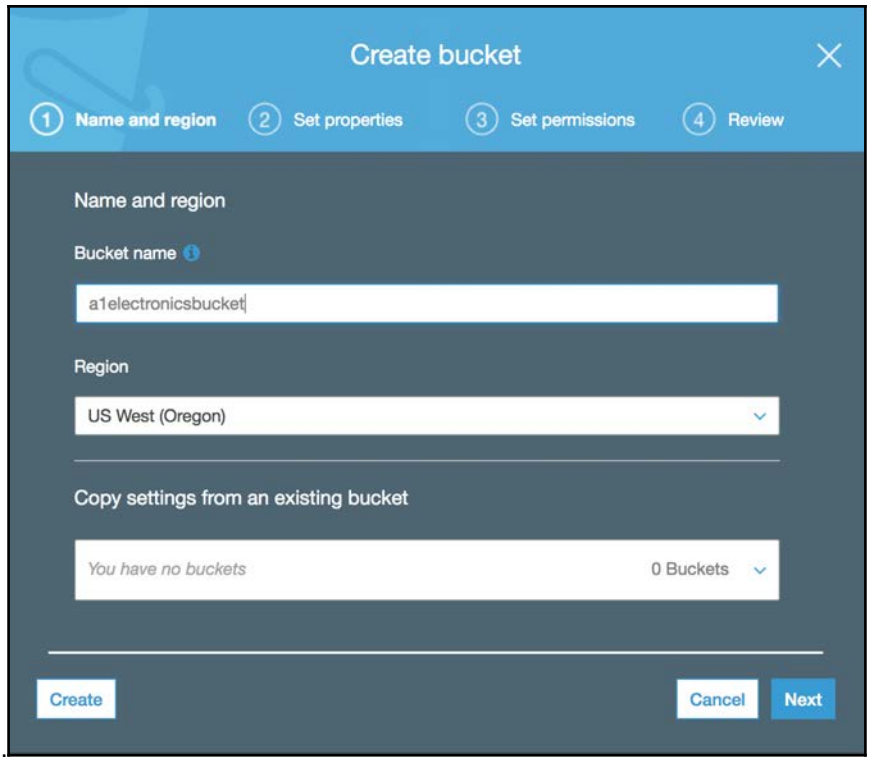
The easiest way to secure data on S3 is via the S3 console.

As we don't have any S3 buckets in our sample application, we will create the bucket and configure it to store data in encrypted form:

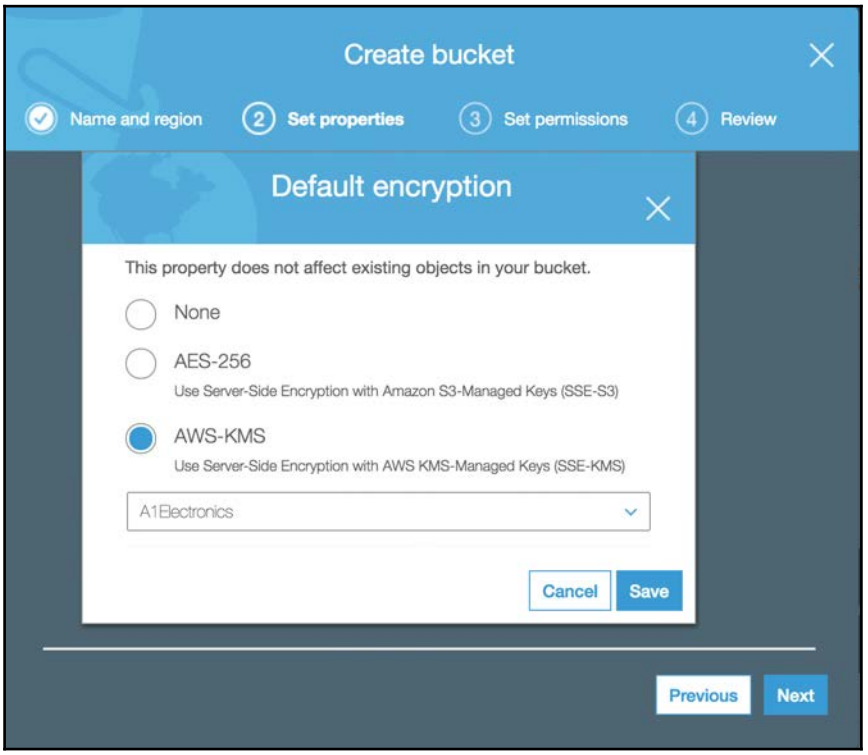
1. In the Amazon S3 console, click on the **Create bucket** button:



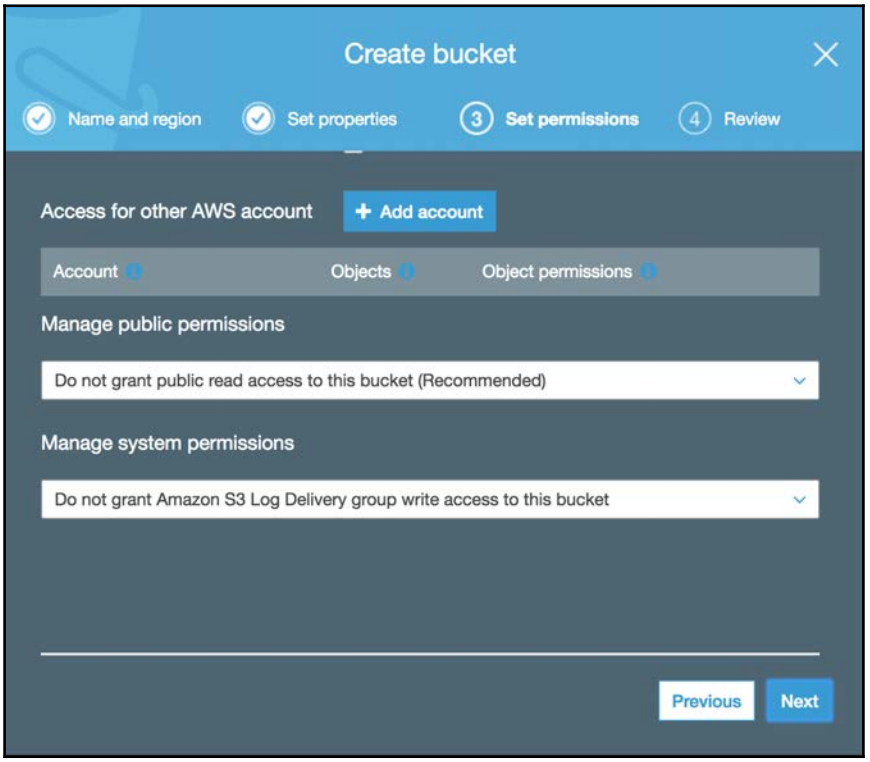
2. In the **Create bucket** pop-up window, specify a name for the bucket and the region. Click on the **Next** button:



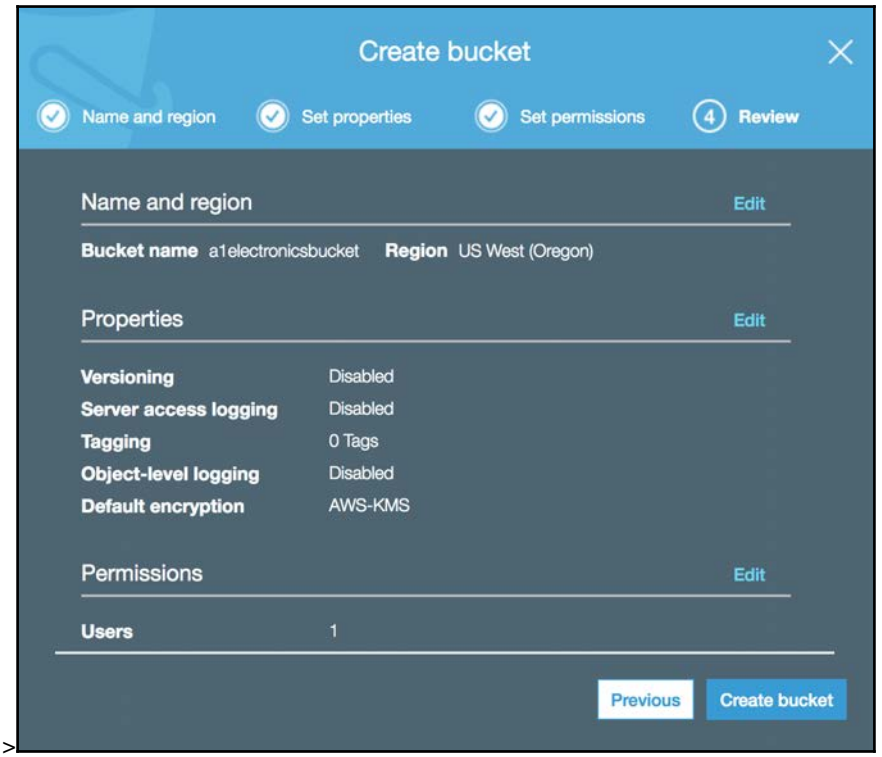
3. Select the **AWS-KMS** option for server-side encryption using AWS KMS Manager:



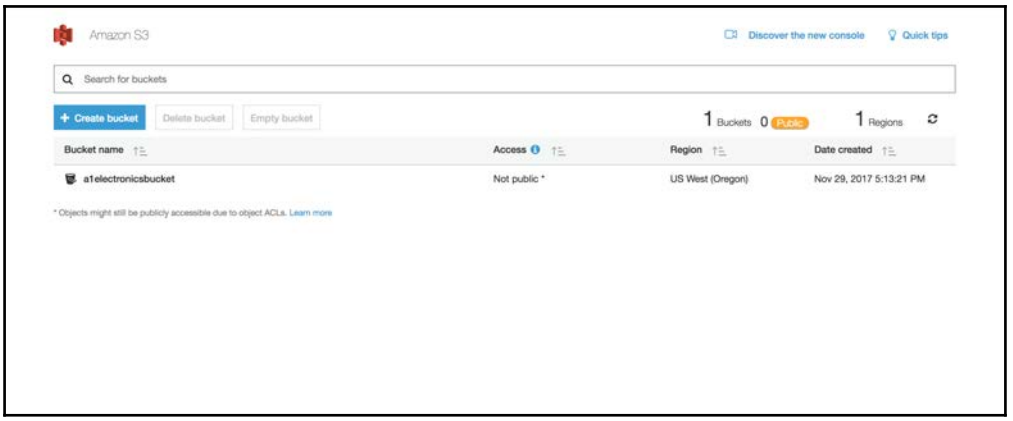
4. Next, specify the permissions on the bucket. Here, we do not grant public access to the bucket as we only access S3 from within our application. Click on the **Next** button:



5. Review the information presented and click on the **Create bucket** button:



6. You should see your newly created S3 bucket as shown:



Securing data on RDS

The RDS service secures the database by encrypting the database volume with the specified encryption key from the KMS. Note that RDS does not encrypt the database at the application level; it encrypts the complete database volume at the OS file level.

The data stored in the database rows is in plain text; the application does not need the encryption key to decrypt the data. If an unauthorized user gets hold of the database volume, it will be of no use to him/her since it is encrypted, and without the encryption key it cannot be decrypted. The option to encrypt the database volume is available at the time of database creation.

Summary

In this chapter, we reviewed some of the strategies you can follow for achieving security in your cloud application. We emphasized the best practices of implementing security using AWS services. We followed this up with several sections on setting up security in our sample application.

In the next chapter, we will shift our focus to production deployments, go-live planning, and operations. We will also discuss data backup and restore, and application monitoring and troubleshooting. We will also include a hands-on section that walks you through these processes for our sample application.

7
Deploying to Production and Going Live

In this chapter, we will focus on making your application live on the cloud. As an enterprise or a start-up, you want to ensure your applications are deployed and supported to best serve your customers. We will discuss the tools, approaches, and best practices in deployment and operations that ensure smooth functioning of your applications in production environments. We will also show you how to deploy the sample application in a production environment.

In this chapter, we shall learn about:

- Managing infrastructure, deployments, and support at scale
- Creating and managing AWS environments using CloudFormation
- Using CloudWatch for monitoring
- Using AWS solutions for backups and archiving
- Planning for production go-live activities

Managing infrastructure, deployments, and support at scale

In recent times, there has been a huge shift in the way organizations manage their cloud environments and applications. This is in response to the ease of operating in the cloud, availability of infrastructure on-demand, and cloud-based PaaS services that can readily be leveraged within your applications. The overall speed and number of deployments have increased greatly, thereby requiring significant levels of automation in application builds, infrastructure provisioning, and deployments. Software development and release is evolving into continuous delivery environments (enabled by features and services provided by cloud vendors).

In such environments, it is important that tasks and processes be highly repeatable, resilient, flexible, and robust. Amazon provides numerous tools, APIs, and services to enable you to create highly-automated DevOps pipelines. These pipelines can help you handle your infrastructure requirements, including provisioning your technology stacks, performing deployments dynamically with zero downtime, and supporting your end-customers at scale. Some of the major AWS services in these areas are AWS CloudFormation, AWS CloudTrail, and AWS CloudWatch. These are described in more detail in the subsequent sections.

Besides AWS tools and services, it is also important that we upgrade your skills and try to stay as current as possible in terms of the new services and features released by Amazon. This is important because the roles of application developers and infrastructure engineers are also evolving rapidly. Increasingly, application developers are taking on end-to-end responsibilities for their specific applications. These responsibilities include tasks that were typically handled by specialized operations and infrastructure teams earlier. At the same time, infrastructure engineers, specialists, and administrators are focusing more on network architecture, infrastructural policies, templates, generic patterns, frameworks and models, AWS service usage guidelines and principles, cloud security, and so on.

We strongly recommend that you actively engage with Amazon architects throughout the development lifecycle. They have done this before and they can help you get it right the first time. In addition, ensure you document everything including your design, code, scripts, infrastructure, templates, policies, processes and procedures, and so on. This will help with your team's technical understanding of the cloud environment, aid the rapid on-boarding of new team members, and help establish standards and guidelines in your organization.

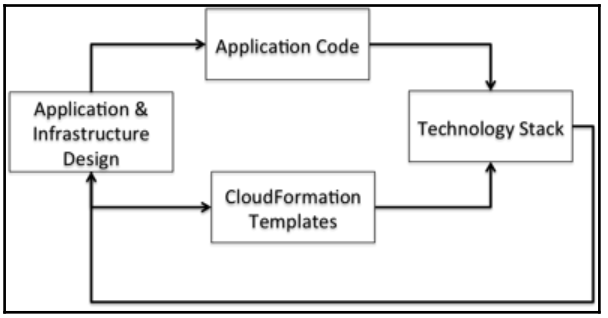
Creating and managing AWS environments using CloudFormation

Your primary goal for deployments includes minimizing the overall time and effort required for it, while having predictability, flexibility, and visibility for each of the steps required to install and run your cloud applications. AWS CloudFormation provides an easy way to create and manage the AWS resources for your application. In addition, CloudFormation allows you to do all this in a declarative and parametrized manner while managing all of the dependencies for you.

It is important that you use CloudFormation right from the beginning even if your initial configuration is simple enough to be provisioned and managed using the console. Also ensure that all subsequent changes to your stacks flow through CloudFormation as well, in order to avoid unpredictable results. If you need to make a change from outside of CloudFormation, then you should have a process in place to make the appropriate changes in the CloudFormation template before any subsequent stack updates. Ensure you protect your stacks from accidental or inadvertent changes by strictly managing changes or updates to your templates using tag-based IAM policies. You can also create stack policies that can prevent changes to certain resources, for example, disallowing any changes to the database while changes are being made to other resources in the stack. Use comments to describe the resources and other elements in your templates. Following these practices will minimize the chance of errors during the provisioning and updating of your environments.

A typical high-level workflow using CloudFormation is shown in the following figure. The business requirements drive your application’s design and infrastructural requirements. Subsequently, these designs and infrastructural requirements are realized in your application's code and templates. CloudFormation templates are ideal for provisioning and replication of your application's technology stack.

The feedback loop helps you address your evolving business requirements, and also improves your designs and processes over time. As costs form an important input that I sent to the feedback loop, you can use the AWS Cost Explorer to obtain the costs associated with your stack (by assigning appropriate tags to your resources):



The technology stack largely consists of hardware, the OS, libraries, and your application packages and/or code. You can define one or more stacks based on your application layers and environments, that is, dev, test, staging, and production. In addition, you can also define nested stacks to address various layers or components in your architecture. CloudFormation templates that reference other templates result in nested stacks or a tree of stacks. This is typically done to drive as much reusability as possible to your deployment processes. For example, if you have several websites sharing common requirements in terms of their load balancing and autoscaling features, then you can create a template for your ELB and autoscaling groups, and reuse it across multiple stacks.

Multiple stacks are typically required not only to organize your implementation according to layers or environments but also because these layers and environments have different characteristics. These characteristics may include different life cycles associated with your AWS resources or different ownership associated with the layers in your architecture. In addition, if you have services and/or databases that are shared by multiple applications, then having a separate stack for them will help you manage these resources better.

Creating CloudFormation templates

AWS CloudFormation provides sample templates that you can use as a starting point for defining your specific requirements. You will need to create one or more templates to translate your design into stacks. For example, if you have designed a services-oriented application then your application contains units of functionality and contracts that define its interfaces. You might also have dependencies between your services. Hence, your stacks will need to reflect these services' characteristics in terms of parameters, output, and so on.

Creating CloudFormation templates is very similar to software development practices. For example, you will need to develop and conduct code reviews, maintain repositories and version control, and test, run, and maintain them. In addition, when you hit errors you will need to debug and fix your code.

In order to minimize errors and the time taken to develop production quality CloudFormation templates, ensure that you:

- Validate the template (check for structure and API usage, JSON syntax, the presence of circular dependencies, and so on).
- Use parameter types to avoid bad input parameters and specify appropriate parameter-related constraints and regex patterns. These parameters are validated at the beginning of the stack creation process, so if there are any errors you will know almost immediately.
- Grant IAM permissions for creating the full stack and all the resources specified in the template. In addition, ensure that permissions are given to create/update the stack as well as rollback the changes.
- Ensure sufficient quotas for all the resource types in your stack, for example, the number of EC2 instances, RDS storage limits, and so on.

It is good practice to leverage `CloudFormation::Init` to declaratively specify the packages to be installed and users to be created, as well as executing configuration scripts and so on. `CloudFormation::Init` makes your application updatable; for example, you can update the running stack with a new version of your application. If you prefer to use tools such as Chef, then you can also use Chef recipes to install and/or update your application in the stack.

As a security best practice, never include secret keys and access keys in your CloudFormation templates. You can leverage IAM roles to achieve the same result. Even if you need to include parameters such as database passwords, you should mark them with a no echo option. That will ensure that the parameter is not revealed in the logs or stack events.

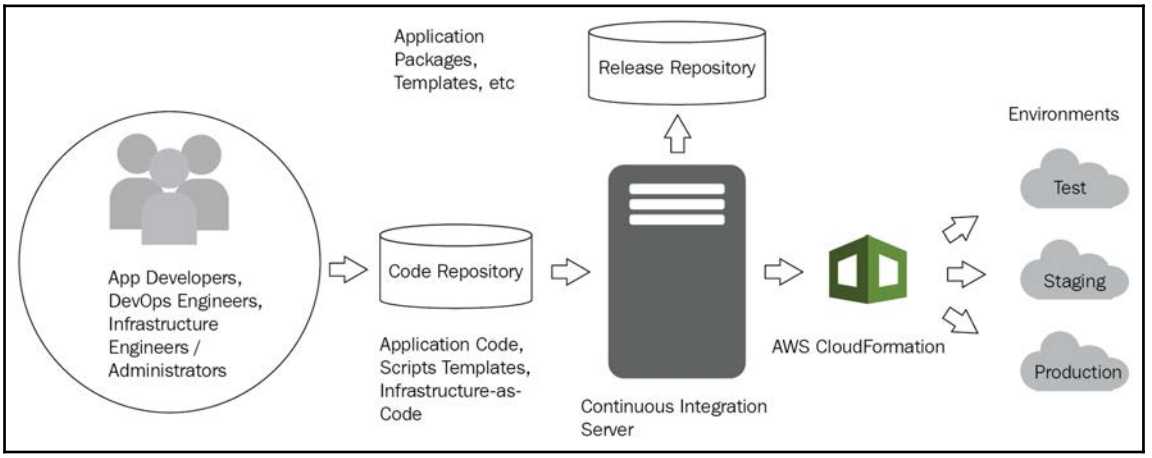
Leverage CloudFormation's integration with other AWS services and features to get a better handle on managing your stack. For example, use CloudFormation's integration with CloudTrail to log CloudFormation API calls. Furthermore, you can query these logs and set alerts. These features can enable you to troubleshoot or debug any issues.

There are other AWS tools that can help you with creating and updating your CloudFormation templates. For example, you can use AWS Config for detecting changes in the stack made from outside of CloudFormation. You can also use CloudFormer to create CloudFormation templates from existing resources in an active stack.

Building a DevOps pipeline with CloudFormation

Application deployments in traditional environments used to take days, and if the deployment required procurement of infrastructure then the deployment cycle would extend to weeks and sometimes even months. With cloud applications, the infrastructure is available on-demand and deployment time is reduced to minutes. In large enterprises, the average number of deployments across their application portfolio now runs into a couple of hundred per day. In order to achieve smooth and error-free deployments with zero downtime, it is imperative to plan, design, and implement a highly-automated DevOps pipeline.

The following figure illustrates a DevOps pipeline incorporating code repositories, a continuous integration environment, AWS CloudFormation, and application environments:



CloudFormation is a key part of your DevOps pipeline, enabling faster production releases. For example, as shown in the figure, you can set up a continuous integration environment that builds your application, packages the application code and CloudFormation templates, and then uses CloudFormation templates to create the stack and deploy your application (in various environments including the final promotion to production).

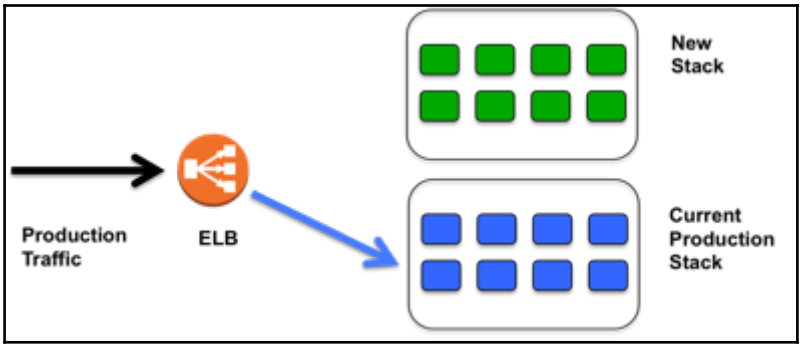
Updating stacks

There are primarily two main approaches to updating your stack—the in-place and Blue-Green approach. Each of these approaches has their own pros and cons, and you should select the approach that is most suitable for your specific situation. You can also start with one approach and then move to a different approach depending on your business needs.

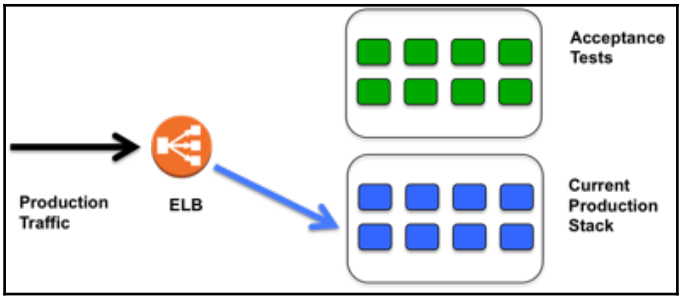
As the name suggests, the in-place updates approach requires you to create a new template and then use that template to update your existing stack by using the update stack API. This approach is faster, more cost-efficient, and the migration of data and application states is much simpler than with the Blue-Green approach.

In the Blue-Green approach, you take the new template and create a completely new and separate stack (from your currently running stack). After you have verified that the new stack is running as per your requirements, you switch production traffic over to it. The main steps in a Blue-Green deployment are illustrated in a series of figures here.

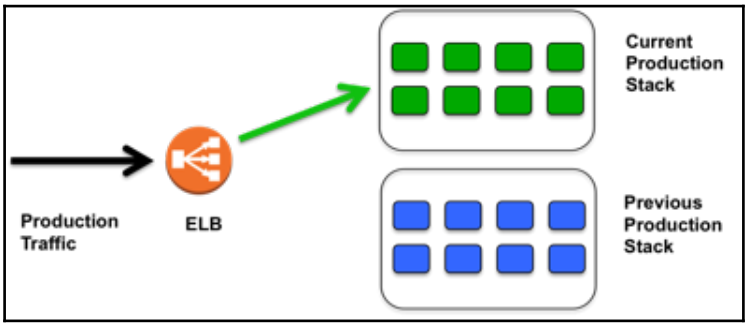
Instances within the production stack are labeled Blue and the instances hosting the new stack are labeled Green. The Blue fleet is currently serving all of the production traffic:



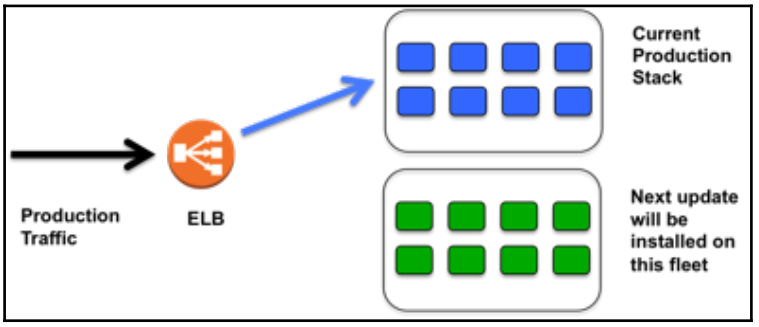
Verification and acceptance tests for the new stack are conducted on the Green fleet while the Blue fleet continues to serve the production traffic:



After the acceptance tests are successfully cleared, the production traffic is switched over to the Green fleet:



The Green fleet is then labeled Blue and is serving all of the production traffic. The fleet that was originally Blue is now labeled Green:



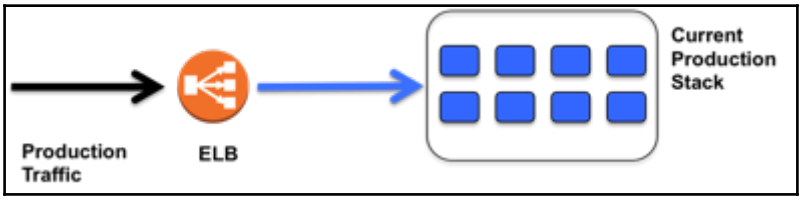
The primary advantage of the Blue-Green approach is that you are not touching the currently running stack at all. You also have the option to fallback to the old stack at any time, easily. However, Blue-Green deployments are expensive, as this approach requires you to spin up a duplicate set of instances. The success of this approach is also highly dependent on the thoroughness of your acceptance tests. In addition, the DNS switch can create issues as DNS clients are not well behaved and require ELB warm-up.

Each of the stack update options have certain desirable characteristics, and you can combine them appropriately to create an approach that works best for you.

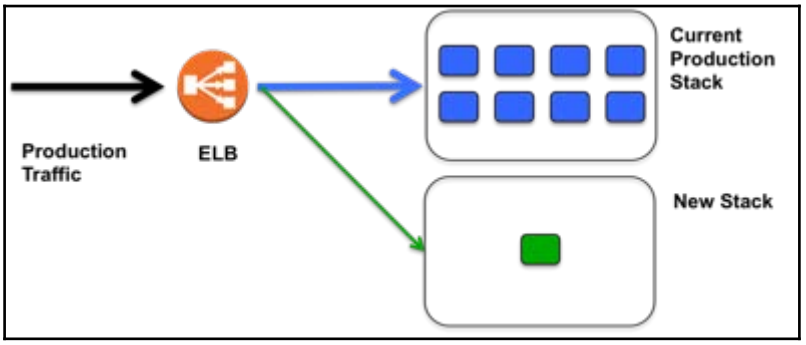
There are several variants of the Blue-Green approach that address some of the shortcomings of the traditional Blue-Green deployment approach discussed previously.

An approach that uses a mix of the Green and Blue instances (with a single ELB) may be useful for certain applications, and is described here.

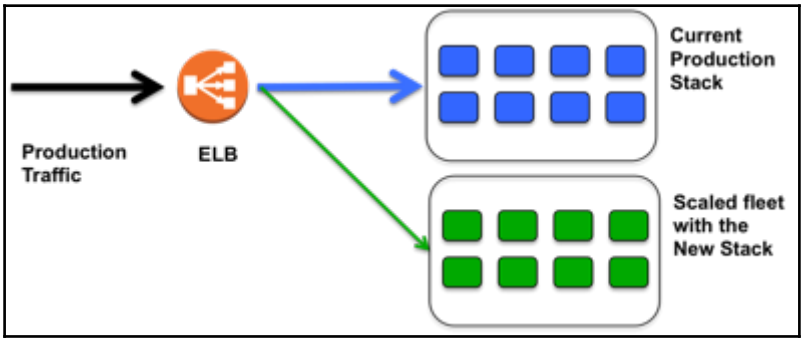
The starting state is a single fleet of instances labeled Blue (belonging to an Auto Scaling group). These instances are serving all of the production traffic:



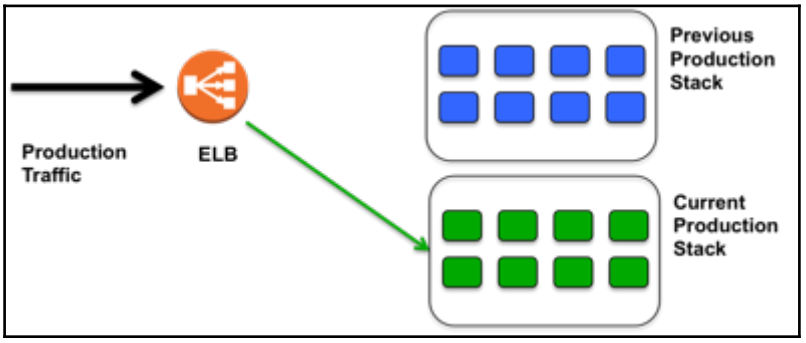
In this stage, you create a new instance hosting the new stack (in a separate Auto Scaling group with a maximum size of 1). While the Blue fleet is serving a majority of production traffic, the Green instance also starts serving some of the production traffic:



In the next step, you scale up the Green fleet:



Finally, you switch over to the Green fleet:



At this stage, you can shut down the Blue fleet and your Green fleet is now the designated Blue fleet. This approach takes into consideration that acceptance tests may not be complete or comprehensive. At the same time, there are no DNS changes or ELB warm-ups required.

A variant of the Blue-Green approach that works well within an Auto Scaling group is rolling updates. The updates are applied to the instances in batches (with zero downtime). CloudFormation ensures that there is always a set of healthy instances serving your customers at all times. In this approach, the Auto Scaling group is divided into several batches, and then the update is applied to the first batch of instances. ELB health checks should be enabled to ensure that the instances are healthy after the update has been applied. If the instances in the batch are healthy, then you can signal back to CloudFormation to update the next batch of instances. These rolling updates, across all your instances, can be achieved using a single CloudFormation template.

Extending CloudFormation

Typically, extensions to CloudFormation are required if your application uses third-party services and you want to include the provisioning of the third-party resources in your CloudFormation template. Extensions to CloudFormation may also be required for AWS services that are not currently supported by CloudFormation, or if you have a requirement to provision on-premises resources as a part of your stack. Two ways of including such resources in your CloudFormation stack are discussed here.

In the first approach, for achieving a tighter integration of such services or custom resources, in our stack the third-party service provider will need to expose a service that can process incoming provisioning-related create, update, and delete requests. CloudFormation will send a message to the third-party service and wait for a response. Upon a success response, CloudFormation will continue with its stack creation process, otherwise it will fail out. This way, CloudFormation can treat the entire stack including the external resources as a single unit that either succeeds fully or fails out in its entirety.

The second approach leverages stack events to achieve the same results. For example, if you want your web application to provision a subscription to a third-party service, then while CloudFormation provisions your web application it produces certain stack events. CloudFormation delivers these events to an SNS topic that is subsequently picked up by a provisioning application (that you have to write) in order to subscribe to the third-party resource. This approach is not as robust as the previous one because CloudFormation is not aware of failure during the provisioning of the third-party service.

Using CloudWatch for monitoring

Amazon CloudWatch enables the monitoring of Amazon services, standard and custom defined metrics, and a variety of logs. Typically, you would want to retrieve metrics for analysis and/or integration with other monitoring tools. For example, you can use AWS Trusted Advisor to analyze your AWS configuration and usage, compare it to the best practices, and to alert you to opportunities to save on costs, help close security gaps, or improve system reliability and performance. CloudWatch provides APIs for retrieving hundreds of metrics by namespace, start and finish times, intervals, and so on.

CloudWatch logs can be monitored for errors, exceptions, HTTP response codes, Amazon S3 logs, and so on. In addition, you can also use the logs to correlate the system status with change events such as when AWS CloudFormation is used to rollout a new stack. We can define metric filters on the logs and raise alerts based on specific thresholds. These alerts can in turn be forwarded to SNS topics for appropriate notifications to be pushed out. The metric filters can be based on literal terms, common log formats, or specified using JSON. In addition, you can combine multiple literal terms, group the terms, count occurrences, and/or specify variable names for log record fields, and so on.

For monitoring API calls to AWS services, you can integrate AWS CloudTrail logs with AWS CloudWatch. You can also choose to receive SNS notifications from CloudWatch for the API activity captured by CloudTrail. Typically, you will turn on this integration from the CloudTrail console or through a CloudFormation template, define a metric filter for your CloudWatch Logs log group, assign a CloudWatch metric to the metric filter, and then create an appropriate CloudWatch alarm.

Other alternatives include subscribing to third-party logging services or rolling out your own solution for centralized monitoring. For example, you can use AWS Kinesis to ingest logging messages, an Elasticsearch cluster for searching through the records efficiently, and a product such as Kibana for visualization support. There are several third-party logging service providers such as Loggly, Splunk, Sumo Logic, and so on. You can subscribe to their services to meet your requirements (at scale).

Using AWS solutions for backup and archiving

Using AWS for backups and archiving is very common and an easy entry point for organizations new to the cloud. The main reasons for the popularity cloud-based backup solutions are AWS' global infrastructure, data durability SLAs, a rich ecosystem of partners and vendors, and compliance with regulations such as HIPAA, PCIDSS, and so on. Taking a phased approach that begins with using cloud storage as a backup data store is a reasonable and common approach. However, taking this further to create your application environment in the cloud can be a good business continuity strategy. It is also easier to track the actual usage of the backup data on the cloud, thereby presenting further opportunities to reduce your overall costs.

In most cases, the enterprise already has a well-established backup strategy, policy, and technology solution in place. They are not looking for a complete replacement. The primary motivation here is to leverage the cloud as a lower cost destination. However, for most early stage start-ups, the cloud may represent their first and only backup solution.

There are third-party solutions such as Commvault that are natively integrated with Amazon S3 and Glacier. Other backup solutions can also be integrated using storage gateways. These approaches can help evolve your backup solution to embrace cloud storage with the least disruption while you continue to use your existing processes. Cloud storage represents "unlimited" capacity so you do not have to worry about closely tracking your tape usage or rotating through them constantly.

It is important to carry out a data classification exercise for all the data in your application. For example, you might classify your data as long-term data to be kept for compliance reasons and unlikely to be restored, very high volume data to be transferred from on-premises storage to the cloud, data shared by multiple applications (document shares, image repositories, and so on), highly available data, data related to online collaboration applications, and so on. This classification can help you choose appropriate solutions for storage and backup. AWS provides different storage classes, that is, S3, Glacier, and EBS, to meet varied data life cycle management requirements. All these data storage services are scalable, secure, and reasonably priced.

It is also important to define certain guiding principles for your backups. For example, you could choose to back up only the data and not the entire VM. This would mean you are choosing to rebuild (using a service such as AWS CloudFormation) instead of restore. Other guidelines may recommend building stateless services and storing all data on Amazon S3 and leveraging services such as SQS, assuming that all instances are temporary or fail sooner or later. You might also want to take snapshots of your EBS volume on other EBS volumes to recover faster from instance failures. In the DevOps environment, owners of applications have increasing responsibilities in terms of their data.

While evaluating the cost of a cloud-based backup strategy, ensure you do not restrict your TCO calculations to hardware and software alone. For a good comparison, ensure you take into consideration costs associated with facilities, maintenance, people and professional services, storage utilization, transportation, cost of capital, and so on. In larger backup environments, the cost of a cloud-based solution compares very favorably versus the total cost of physical media, robot systems, and other costs mentioned earlier.

Planning for production go-live activities

In this section, we will cover the final steps required for a new application to go live on the cloud. By this time, your application should have been fully tested (against functional and non-functional requirements) and accepted by the business. All your templates for automated production infrastructure provisioning and deployment scripts should be tested and ready to go, and backup policies and disaster recovery strategies documented and tested.

It is very useful to create a comprehensive checklist for executing the actual go-live process. This will ensure that you are systematically executing each step in the process, verifying intermediate results (for example, complete and correct data migration), communicating to the stakeholders at regular intervals, making go/no-go decisions at the appropriate times, having a rollback strategy clearly defined, and so on.

As a good practice, after you have deployed your application in the production environment, run a set of predefined tests to ensure your application is functioning as required. You should also test that application monitoring and logging is functioning as expected. Finally, ensure that you engage internal and/or external specialists to conduct a penetration test. These tests could result in changes to the application as well as some infrastructure settings, therefore plan sufficient time in your schedule for a couple iterations of the penetration test. After you have cleared the penetration test, you should be officially live and actively serving your customers.

At this time, you may also want to schedule a team meeting to analyze what worked well versus what could have been done better during the project. It is also useful to document lessons learnt and best practices for your specific situation, and plan your next release of the application with new features, bug fixes, and tweaks to your infrastructure.

In the next section, we will walk you through the deployment-related activities for our sample application.

Setting up for production

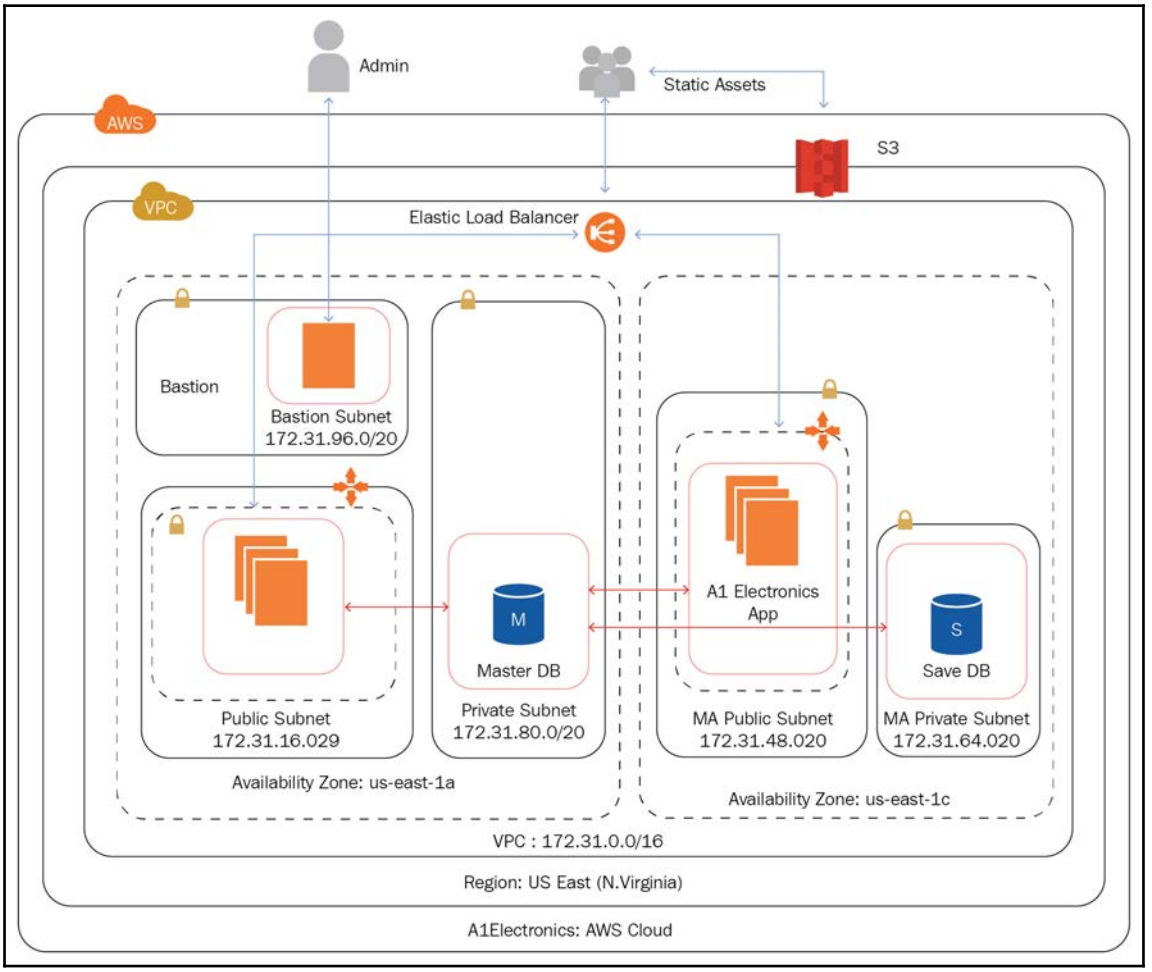
This is it! The final section in which AWS will be configured to host the application for production deployment. The key issues in production setup are the health monitoring of the application, disaster recovery (both for data and infrastructure), and a secure production environment (with reasonable ongoing costs).

AWS production deployment architecture

For the first step, we need to design the deployment architecture. You can architect AWS in several different ways to meet your business requirements. The deployment architecture presented here takes into consideration security practices, and is specifically designed for scalability and high availability. In addition, it is an extension to the one presented for HA in `Chapter 5`, *Designing for and Implementing High Availability*.

Let's re-examine the choices we made for the selection and configuration of AWS resources. The choices for regions, availability zones, ELB, ASG, RDS, and S3 have already been covered in chapters 3 to 5. All the AWS resources needed for production setup have been discussed previously.

The following diagram represents the deployment architecture for our sample application:



VPC subnets

The first step is to logically partition the VPC into separate subnets based on our requirements. Next we apply security groups (firewalls) to each of the subnets to accept connections on fixed TCP ports (from predefined subnets). The main purpose of having separate subnets is to secure the hosts by restricting access to them. For example, we host the RDS MYSQL database server in a private subnet that accepts connections on port 3306 only. This access is restricted to be from two public subnets. The VPC and the subnets created within the VPC are listed here:

- Subnet at `172.31.112.0/20` that hosts the bastion host and accepts SSH connection from trusted sources only.
- Public subnets hosting the EC2 instances in the auto-scaling group for the application. It accepts HTTP and HTTPS connections from any source.
- Private subnets at `172.31.80.0/20` and `172.31.96.0/20` to host the database servers and accept MYSQL connections only from defined public and bastion security groups.

The default VPC subnets configured in `Chapter 5`, *Designing for and Implementing High Availability*, were `172.31.16.0/20` in AZ **us-east-1a** and `172.31.48.0/20` in AZ **us-east-1c**. We will continue to use them as our public subnets.

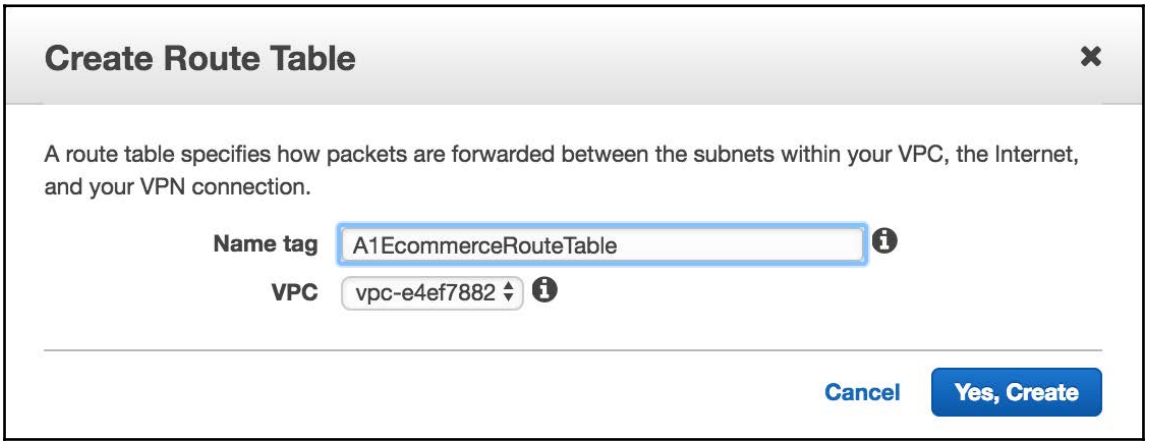
Private subnet

Any EC2 running on a private subnet can be accessed from another EC2 instance from within VPC network or over a VPN network. The instances running are not accessible via the public internet. Each VPC has a default internet gateway associated with it. A new subnet is always created as a public subnet. The public subnet can be changed to a private subnet by assigning its route table to a private route table:

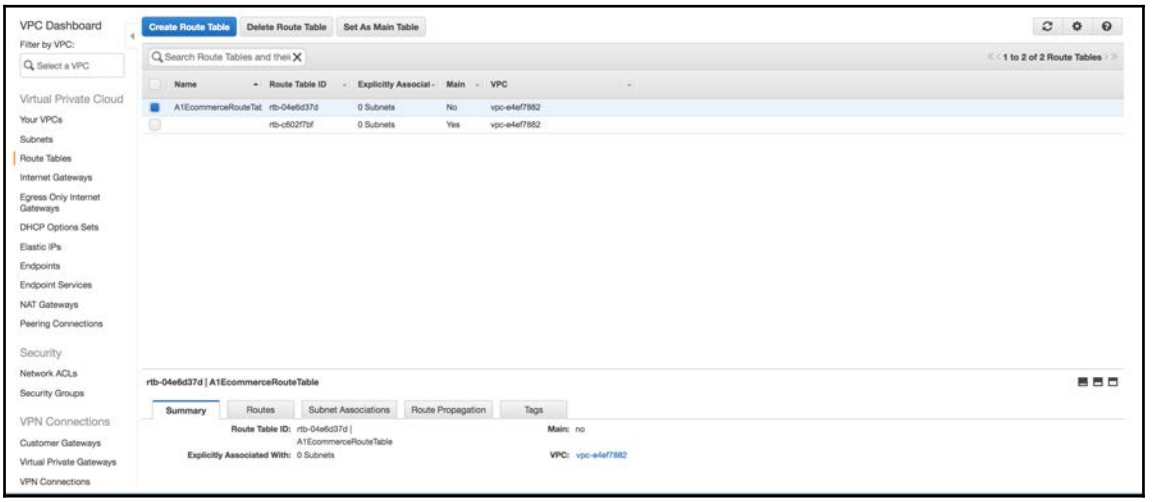
1. The first step is to create a private route table:
    1. From the VPC dashboard, navigate to **Route Tables** and click on the **Create Route Table** button:



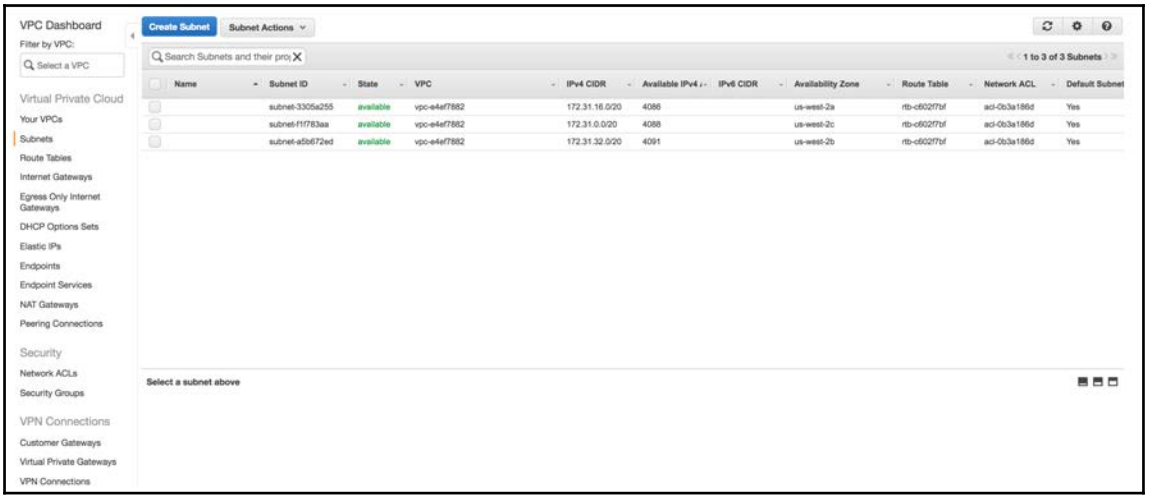
    2. In the **Create Route Table** popup, assign the name of the route table in the **Name** tag. Click on the **Yes, Create** button:



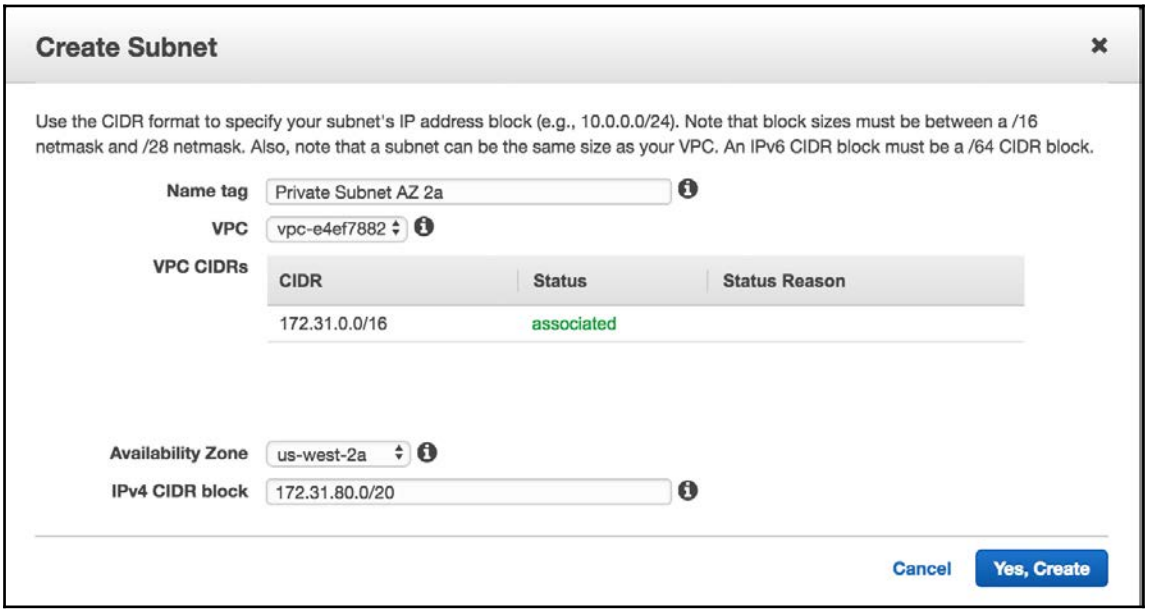
3. You should see the following screen:



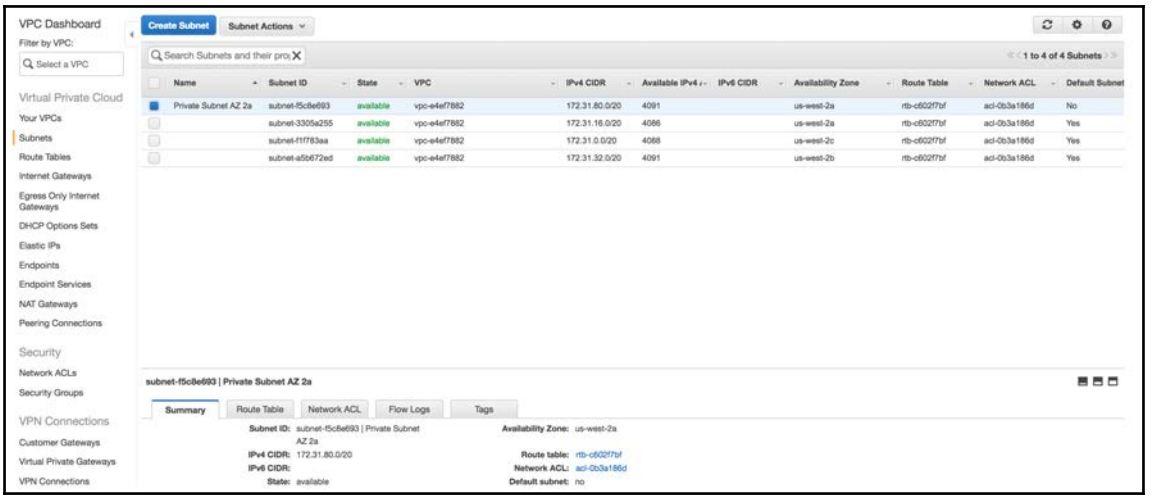
4. The next step is to create a subnet:
    1. From the VPC dashboard, navigate to **Subnets** and then click on **Create Subnets**:



2. **Name** tag: Specify a name for the subnet. This name will be reflected in the VPC dashboard.
3. **VPC**: Choose the VPC in which this subnet will be created. Select the option containing `172.31.0.0/16` from the dropdown if you have more than one VPC.
4. **Availability Zone**: The availability zone in which this subnet will be created. From the dropdown, select **us-west-2a**; this is one of the two private subnets. The other one will created in the **us-west-2c** Availability Zone as per the deployment architecture.
5. **CIDR block**: **Classless Inter-Domain Routing** (**CIDR**) defines a range of IP addresses to be allocated to the hosts in the subnet. In this case, `172.31.80.0/20` defines the IP address range from `172.31.80.0` to `172.31.95.255` (a total of 4,096 hosts):



6. You should see the following screen:



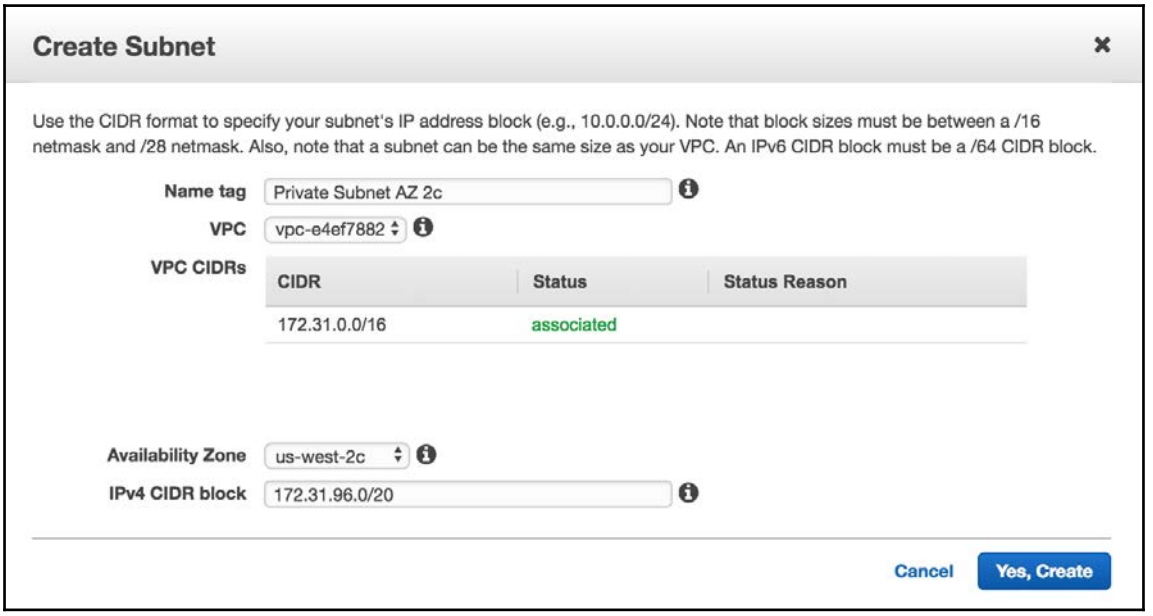
7. The last step is to associate the private route table created in step 1 with the subnet created in step 2:
    1. From the VPC dashboard, navigate to **Subnets** and click on the subnet created in step 2.
    2. Navigate to **Route Table** tab in the bottom pane and click on **Edit**:



3. From the **Change To** dropdown, select the route created in step 1:

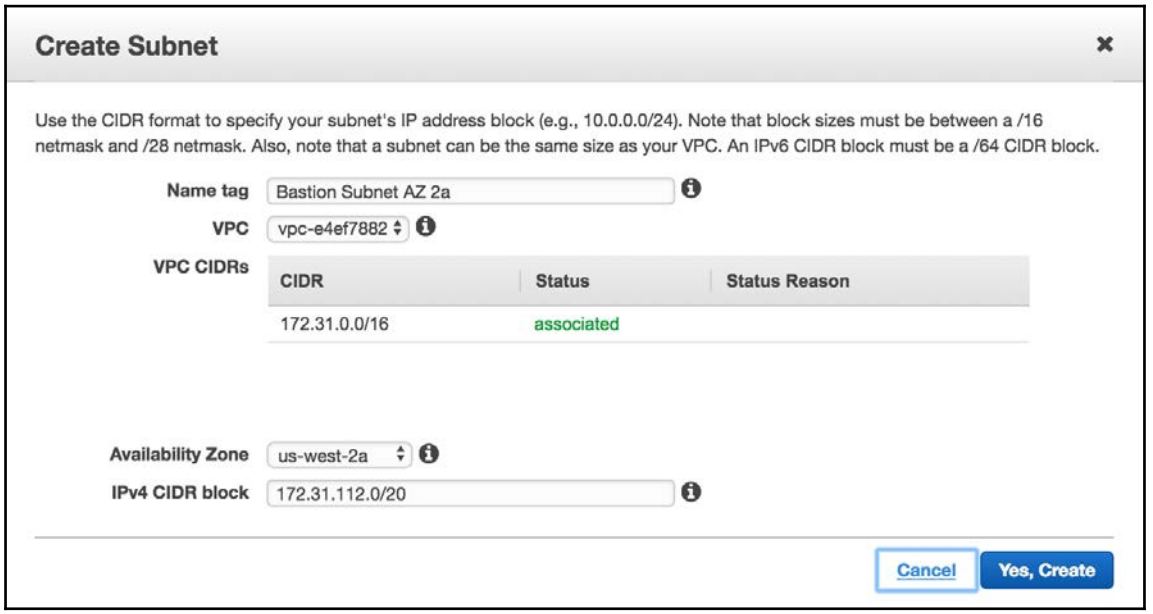


Similarly create another subnet, Private Subnet, with CIDR block `172.31.96.0/20` in the Availability Zone **us-west-2c**; assign the private route table to it (created in step 1):



Bastion subnet

Create another subnet named bastion, with CIDR block `172.31.112.0/20` in Availability Zone **us-east-1a**:



There is no need to assign a private route table to it as the EC2 instances running in this subnet will be accessed by clients from the public internet.

Bastion host

A **bastion host** is a secure host that accepts SSH connections only from trusted sources. A trusted source is the static IP address of your internet connection. This ensures that the access to your AWS resource is from a machine from within your network. A bastion is used to administer your AWS network and instances. All instances accept SSH connections only from the bastion security group.

Security groups

The traffic between the instances is governed by the ingress (inbound) and egress (outbound) rules defined in the security groups. Listed here are recommended security groups and their inbound and outbound rules. Please refer to Chapter 2, *How Are Cloud Applications Different?*, for how to create security groups.

ELB Security Group Recommended Rules: Apply this security group to the ELB.

- Inbound:

Source (CIDR)	Protocol	Port Range	Comments
0.0.0.0/0	TCP	8080	Accept HTTP traffic from anywhere.
0.0.0.0/0	TCP	8443	Accept HTTPS traffic from anywhere.

- Outbound:

Destination (CIDR)	Protocol	Port Range	Comments
ID of Web security group	TCP	8080	Route HTTP traffic to instances that have a web security group assigned.
ID of Web security group	TCP	8443	Route HTTPS traffic to instances that have a web security group assigned.

Recommended Rules for the Web Security Group: Apply this security group to EC2 instances running on public networks in both the Availability Zones. This security group is for the web servers.

- Inbound:

Source (CIDR)	Protocol	Port Range	Comments
ID of ELB security group	TCP	8080	Accept HTTP traffic from the load balancer.
ID of ELB security group	TCP	8443	Accept HTTPS traffic from the load balancer.
ID of Bastion security group	TCP	22	Allow SSH traffic from the bastion network.

- Outbound:

Destination (CIDR)	Protocol	Port Range	Comments
ID of Database security group	TCP	3306	Allow MYSQL access to the database servers assigned to the database security group.

Recommended Rules for the Bastion Security Group: Apply this security group to EC2 instances running on the bastion network.

- Inbound:

Source (CIDR)	Protocol	Port Range	Comments
MyIP	TCP	22	Accept SSH connection for your fixed static IP. This implies you can connect to the bastion sever from only the IP address. If you do not have a static IP, change the source to 0.0.0.0/0.

- Outbound:

Destination (CIDR)	Protocol	Port Range	Comments
ID of Database security group	TCP	3306	Allow MYSQL access to the database servers to administer the MYSQL database.
ID of Web security group	TCP	22	Allow SSH access to the web server instances for administration running on the public network.

Recommended Rules for the Database Security Group: Apply this security group to EC2 instances running on the private network in both the availability zones. This security group is for the database servers:

- Inbound:

Source (CIDR)	Protocol	Port Range	Comments
ID of Web security group	TCP	3306	Accept MYSQL connections from the web application running on the public network.

ID of Bastion security group	TCP	3306	Allow MYSQL access to the database servers to administer the MYSQL database.

- Outbound:

Destination (CIDR)	**Protocol**	**Port Range**	**Comments**
None	None	None	Delete all outbound rules.

Infrastructure as Code

So far we have been setting up the AWS infrastructure via the Amazon AWS console, which is quite helpful in the initial stages when you are learning the ropes. However, it is good practice to build your cloud infrastructure via code.

AWS provide two services, CloudFormation and AWS OpWorks. CloudFormation focuses on providing foundational capabilities for the full breadth of AWS services, while AWS OpWorks focuses on deployment, monitoring, auto-scaling, and automation, and supports a narrower range of application-oriented AWS resource types including EC2 instances, EBS volumes, Elastic IPs, and CloudWatch metrics.

Setting up CloudFormation

While working with CloudFormation, you define the AWS resources you need and then wire them together as per your architecture's requirements:

```
{
    "AWSTemplateFormatVersion" : "",
    "Description" : "",
    "Parameters" : {
    },
    "Mappings" : {
    },
    "Conditions" : {
    },
    "Resources" : {
    },
    "Outputs" : {
    }
}
```

The template JSON file includes the following sections; only the `Resources` section is mandatory while the rest are all optional:

- `AWSTemplateFormatVersion`: Defines the capabilities of the template. Only one version has been defined so far.
- `Description`: Free text; use it to include comments for your template. It should always follow `AWSTemplateFormatVersion`. The max size of this free text is 1,024 characters.
- `Parameters`: It is used to pass values to the template while creating the stack, and helps customize the template each time you create a stack.
- `Mappings`: As the name suggests, it is map of key-values pairs. For example, this can be used to select the correct AMI base image for a region where the stack is being created; the key will be the region and the value will be the AMI ID of the base image.
- `Conditions`: Since JSON cannot have any logic embedded in it, a section was created to implement the basic conditional logic evaluation within the JSON template. The conditional logic functions available are `AND`, `OR`, `EQUALS`, `IF` and `NOT`.
- `Resources`: This is the section in which you define your AWS resources and wire them together. This section is mandatory.
- `Outputs`: In this section, you declare the values to be returned upon creation of each AWS resource. This is useful, for example, to find the ELB URL or a RDS endpoint.

Since a full CloudFormation template for our production deployment architecture can run into hundreds of lines of JSON code, we will only present the creation of key AWS resources; however, the complete script, `a1ecommerceaws.json`, is available for download in the source code repository.

- `VPC`: Here, a different CIDR block is used for production instead of the default `172.31.0.0/16`. You can skip this if you want to keep the default CIDR block, but remember to change the reference to the VPC in the subnets:

```
"VPC":{
   "Type":"AWS::EC2::VPC",
   "Properties":{"CidrBlock":"10.44.0.0/16",
       "EnableDnsSupport" : "true",
           "EnableDnsHostnames" : "true",
           "InstanceTenancy" :{"Ref":"EC2Tenancy"},
       "Tags":[{"Key":"Application",
               "Value":{"Ref":"AWS::StackName" }
```

```
            },
            {"Key":"Network","Value":"Public" },
            {"Key":"Name","A1Ecommerce Production"}
          ]
0   }
},
```

- `Subnets`: As an example only one sample is presented here for the public subnet. Remember to replace `CidrBlock` with `172.31.16.0/24` and VPC ID with the default VPC identifier, for example `vpc-e4ef7882`. The `DependsOn` is not required if you wish to create the subnets in the default VPC. Similarly, create the other subnets:

```
"PublicSubnet":{
"DependsOn":["VPC"],
   "Type":"AWS::EC2::Subnet",
   "Properties":{
       "VpcId":{"Ref":"VPC"},
       "CidrBlock":"10.44.0.0/24",
       "AvailabilityZone":"us-west-2a",
       "Tags":[
           {"Key":"Application",
             "Value":{"Ref":"AWS::StackName"}
           },
           {"Key":"Network","Value":"Public"},
           {"Key":"Name","Value":"Public Subnet"}
       ]
   }
}
```

- `Security Group`: As an example, the ELB security group is specified here. Similarly, create the other security groups:

```
"ELBSecurityGroup":{
   "DependsOn":["VPC"],
   "Type":"AWS::EC2::SecurityGroup",
   "Properties":{
       "GroupDescription":"ELB Base Security Group",
       "VpcId":{"Ref":"VPC"},
       "SecurityGroupIngress":[
           {"IpProtocol":"tcp","FromPort":"80",
             "ToPort":"80","CidrIp":"0.0.0.0/0"},
           {"IpProtocol":"tcp","FromPort":"443",
             "ToPort":"443","CidrIp":"0.0.0.0/0"},
       ],
       "Tags":[{
```

```
                "Key":"Name",
                "Value":"ELB Security Group"}
        ]
    }
},
"ELBSecurityGroupEgress80":{
    "DependsOn":["ELBSecurityGroup"],
    "Type":"AWS::EC2::SecurityGroupEgress",
    "Properties":{
        "GroupId":{"Ref":"ELBSecurityGroup"},
        "IpProtocol":"tcp","FromPort":"8080",
        "ToPort":"8080",
        "DestinationSecurityGroupId":{
            "Fn::GetAtt": [ "WebSecurityGroupPublic","GroupId" ]
        }
    }
},
"ELBSecurityGroupEgress443":{
    "DependsOn":["ELBSecurityGroup"],
    "Type":"AWS::EC2::SecurityGroupEgress",
    "Properties":{"GroupId":{"Ref":"ELBSecurityGroup"},
        "IpProtocol":"tcp","FromPort":"8443",
        "ToPort":"8443",
        "DestinationSecurityGroupId":{
            "Fn::GetAtt": [ "WebSecurityGroupPublic","GroupId" ]
        }
    }
 }
```

- `RDS`: An RDS subnet needs to be created so that the RDS service can run in the correct availability zones, `us-west-2a` and `us-west-2c`:

```
"RDSSubnetGroup":{
"Type" : "AWS::RDS::DBSubnetGroup",
"Properties":{
   "DBSubnetGroupDescription":"Availability Zones for CanvasDB ",
   "SubnetIds" : [ { "Ref" : "PrivateSubnet" },
                   { "Ref" : "DbPrivateSubnet" }
                 ]
"A1EcommerceMasterDB" : {
   "Type" : "AWS::RDS::DBInstance",
     "Properties" : {
       "DBName" :"a1ecommerce",
       "DBInstanceIdentifier" : "a1ecommerce",
       "AllocatedStorage" : "5",
       "DBInstanceClass" : "db.t1.micro",
       "BackupRetentionPeriod" : "7",
```

```
        "Engine" : "MySQL",
        "MasterUsername" : "a1dbroot",
        "MasterUserPassword" : "a1dbroot",
        "MultiAZ" : "rue",
        "Tags" : [{ "Key" : "Name", "Value" : "A1Ecommerce Master
Database" }],
        "DBSubnetGroupName":{"Ref":"RDSSubnetGroup"},
        "VPCSecurityGroups": [ { "Fn::GetAtt": [
"PrivateSecurityGroup", "GroupId" ] } ],
      },
      "DeletionPolicy" : "Snapshot"
}
```

- `ELB`: The ELB is straightforward. It routes the incoming traffic to the two subnets in the two availability zones and is assigned the ELB security group:

```
"ElasticLoadBalancer":{
   "Type":"AWS::ElasticLoadBalancing::LoadBalancer",
   "DependsOn":["PublicSubnet","HASubnet"
   ],
   "Properties":{
     "Subnets":[{"Ref":"PublicSubnet"},
       {"Ref":"HASubnet"}
     ],
     "CrossZone":"true",
     "Listeners":[{
         "LoadBalancerPort":"8080",
         "InstancePort":"8080",
         "Protocol":"HTTP"
       },
        {
         "LoadBalancerPort":"8443",
         "InstancePort":"8443",
         "Protocol":"TCP"
       }
     ],
     "ConnectionDrainingPolicy":{
       "Enabled":"true","Timeout":"60"
     },
     "SecurityGroups":[{
         "Ref":"ELBSecurityGroup"
       }
     ],
     "HealthCheck":{
       "Target":"HTTP:8080/index.html",
       "HealthyThreshold":"3",
       "UnhealthyThreshold":"5",
```

```
        "Interval":"30",
        "Timeout":"5"
      }
    }
}
```

- `Launch Configuration`: The `ImageId` is your base AMI instance that the Auto Scaling group will launch. Replace the image ID with your own AMI:

```
"LaunchConfig":{
"Type":"AWS::AutoScaling::LaunchConfiguration",
"Properties":{
   "KeyName":{
     "Ref":"KeyPairName"
   },
   "ImageId":"i-3a58b4cd"
   "SecurityGroups" : [ { "Ref" : "WebSecurityGroupPublic" } ],
   "InstanceType":{
     "Ref":"EC2InstanceASG"
   },
}
}
```

- `Scaling Configuration`: There are two scaling configurations, one to scale up and the other to scale down. Auto scaling will add/remove new EC2 instances as defined in the `ScalingAdjustment` field whenever the alarm goes off. Due to lack of space, only `WebServerScaleUpPolicy` is presented:

```
"WebServerScaleUpPolicy":{
"Type":"AWS::AutoScaling::ScalingPolicy",
"Properties":{
   "AdjustmentType":"ChangeInCapacity",
   "AutoScalingGroupName":{
     "Ref":"WebServerGroup"
   },
   "Cooldown":"60",
   "ScalingAdjustment":"1"
}
},
"WebServerScaleDownPolicy":{
.
.
}
```

- `Scaling Alarms`: Next are the alarms that are the qualifiers for the Auto Scaling group to add or remove EC2 instances. In this example, a new EC2 instance is added whenever the average CPU load is > 90% for a period of 5 minutes, and an EC2 instance is removed whenever the average CPU load is < 70% for a period of 5 minutes. Due to lack of space, only `CPUAlarmHigh` is presented here:

```
"CPUAlarmHigh":{
"Type":"AWS::CloudWatch::Alarm",
"Properties":{
   "AlarmDescription":"Scale-up if CPU > 90% for 5 minutes",
   "MetricName":"CPUUtilization",
   "Namespace":"AWS/EC2",
   "Statistic":"Average",
   "Period":"300",
   "EvaluationPeriods":"2",
   "Threshold":"90",
   "AlarmActions":[
     {"Ref":"WebServerScaleUpPolicy"}
   ],
   "Dimensions":[{
       "Name":"AutoScalingGroupName",
       "Value":{"Ref":"WebServerGroup" }
     }
   ],
   "ComparisonOperator":"GreaterThanThreshold"
}
},
"CPUAlarmLow":{
.
.
},
```

- `Auto Scaling Group`: Finally, the Auto Scaling group itself is configured to send messages to the SNS topic upon the launch and termination of EC2 instances:

```
"WebServerGroup":{
   "Type":"AWS::AutoScaling::AutoScalingGroup",
   "DependsOn":[
     "LaunchConfig","ElasticLoadBalancer"
   ],
   "Properties":{
     "AvailabilityZones" : [
         {"Fn::GetAtt" : [ "HASubnet" , "AvailabilityZone"] },
         {"Fn::GetAtt" : [ "PublicSubnet" , "AvailabilityZone"] }
     ],
```

```
        "LaunchConfigurationName":{
          "Ref":"LaunchConfig"
        },
        "MinSize":"1",
        "MaxSize":"1",
        "LoadBalancerNames":[
          {"Ref":"ElasticLoadBalancer"}
        ],
        "VPCZoneIdentifier":[
          { "Ref" : "HASubnet" },
          { "Ref" : "PublicSubnet" }
        ],
        "NotificationConfiguration":{
          "TopicARN":{"Ref":"A1SNSInfraAlert"
          },
          "NotificationTypes":[
            "autoscaling:EC2_INSTANCE_LAUNCH",
            "autoscaling:EC2_INSTANCE_LAUNCH_ERROR",
            "autoscaling:EC2_INSTANCE_TERMINATE",
            "autoscaling:EC2_INSTANCE_TERMINATE_ERROR"
          ]
        }
      }
    },
```

Centralized logging

When you move from a static environment to a dynamically scaled cloud-based environment, you need to pay close attention to the way you store, capture, and analyze log files generated by the OS and your application. As EC2 instances are instantiated and deleted by the autoscaling group, dynamically storing the log files locally is not recommended. Hence, there is a need for a centralized logging service to which all the applications and OSs log their data to; this makes it very convenient to search, view, analyze, and take action regarding the logs in real time from a centralized console.

You have the option to either rollout your own centralized logging infrastructure using the open source ELK stack (Elasticsearch, Logstash, and Kibana), or subscribe to one of the many third-party logging service providers. Since this a book is about AWS, we will work with CloudWatch as the logging and monitoring service.

Setting up CloudWatch

To enable logging from the EC2 instance to CloudWatch, a logging agent needs to be installed on the EC2 instances. As an example, we will show you the logging related to a Tomcat access log file. Other log files can be handled similarly. Ensure you install this agent in your base AMI image:

1. Install the AWS command client library as described in Chapter 4, in the *Scripting Auto Scaling* section.
2. The next step is to install the logging agent itself. Since our EC2 is based on Ubuntu, the agent needs to be downloaded and installed on the EC2 instance:
    1. Download the `CloudWatch` logging agent:

```
wget
https://s3.amazonaws.com/aws-cloudwatch/downloads/latest/awslogs-ag
ent-setup.py
```

    2. Install and configure the `Cloudwatch` agent. The `–region` command-line parameter specifies the AWS region in which your current AWS EC2 instances and infrastructure is running. The log file to push to CloudWatch is defined step 4; make sure the filename exists:

```
sudo python ./awslogs-agent-setup.py --region --us-west-2
Launching interactive setup of CloudWatch Logs agent ...
Step 1 of 5: Installing pip ...DONE
Step 2 of 5: Downloading the latest CloudWatch Logs agent
bits ... DONE
Step 3 of 5: Configuring AWS CLI ...
AWS Access Key ID [None]:
AWS Secret Access Key [None]:
Default region name [us-west-2]:
Default output format [None]:
Step 4 of 5: Configuring the CloudWatch Logs Agent ...
Path of log file to upload [/var/log/syslog]:
/var/log/tomcat7/access_log.log
Destination Log Group name
[/var/log/tomcat7/access_log.log]:
Choose Log Stream name:
1. Use EC2 instance id.
2. Use hostname.
3. Custom.
Enter choice [1]: 1
Choose Log Event timestamp format:
1. %b %d %H:%M:%S   (Dec 31 23:59:59)
2. %d/%b/%Y:%H:%M:%S (10/Oct/2000:13:55:36)
```

```
3. %Y-%m-%d %H:%M:%S (2008-09-08 11:52:54)
4. Custom
Enter choice [1]: 3
Choose initial position of upload:
1. From start of file.
2. From end of file.
Enter choice [1]: 2
More log files to configure? [Y]: n
```

3. After configuring and installing the CloudWatch instance in your base AMI image, start the logging agent:

```
sudo service awslogs start
```

4. From the CloudFormation web console, navigate to Logs; there will be an entry for `/var/log/tomcat7/access_log.log`, which implies the log agent has been installed and configured correctly.

Summary

In this chapter, we reviewed some of the strategies you can follow for deployment of your cloud application. We emphasized the importance of automating your infrastructure and setting up a DevOps pipeline. We followed this up with sections on setting up and monitoring a cloud-based backup solution. We also included a brief section on going live with your application on the cloud. Finally, we described deployment-related steps for our sample application.

The next chapter allows you to using the methods from the chapters thus far to design an end-to-end, AWS-based, Big Data application.

8

Designing a Big Data Application

In this chapter, we will present the design considerations for big data applications using AWS services. More specifically, we will explore AWS services and platforms such as Kinesis, EMR, Apache Spark, SageMaker, and Glue that are often the key components of such applications. Our focus will be on the best practices for using these AWS services in various big data applications such as machine learning and streaming analytics applications. Finally, in the hands-on exercise, we will create EMR-Spark clusters.

In this chapter, you will learn about the following:

- Characteristics of a big data application
- Analyzing streaming data with Amazon Kinesis
- Best practices for building serverless big data applications
- Best practices for distributed machine learning and predictive analytics on AWS
- Using Amazon SageMaker for machine learning applications
- Best practices for using Amazon EMR
- Understanding the security options for Amazon EMR and serverless applications
- Creating an EMR cluster

Introducing big data applications

Big data applications require the ability to ingest and store massive amounts of structured and unstructured data. We also need to be able access and/or process the data flexibly and securely. Additionally, we would like to future proof our big data solution (design and implementation) against rapidly evolving business use cases and technology.

There are three typical types of data-driven development:

1. Historical analysis and reporting supported by using services such as Amazon Redshift, Amazon RDS, Amazon S3, and Amazon EMR
2. Real-time processing and dashboards supported by using services such as Amazon Kinesis, Amazon EC2, and AWS Lambda
3. Intelligent applications supported by using services such as Amazon Deep Learning AMI, Amazon machine learning, and Amazon SageMaker

Traditionally, batch processing has been used to process massive volumes of data such as hourly server logs, generating weekly or monthly bills, daily website clickstream analysis, and daily fraud reports. As machine learning applications are increasingly becoming mainstream, such jobs have also included training machine learning models. However, recently, there is significant shift towards real-time streaming applications.

Organizations want to use streaming data as the incoming data loses value over time. They want to ingest data as it is generated and analyze it in real time to get insights, immediately. Examples of real-time data include events from mobile apps, web clickstream, application logs, IoT sensors, and so on. Stream processing may include computation of real-time metrics, real-time spending alerts/caps, real-time clickstream analysis, real-time fraud detection, and so on. For example, applications such as web analytics and leaderboards ingest web application data, compute top 10 users and persist to feed live apps. The continuous stream of data is typically processed over moving time windows or over a number of events. Similarly, in IoT applications sensor data is ingested, and metrics such as average temperature is computed every 10 seconds, and the time series analytic is then persisted to a serving database.

The main components of a streaming application include:

- **Data producer**: This continuously creates data and continuously writes the data to a stream.
- **Streaming service**: This durably stores data and provides temporary buffer for data preparation/pre-processing. This service needs to support a very high throughput.

- **Data consumer**: This continuously processes the data and also cleans, prepares, and aggregates incoming data.

Real-time analytics requirements have components that ingest, transform, analyze, react, and persist the event data. Such applications need to be durable, continuous, fast, reactive, available, and reliable.

There are three common patterns for streaming applications:

- **Streaming ingest-transform-load**: This delivers data to analytical tools faster and cheaper
- **Continuous metric generation**: This computes analytics as the data is generated
- **Actionable insights**: This reacts to analytics based off of insights.

The next wave of business applications includes predictive analytics applications. Predictive analytics is important because companies have been accumulating big data about customers, product/services, and operations for many years now and big data technologies have provided proven solutions to store and process this data. Companies are feeling an increasing pressure to turn data into insights about trends, classifications, detect anomalies, and provide feedback loops to improve their businesses. There is a strong desire to evolve from backward-looking monthly or quarterly reports to real-time alerts, and now to predict the future.

Enterprises want to answer customer-related, product-related, and business operations related questions. For instance, customer predictions include: Which customers are likely to be the most profitable? How much revenue should I expect this customer to generate? Which customers are likely to churn? Among all of our customers, which are likely to respond to a given offer? What's the probability that a given customer will respond to a given offer?

Similarly, product or service predictions typically include: what products should we offer or develop? What items are likely to be purchased together? Business operations predictions could include questions such as: are the metrics for a service nominal or anomalous? Is a specific equipment likely to fail within a given time period?

AWS components used in big data applications

AWS has managed service offerings to address typical requirements of big data applications. The following is the list of requirements and some of the associated AWS services that can be leveraged to address them:

- **Ingest and store**: Kinesis Streams, Kinesis Firehose, and Amazon S3
- **Prepare and transform**: AWS Lambda and AWS Glue
- **Analyze and reason**: Kinesis Analytics and Amazon Athena
- **Machine learning**: Amazon SageMaker
- **Access and user interface**: Amazon QuickSight and third-party tools

Additionally, there are security-related AWS services, features, and facilities available to ensure secure implementation of big data applications.

Analyzing streaming data with Amazon Kinesis

Streaming data processing is continuous and is done in real-time. You are continuously writing data to a streaming service such as Kinesis. Typically, operating on small-sized events (say a 1 KB event), writing to a stream, aggregating that data, and then persisting to Amazon S3.

Amazon Kinesis is made up of three services:

- **Amazon Kinesis Data Streams:** This helps you to build custom applications that process and analyze streaming data. You can build real-time applications with framework of choice – Kinesis Analytics, Spark on EMR, custom code on EC2, or custom code on Lambda. It is easy to administer, secure, and uses durable storage.
- **Amazon Kinesis Data Analytics:** This helps you to easily process and analyze streaming data with standard SQL. You can build powerful real-time applications such as continuous anomaly detection, continuous time-series analysis, continuous filtering, aggregation, and enrichment. It is easy to use, fully managed, and provides automatic elasticity.

- **Amazon Kinesis Data Firehose:** This helps you to easily load streaming data into AWS. It requires zero-administration and provides seamless elasticity. It supports direct-to-data store integration to Amazon S3, Amazon Redshift, and Amazon ElasticSearch Service. Typically, if you have many small files, then you would aggregate to larger files on S3 to optimize batch processing. It is serverless and supports continuous data transformations, compression, encryption, and you can use AWS Lambda to create a serverless ETL pipeline to get the data into AWS.

Best practices for serverless big data applications

There is a massive shift toward serverless computing for big data and other applications on AWS Cloud. The popularity of serverless stems from the huge advantages it provides in terms of having no servers to provision or manage; it scales with usage and you never pay for the idle time. Additionally, availability and fault tolerance are built-in.

Serverless nicely fits into big data platforms as you can flexibly mix and match serverless, managed, and virtualized services. These services can be easily leveraged to rapidly ingest, categorize, and discover your data, allow easy query and analysis of your data, transform and load data, provide custom event-based handlers, and so on. Overall, a serverless approach allows you to focus more on analytics / use cases and not on infrastructure or servers.

A serverless strategy can be applied to almost all aspects of big data applications, including data processing, data warehousing, reporting, real-time processing, predictive analytics, and artificial intelligence.

Some examples of AWS serverless options include:

- AWS Lambda (serverless compute) lets you run your code in the cloud. It is a fully managed and highly available service that is triggered through an API or state changes in your setup. It scales automatically to match the incoming event rate. It provides support for code written in Node.js (JavaScript), Python, Java, and C#. It is currently charged per 100 ms execution time.
- Amazon Athena (serverless interactive query service) allows you to query directly from Amazon S3 using ANSI SQL. It supports multiple data formats and you pay per query.

- AWS Glue (serverless catalog and ETL/ELT service) helps crawl, discover, and organize data. It is supports integration with managed and serverless analytics, Job Authoring and Job Execution (serverless ETL). You pay for what you consume.
- Amazon Kinesis (serverless streaming) makes it easy to capture, deliver, and process streams on AWS.

Best practices for using Amazon EMR

Amazon has made working with Hadoop a lot easier. You can launch an EMR cluster in minutes for big data processing, machine learning, and real-time stream processing with the Apache Hadoop ecosystem. You can use the Management Console or the command line to start a few nodes or a thousand nodes with ease.

Like EC2, EMR pricing is now changed from pay per hour to pay per second. If you stop your cluster—you stop paying immediately. This results in lower costs and you don't have to worry about the hourly boundary, anymore.

EMR makes a whole bunch of the latest versions of open source software available to you. There are 19 open source projects, presently. New releases are made every 4 to 6 weeks so latest versions of the open source projects are available. This is very useful especially for rapidly evolving open source projects such as Apache Spark where each release contains critical bug fixes and features. However, you are not forced to upgrade but a new release is made available, if you choose to use it. With EMR, you can spin up a bunch of instances and you could process massive volumes of data residing on S3 at a reasonable cost.

A variety of different cluster management options are supported, including YARN. You can run HBase, Presto (low latency, distributed SQL engine), Spark, Tez, and a variety of frontend tools such as ganglia, Zeppellin, notebooks, and SQL editors. Additionally, connectors to a variety of different AWS services are also available, for example, you can use Spark to load Redshift (using the Redshift connector, which under the hood uses Redshift commands to get a good throughput). You can access DynamoDB for analytics applications; use Sqoop to access relational data and so on.

One particularly interesting connector is AWS Glue. AWS Glue comprises three main components:

- **ETL service**: This lets you drag things around to create serverless ETL pipelines
- **AWS Glue Data Catalog**: This is a fully managed Hive metastore-compliant service. Earlier, the systems ran an external Hive metastore database in RDS or Aurora. This was great, if you shut down your cluster, all your metadata was persisted so you didn't have to recreate your tables with extra durability and availability (in case something happened to your metastore with MySQL on the master node). With Glue all that is fully managed. You have an intelligent metastore—you don't have to write DDL to create a table, you could just have Glue crawl your data, infer what the schema is, and create those tables for you. You can also have it add partitions. One thing that can be painful is to add partitions—if you are constantly updating your Hive tables, you need a process to kick-off to load that partition in—Glue catalog can do it for you. You can have a variety of complex data types that it supports as well.
- **Crawlers**: The crawlers let you the crawl data to infer the schema.

AWS Glue is a managed service, so you spend less time monitoring; as a fully managed service, it is also responsible for replacing unhealthy nodes and auto-scaling. It is easy to enable security options. It supports full customization and control, and you don't have to waste time creating and configuring the cluster. In most cases, the default settings are good enough, but even if you wanted to change them or install custom components, you have root access over all the boxes, so you can make any changes you need.

Understanding common EMR use cases

Using HBase for random access at a massive scale involves a lot of customers who are running HBase with HDFS. Now there is support for HBase using S3 object store for HFiles. Also, there is the ability to use Read Replica HBase cluster in another AZ. Shifting to S3 can save you 50% or higher on storage costs. Instead of sizing the cluster for HDFS, they can now size it for the amount of processing power required for the HBase Region Servers. The S3 option is also good for load balancing and disaster recovery across AZs. As S3 is available across a region, you don't have to replicate the data twice, that is, you don't need two full HDFS clusters. Now you can set up a smaller cluster for the Read Replicas that point to the same HFiles and you can drive the read traffic through there.

Real-time and batch processing involves utilizing EMR; you can use Kinesis for pushing data to Spark. Use Spark Streaming for real-time analytics or processing data on-the-fly and then dump that data into S3. If you don't have real-time processing use cases, then Kinesis Firehose is a great alternative too. The data can be cataloged in the Glue Data Catalog and then you can have the data accessible via a variety of different analytical engines. EMR supports several analytical engines including Hive, Tez, and Spark. Once the data is in the Data Catalog on S3, you can use Athena (serverless SQL queries), Glue ETL (serverless ETL), and Redshift Spectrum.

Data exploration with Spark using Zeppelin or Jupyter notebook. This allows you to arm your data scientists with a way to explore large amounts of data (instead of using one node, now you can spread the data across the cluster). It also makes it easy to move it to production.

There is a big rise in the use of Presto for ad hoc SQL queries (in combination with Athena). They approach the same thing from two different angles. Presto gives you advanced configurations and a way to build exactly what you need for your use case but you have to deal with the cluster management versus Athena where you just go to the console and start writing SQL. Now many BI tools support Presto as well for supporting low latency dashboards. You can also do traditional batch processing workloads using Spark.

Deep learning with GPU instances is where you can launch GPU hardware for EMR. There's support for MxNet. You can do end-to-end data engineering work. Support for TensorFlow is coming.

Typical ML projects implement a multi-step process, including ETL, feature engineering, model training, model evaluation, model deployment, and model scoring and updates. Such pipelines need to support batch model training and real-time ML model serving. Using Apache Spark for implementing ML pipelines is very popular as it supports each step in a ML pipeline, scales for small and large jobs, good ML libraries, and has an active user base.

There are several options for deploying Spark on AWS. For example, you can use EC2 as it can support for batch/streaming, integrates with tooling, spin up/down clusters, larger/smaller clusters. Additionally, it also has support for different versions of Hadoop and Spark. However, using EC2 for Spark deployment places a huge management burden on us. Hence, EMR can be a simpler and better alternative here. It is simple to provision and you can use a wizard (and then generate the commands for the command line from it, if required). You can create tags for cost management and send logs to S3.

Lowering EMR costs

If you are paying for Hadoop nodes that are not doing anything, then you are just burning money. There are ways you can batch up your workloads. Take an inventory of the jobs you have and tweak them to run in a batch mode and shutdown the cluster in-between those times. You can separate out clusters with auto-scaling instead of sizing and running it for all your workloads. You should shutdown the cluster when you can, to stop paying for it, unnecessarily. You can use Amazon Linux AMI with preinstalled customizations for faster cluster creation and use auto-scaling to minimize costs for long-running clusters.

Using Amazon EC2 Spot and Auto Scaling

Using spot instances is a great way to save money compared to using on-demand instances. However, be more careful in a SLA-driven environment (where you cannot withstand any failures). In most cases, the odds of a failure are pretty low, and Hadoop itself can handle several node failures so that even if some nodes are taken away you may still be good—you should run task nodes that don't have data so as to not impact HDFS. But still, there might be failures and you could lose a bunch of nodes and Spark may not be able to re-compute a DataFrame. Having the logic to just kill the cluster and create a new one on-demand for that one job is more expensive but over a period of time, the savings still make it a worthwhile approach. Templatize the architecture so that you can quickly recover if something happens, is doable and recommended when you are using spot instances, and you have workloads that are SLA bound.

Using instance fleets is similar to using spot fleets; however, you do not specify the instance to use. You will specify a list of instance types and Amazon will do the best based on the availability of instances to give some combination of instances you want. This solves the following three problems:

- Provision across instance types. When requesting instance types sometimes they are not available, then AWS will try to get the capacity you need based on instance types available from the list of instance types you specify. You give four or five different choices and AWS will try to provision the end capacity you need.
- You can also specify the list of AZs and EMR will select the optimal EC2 AZ to launch the cluster in.
- If spot instances are not available and I still need to run the job, then the on-demand feature can be used. For example, if I can't get all the instances I need for 30 minutes, then switch to on-demand.

EMR auto-scaling—whether for long running jobs or even for transient workloads to some degree—uses some of the features of application auto-scaling under the hood, but AWS has stitched things together for you. You need to specify what you need and AWS does the rest. (No need to specify CloudWatch metrics yourself.) You can use a variety of different CloudWatch metrics such as YARN memory or the ratio of containers pending to containers active, which is a kind of a proxy for if you gave me another node do you have any YARN containers to put on it. So give me another node.

There are a bunch of CloudWatch metrics that can be used, but also use custom metrics—for example, there is currently no metric for aggregate CPU on the cluster, but if you have installed Ganglia, then there is—so pump that to CloudWatch and scale based on that.

Auto-scaling points involves EMR scales-in at YARN task completion. It selectively removes nodes with no running tasks (that is, a YARN container); `yarn.resourcemanager.decommissioning.timeout`(default is one hour). If a particular node is running a YARN container then it will wait for this value (1 hour), and if no other node is free then it will take this node and kill the YARN container. If you have a use case that involves caching a large Spark DataFrame and all the YARN containers are required, then you can set this parameter to an arbitrary high value to prevent a node from being taken away while it is in use. Ultimately, the configuration parameter settings depend on your specific use case.

Best practices for distributed machine learning and predictive analytics

Enterprise predictive analytics applications need to be highly scalable and need to support agility in feature development and deployments. There are a plethora of open source tools and solutions available. However, using managed services eliminates infrastructure-related heavy lifting and leads to agility.

Such applications use a variety of supporting AWS services, including the following:

- **Amazon S3:** Store anything (object store), Scalable/Elastic, High durability, effectively infinite inbound bandwidth, extremely low cost, and data layer for virtually all AWS services. Amazon S3 is a great choice for cloud-based data lake because of durability (designed for 11 9s durability), availability (designed for 99.99% availability), high performance (features such as multiple uploads, Range GET), ease of use (supports simple, REST APIs, AWS SDKs, read-after-create consistency, event notification, and lifecycle policies), scalability (save as much data as you need, scale storage and compute independently, no minimum usage commitments), and is integrated with other services such as Amazon EMR, Amazon Redshift, and Amazon DynamoDB.
- **AWS Glue:** This provides a managed ETL engine, and a Job scheduler. It is built on Apache Spark and is integrated with S3, RDS, Redshift, and any JDBC data store.
- **Glue Data Catalog:** This manages table metadata through a Hive Metastore API or Hive SQL, and is supported by tools such as Hive, Presto, and Spark. There are several extended services available to support additional functionality such as search metadata for data discovery, connection information for JDBC URLs and credentials, classification for identifying and parsing files, versioning of table metadata as schemas evolve and other metadata are updated. You can populate the catalog using Hive DDL, bulk import, or automatically through crawlers.
- **Glue Crawlers:** The Glue crawlers are used to auto-populate data catalog. They come with built-in classifiers that detect the file type and extract the record schema (record structures and data types). It can also auto-detect Hive-style partitions and group similar files into one table. You can set up a schedule for running the crawlers. They are implemented as a serverless service so you only pay when the crawlers are actually run.
- **Amazon EMR:** This includes scalable Hadoop clusters as a service. The service includes Hadoop, Hive, Spark, Presto, HBase, etc. distributions. It is easy to use and a fully managed service. You have a choice of using On-demand, reserved, or spot instances for your cluster. Storage options include HDFS, S3, and Amazon EBS filesystems. It is integrated with AWS Glue, and you can configure end-to-end security for the data-in-flight and data-at-rest.

Using Amazon SageMaker for machine learning

Typically, the machine learning process comprises the following steps:

- **Data wrangling:** This involves setting up and managing notebook environments. This will help you to get data to notebooks securely.
- **Experimentation:** This involves setting up and managing clusters. This will help you scale/distribute ML algorithms.
- **Deployment:** This involves setting up and managing inference clusters. This will help you manage and autoscale inference APIs with testing, versioning, and monitoring.

The preceding cycle can take as long as 6 to 18 months. However, given the potential of machine learning and AI to empower and improve businesses, the effort is often worthwhile. Challenges in large-scale machine learning include storing and processing lots of data, model selection, and production deployments and model updates. Hyperparameter tuning is expensive and incremental training is a problem (significant repetitious training on data already trained previously). As the amount of investment required goes up with increasing data/model size, businesses limit the size of the data used for training. This results in wasted opportunity due to unused data (as the data is incrementally dropped from the training set with time).

Amazon SageMaker aims to make the machine learning process easier. It is a managed service that provides the quickest and easiest way for data scientists and developers to get ML models from idea to production. It provides an end-to-end machine learning platform by supporting data exploration, model training, and hosting with minimal setup, and you pay by the second.

Amazon SageMaker requires minimal setup for data exploration and resizable as per your needs. This involves common tools pre-installed with easy access to your data sources. There are no servers to manage and a modular architecture lets you use only what you need. It has a pay as you go model and a free trial is available to you get started quickly.

Amazon optimized algorithms can be applied for distributed training using the AWS SDK, or Apache Spark SageMaker Estimators, deep learning scripts (using TensorFlow and Gluon), or your custom algorithm Docker image. Algorithms designed for huge datasets: Streaming datasets, for cheaper training. Train faster in a single pass. Greater reliability on extremely large datasets.

Choice of several ML algorithms include XGBoost, Factorization Machines, and Learn Linear for classification and regression, K-means and PCA for clustering and dimensionality reduction, image classification with convolutional neural networks, LDA, and NTM for topic modeling, and seq2seq for translation—with more algorithms to follow.

Amazon SageMaker makes a whole bunch of machine learning tasks ranging from the management of notebook environments to easy data exploration in Jupyter notebooks. It also supports one-step deployment to quickly deploy ML models in production. It provides a low latency, high throughput, and high-reliability service.

You can get started using Amazon SageMaker with notebook samples. You will need to modify the code to access your own data sources.

Understanding Amazon SageMaker algorithms and features

In this section, we present some of the key algorithm design choices and features in Amazon SageMaker that make them an excellent option for your machine learning applications. The following are some of its key features:

- **Based on streaming**: Amazon SageMaker algorithms are all based on streaming (data points seen only once). Based on the streaming data, a state (of fixed size) is set; so it does not matter how much data you have streamed (the data structure used is not going to grow in size). Hence, the memory footprint of the algorithm is going to be fixed and the run time/costs remain linear to the size of the data.
- **Support for incremental training**: These algorithms are designed to support incremental training. For example, if you are training on 2 days' worth of data, then you will have to train on day 1 and 2 data, and later on day 2 and day 3 data. In such a situation, with these algorithms, you can serialize and persist the state after days 1 and 2. When day 3 data is available, you can deserialize the state (at this stage you are exactly at the end of day 2), and then you can proceed with just processing day 3 data. Hence, you save on compute because you don't have to retrain on day 2 data. Another significant advantage is that you no longer have to choose how far back you want to go for including the data in your training set. In our example, you end up effectively training on all of the data from days 1, 2, and 3. So, overall it becomes faster, cheaper, and more accurate.
- **Support for GPUs**: All the algorithms work on both CPU and GPU.

- **Distributed architecture**: It is linearly scalable. If each node has one third of the data, then it will run in one third of the time. As the states for each node may be different after the training is completed, there is also a shared state (a global state that all these nodes share). The local state is synchronized with the global state.
- **Support for efficient model selection**: Amazon Kinesis streams can be consumed as input streams. This allows you to do post training processing (and not only pretraining) to explore different models. For model selection, hyperparameter tuning can be done based on the state to generate multiple models instead of retraining on the same data.
- **Support for abstraction and containerization**: You can build on a desktop (using CPU) and then deploy in a distributed GPU environment. These features also give you superior production readiness. The deployment can run the same solution on 1 GB or 1 TB of data.

Amazon SageMaker algorithms can be used from the command line (specify the algorithm, input data, and hardware), and also from the Amazon SageMaker Notebooks. You can also deploy directly from the notebook itself as shown in `Chapter 9`, *Implementing a Big Data Application*.

Security overview for big data applications

In this section, we will shift our focus to AWS big data application security features. More specifically, we will discuss the features and options for securing EMR clusters and serverless applications.

Securing the EMR cluster

In this section, we provide a quick overview of EMR security, including encryption, authentication, and authorization.

Encryption

EMR supports end-to-end encryption for a variety of different frameworks. You can configure all this in a couple of clicks. You can do S3 server-side or client-side encryption, encrypt all the local disks of your cluster so any executor spills or HDFS blocks getting written get encrypted on the local disk filesystem. You can also encrypt Spark, Tez, MapReduce, HBase, Hive, Presto, and Pig for all the data blocks in flight.



For more details on encrypting data-at-rest and data-in-transit, refer to AWS documentation: `https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-data-encryption.html`.

Authentication

EMR step (EMR API) is a straightforward way using AWS credentials to authenticate users. LDAP support is available for HiveServer2, Presto Coordinator, Spark Thrift Server, Hue Server, and Zeppelin Server. For SSH with EC2 key pair, the user ID is `hadoop` (super user).

Authentication with Kerberos involves using a secret-key cryptography to provide strong authentication so that passwords or other credentials aren't sent over the network in an unencrypted format. You can do cross-realm with AD and have all the corporate users SSH in as themselves. This is useful for auditing and governance purposes. You can easily launch a large cluster that is fully *Kerberosized* with KDC on the master node with service principals for all cluster nodes. So, if you want to do a one-way trust with your AD, you can SSH in as yourself.



For more details on using Kerberos authentication, refer to AWS documentation: `https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-kerberos.html`.

Authorization

You could be storage-based using ACLs or permissions on storage level to determine whether you can do the job or not. HDFS supports ACLs and you can configure it. EMRFS/S3 (S3 access control). EMRFS is storage-based, fine-grained authorization useful in shared cluster scenario where each user assumes a different IAM role to limit what they can or can't access on S3. You can map an IAM role to users, groups, or S3 prefixes.



For more details on using EMFRS authorization for data on S3, refer to AWS documentation: `https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-emrfs-authz.html`.

HiveServer2 and Presto support SQL standards based access control. For example, a given user can access or not access a specific table. HBase supports cell level access control. With Kerberos, you can authenticate users to manage access to YARN queues. At EMR cluster level ,you can use IAM and tags to implement access control. Finally, you can use Apache Ranger (essentially a policy engine for Hadoop) on edge node (using CloudFormation).

Securing serverless applications

We want to prevent unauthorized access and use of information. Additionally, we want to ensure that the application or service works as intended and only as intended. Another advantage of serverless is that we are working in a SaaS model where security is almost entirely the cloud provider's responsibility (in the shared responsibility model).



For more details, refer to *Securing Serverless Applications - Step-by-Step*, *Mark Nunnikhoven*, `https://www.youtube.com/watch?v=B3j4xql7we0`.

Securing serverless applications comprise three components: AWS services, application code, and data flows.

The following are the steps you need to perform for serverless security:

1. What data is involved in the app? Map out the data involved in the application to understand your risk. For example, PII data, credit card details, and other such sensitive data and the code that processes it.
2. What is the value of that data? After mapping the data involved in the application you should assign a value to it based on the risk or sensitivity of data.
3. What services access the data? Identify each of the services and how they can configure them. List the services used and assign a risk score to them.
4. Verify compliance eligibility. For example, if there are payments involved in your application, then the credit card information is sensitive and comes under PCI compliance. AWS has certified services for processing this data but remember to include supporting services such as AWS IAM and AWS KMS to your list.
5. Configure each service appropriately. S3 and IAM defaults are nonpublic and deny everything, which is a good thing. You need to understand the configuration parameters and choices for each service.
6. Add automated tests for each configuration. Test it after you have configured the services to ensure what you set up is exactly what is running in production. It's good practice to test constantly—re-run all the tests after any changes.

7. Write better code. Code quality is a problem and we can do better from a code perspective. Ensure that the quality of code is improving over time with more experience.
8. Reduce and verify dependencies. Be careful about imports (dependencies include code written by different people at different skill levels).
9. Add automated tests for the code. Test the code thoroughly. Creating automated test suites will help ensure the process is run as frequently as necessary with minimum effort.
10. Security test or profile the code. Look for known issues in the code. There are many static analysis tools available, including the open source ones.
11. Monitor the flow of information. You can monitor the flow of information (after the mapping is done). AWS X-Ray can be used to help analyze and debug the distributed applications. Getting to know how information is flowing in your application can also help with configuring the security for your applications. You can leverage Amazon CloudWatch to set up a trigger that sends the events to AWS Lambda. You can then create a whitelist of known good events/service calling which can itself be a serverless application. If a bad event is received, then you can send appropriate alerts. You can also leverage Amazon Macie to automatically discover, classify, and protect sensitive data in AWS. Amazon Macie recognizes sensitive data such as personally identifiable information (PII), and provides you with dashboards and alerts with respect to how this data is being accessed or moved. Currently, Amazon Macie supports S3 storage only. You can also analyze CloudTrail data for anomalies and configuration changes, for example, changing a S3 bucket access from private to public will raise an alert.

Understanding serverless application authentication and authorization

Learn how to implement identity management for your serverless apps using Amazon Cognito User Pools, Amazon Cognito Federated identities, Amazon API Gateway, AWS Lambda, and AWS Identity and Access Management (IAM).

Sign-up and sign-in. How do you store credentials? Never store passwords in plaintext. It is vulnerable to rogue employees. A hacked DB can result in all the stored passwords being compromised. The use of hashed passwords does not solve problem because of MD5/SHA1 collisions, the use of Rainbow tables (for reversing cryptographic hash functions, usually used for cracking password hashes), and dictionary and brute-force attacks (GPUs are capable of computing billions of hashes/second). A better approach is to salt hash these passwords by incorporate app-specific salt and random user-specific salt or use algorithm with a configurable number of iterations to slow down brute-force attacks. But what if you did not want to transfer the password over the wire or even save it in the database. In such cases you can use Secure Remote Password (SRP) protocol or Verifier-based protocol. Using these protocols ensures that passwords never have to travel over the wire and they are resistant to several attack vectors. They ensure **Perfect Forward Secrecy** (**PFS**) property for secure communications in which compromises of long-term keys do not compromise past session keys

So now you have taken care of Secure Password handling requirement; what about other requirements such as multifactor authentication, enforcing password policies, and encrypting all data server-side. On top of it, what about user flows such as registration, verification of email/phone number, secure sign-in, forgot password, change password, and sign-out functionality. Thankfully, there is an AWS service for it—Amazon Cognito. More specifically, Amazon Cognito User Pools (managed user directory).

A typical set of steps from a mobile app to Amazon Cognito User Pools include the following:

1. A register request is sent from the mobile app
2. A verification SMS/email is sent by Cognito
3. The user confirms the registration
4. Cognito responds with a successful registration message
5. An authenticate (via SRP) request is sent from the mobile app
6. Cognito returns the requisite JWT token (JSON Web tokens)

An alternate flow after successful registration is also possible if you want to implement CAPTCHA or MFA for Authentication (via SRP). You can define a custom authentication challenge. You can hook in a lambda function into the flow. Here, the steps after a successful registration will include the following:

1. An authenticate (via SRP) request is sent from the mobile app
2. A custom challenge (CAPTCHA, Custom 2FA) is presented by Cognito

3. A challenge response is sent by the mobile app (with a Lambda function triggered to verify the authentication challenge response)
4. Cognito returns the requisite JWT token

In fact, you can hook in Lambda triggers at each of the preceding steps, as required.

The JWT token consists of a header, payload, and a signature (all three together as a Base64 encoded blob of text makes up a JWT token).

You get three types of tokens from Cognito, which are as follows:

- **Access token**: This provides access or programmatic API interactions.
- **Identity token**: This can be used for downstream pseudo-authentication. You can use this token to dynamically change the experience of the user by just reading the token.
- **Refresh token**: This token is required because the Access and Identity tokens from Cognito are only good for one hour when issued. But a refresh token can be valid for as long as 30 days. You can use the refresh token to get a new access or identity token.

With Cognito User Pools, a user will be able to register, sign-in, and sign-out. Now, if the user needs to access AWS resources, for example, each user has a profile and they want their profile picture. We want to take advantage of Amazon S3 for that. To give them secure fine-grained access to S3, we need Amazon Cognito Federated Identities. This service allows you to exchange tokens from User Pools for AWS native credentials for federating access to AWS resources. Amazon Cognito Federated Identities calls AWS Security Token Service (STS), which is essentially a cloud-based token vending machine.

Prior to December 2016, Amazon Cognito Federated Identities had unauthenticated and authenticated user roles. Now, for fine-grained, role-based access control, you have two more features for authenticated users Choose a role from rule and Choose a role from token. Using the Cognito groups feature (for Choose role from token), you can specify an IAM role for each group and the users of that group will have the associated permissions. Note that a user can have only one IAM role active at a time.

You've set up the basics. Now, let's look at authorization requirements of your serverless application. If your application uses API Gateway, Lambda, and DynamoDB, then you can actually offload all of your authorization requirements to the API Gateway.

API Gateway supports three types of authorizations, which are as follows:

- Amazon Cognito User Pools using User Pools Authorizers. (We cannot differentiate admin user from other users; hence, we will need more fine-grained authorization.)
- Amazon Cognito Federated Identities using AWS IAM authorization. This is the most common option used by most businesses. It provides fine-grained access control.
- Custom Identity Providers using Custom Authorizers.



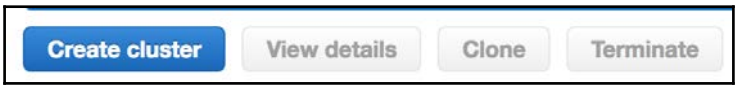
For more details, refer to *Serverless Authentication and Authorization, Justin Pirtle and Vladimir Budilov*, `https://www.youtube.com/watch?v=B3j4xql7we0`.

Configuring and using EMR-Spark clusters

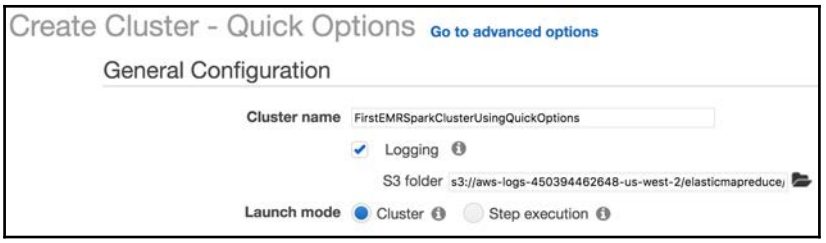
In this section, we will present two simple examples of EMR clusters suitable for basic Spark development. In the first example, we will spin up an EMR cluster, start the Spark shell, and do some Spark-Scala work. In the second example, we will spin up an EMR cluster and run a simple Spark program.

Follow the step-by-step instructions specified next for this hands-on exercise:

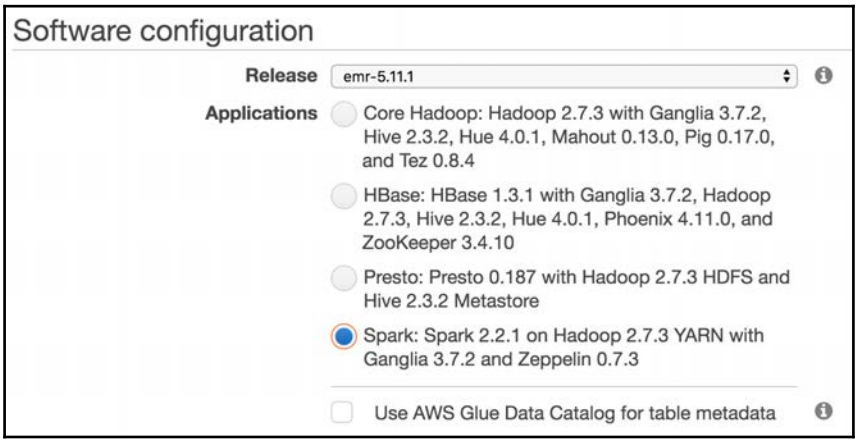
1. Log in to the AWS Management Console and open the Amazon EMR console and click on the **Create cluster** button:



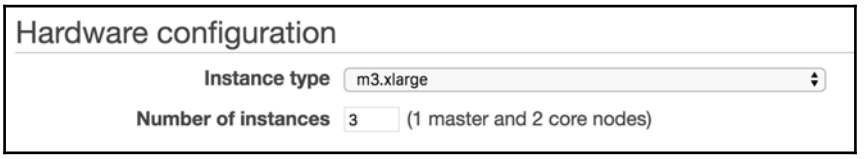
2. We will use the **Create Cluster - Quick Options** for selecting the options for our cluster. Specify a name for the cluster (`FirstEMRSparkClusterUsingQuickOptions`). Choose **Launch mode** as **Cluster**:



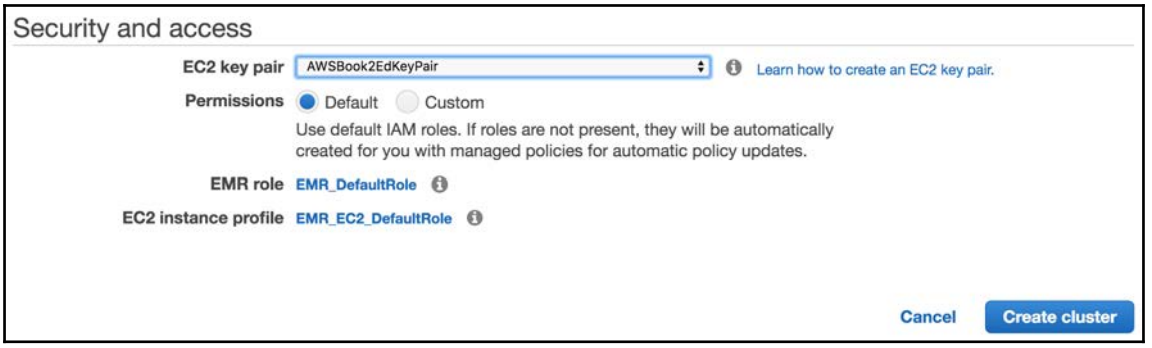
In the **Software configuration** section, ensure you have selected the latest available version of EMR and select the Spark option:



Leave the defaults for **Hardware configuration** unchanged. We will be creating a three-node cluster with suggested instance types:



Select your EC2 key pair from the dropdown list. Click on the **Create cluster** button:



3. It can take a few minutes for the cluster to start up. You should see the following screen with a **Starting** message displayed (in green):



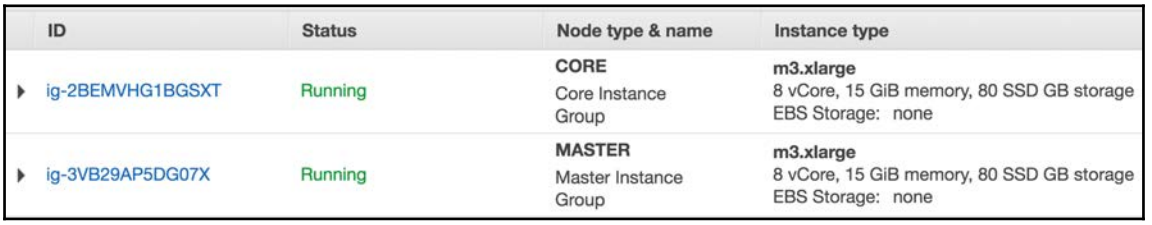
4. After the cluster is up and running, the **Starting** message changes to **Waiting**:



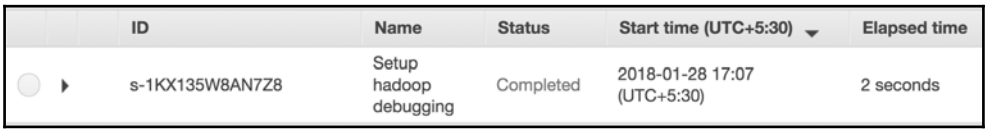
5. You can click on the AWS CLI export button to capture the EMR create cluster command that you can use to create this cluster (in the future) from the command line in one step:



6. You can view the nodes and their status by clicking on the **Hardware** tab:



7. You can view the defined jobs or steps by clicking on the **Steps** tab:



8. Click on the **Clusters** link on the left-side menu for a listing of your clusters and their status:



9. You can also list all the EMR clusters using AWS CLI command. Note the ID value for the cluster:

```
Aurobindos-MacBook-Pro-2:Downloads aurobindosarkar$ aws emr list-
clusters
```

10. Go to the IAM console and edit the default security group inbound rules to include port 22. SSH into the EMR Master node as shown here (copy the public DNS for the master node from the Summary tab):

```
ssh -i ./AWSBook2EdKeyPair.pem hadoop@ec2-34-216-77-144.us-
west-2.compute.amazonaws.com
```

```
Last login: Sun Jan 28 11:37:10 2018

       __|  __|_  )
       _|  (     /   Amazon Linux AMI
      ___|\___|___|

https://aws.amazon.com/amazon-linux-ami/2017.09-release-notes/
6 package(s) needed for security, out of 7 available
Run "sudo yum update" to apply all updates.

EEEEEEEEEEEEEEEEEEE MMMMMMMM           MMMMMMMM RRRRRRRRRRRRRRR
E::::::::::::::::::E M:::::::M         M:::::::M R::::::::::::::R
EE:::::EEEEEEEEE:::E M::::::::M       M::::::::M R:::::RRRRRR:::::R
  E::::E       EEEEE M:::::::::M     M:::::::::M RR::::R      R::::R
  E::::E             M::::::M:::M   M:::M::::::M   R:::R      R::::R
  E:::::EEEEEEEEEE   M:::::M M:::M M:::M M:::::M   R:::RRRRRR:::::R
  E::::::::::::::E   M:::::M  M:::M:::M  M:::::M   R::::::::::::RR
  E:::::EEEEEEEEEE   M:::::M   M:::::M   M:::::M   R:::RRRRRR::::R
  E::::E             M:::::M    M:::M    M:::::M   R:::R      R::::R
  E::::E       EEEEE M:::::M     MMM     M:::::M   R:::R      R::::R
EE:::::EEEEEEEE::::E M:::::M             M:::::M   R:::R      R::::R
E::::::::::::::::::E M:::::M             M:::::M RR::::R      R::::R
EEEEEEEEEEEEEEEEEEE MMMMMMM             MMMMMMM RRRRRRR      RRRRRR

[hadoop@ip-172-31-23-43 ~]$
```

11. You can display the details of your cluster using the describe-cluster command and specifying the cluster ID (copy the cluster ID value from the EMR Clusters console). The output provides cluster-level details such as status, hardware and software configuration, VPC settings, bootstrap actions, and instance groups.

```
[hadoop@ip-172-31-23-43 ~]$ aws emr describe-cluster --cluster-id
j-1O96RAZBTX0VI

{
    "Cluster": {
        "Status": {
            "Timeline": {
                "ReadyDateTime": 1517139426.457,
                "CreationDateTime": 1517139044.608
            },
            "State": "WAITING",
            "StateChangeReason": {
                "Message": "Cluster ready after last step
completed."
            }
        },
        "Ec2InstanceAttributes": {
            "EmrManagedMasterSecurityGroup": "sg-c086ddbc",
            "RequestedEc2AvailabilityZones": [],
            "RequestedEc2SubnetIds": [
                "subnet-3305a255"
            ],
            "Ec2SubnetId": "subnet-3305a255",
            "IamInstanceProfile": "EMR_EC2_DefaultRole",
            "Ec2KeyName": "AWSBook2EdKeyPair",
            "Ec2AvailabilityZone": "us-west-2a",
            "EmrManagedSlaveSecurityGroup": "sg-8482d9f8"
        },
        "Name": "FirstEMRSparkClusterUsingQuickOptions",
        "ServiceRole": "EMR_DefaultRole",
        "Tags": [],
        "TerminationProtected": false,
        "ReleaseLabel": "emr-5.11.1",
        "NormalizedInstanceHours": 0,
        "InstanceCollectionType": "INSTANCE_GROUP",
        "Applications": [
            {
                "Version": "3.7.2",
                "Name": "Ganglia"
            },
            {
                "Version": "2.2.1",
```

```
                    "Name": "Spark"
                },
                {
                    "Version": "0.7.3",
                    "Name": "Zeppelin"
                }
            ],
            "KerberosAttributes": {},
            "MasterPublicDnsName": "ec2-34-216-77-144.us-
west-2.compute.amazonaws.com",
            "ScaleDownBehavior": "TERMINATE_AT_TASK_COMPLETION",
            "InstanceGroups": [
                {
                    "RequestedInstanceCount": 2,
                    "Status": {
                        "Timeline": {
                            "ReadyDateTime": 1517139426.484,
                            "CreationDateTime": 1517139044.625
                        },
                        "State": "RUNNING",
                        "StateChangeReason": {
                            "Message": ""
                        }
                    },
                    "Name": "Core Instance Group",
                    "InstanceGroupType": "CORE",
                    "EbsBlockDevices": [],
                    "ShrinkPolicy": {},
                    "Id": "ig-2BEMVHG1BGSXT",
                    "Configurations": [],
                    "InstanceType": "m3.xlarge",
                    "Market": "ON_DEMAND",
                    "RunningInstanceCount": 2
                },
                {
                    "RequestedInstanceCount": 1,
                    "Status": {
                        "Timeline": {
                            "ReadyDateTime": 1517139395.559,
                            "CreationDateTime": 1517139044.624
                        },
                        "State": "RUNNING",
                        "StateChangeReason": {
                            "Message": ""
                        }
                    },
                    "Name": "Master Instance Group",
                    "InstanceGroupType": "MASTER",
```

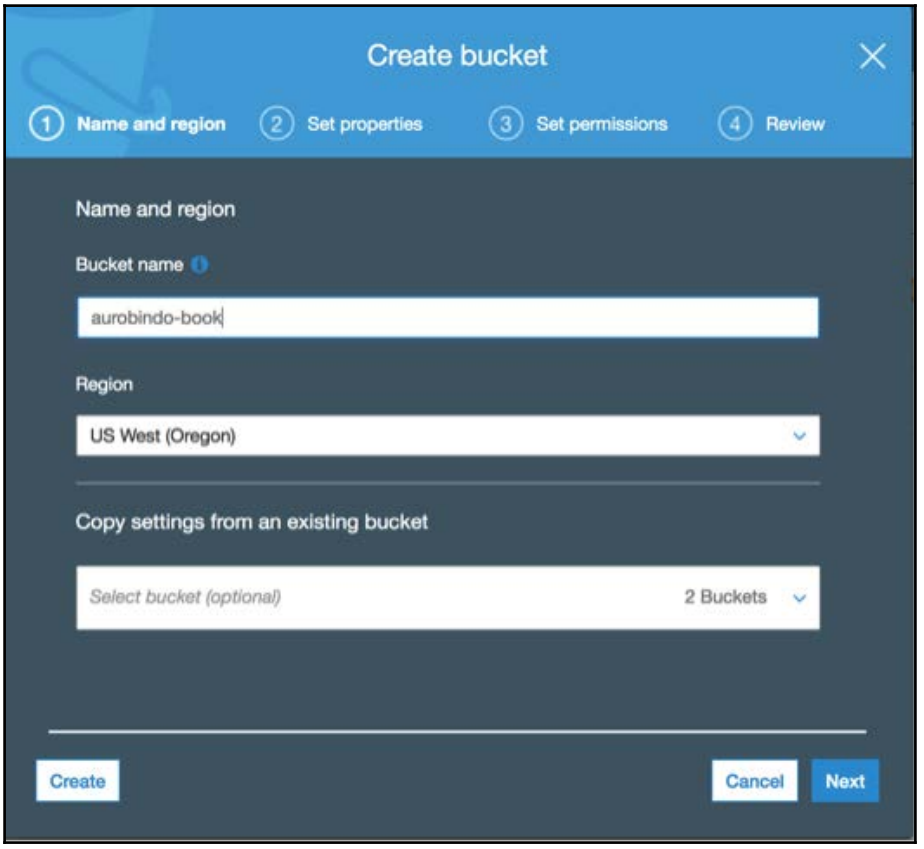
```
                    "EbsBlockDevices": [],
                    "ShrinkPolicy": {},
                    "Id": "ig-3VB29AP5DG07X",
                    "Configurations": [],
                    "InstanceType": "m3.xlarge",
                    "Market": "ON_DEMAND",
                    "RunningInstanceCount": 1
                }
            ],
            "VisibleToAllUsers": true,
            "BootstrapActions": [],
            "LogUri": "s3n://aws-logs-450394462648-us-
west-2/elasticmapreduce/",
            "AutoTerminate": false,
            "Id": "j-1O96RAZBTX0VI",
            "Configurations": [
                {
                    "Properties": {
                        "maximizeResourceAllocation": "true"
                    },
                    "Classification": "spark"
                }
            ]
        }
}
```

12. You can start the Spark shell as shown here:

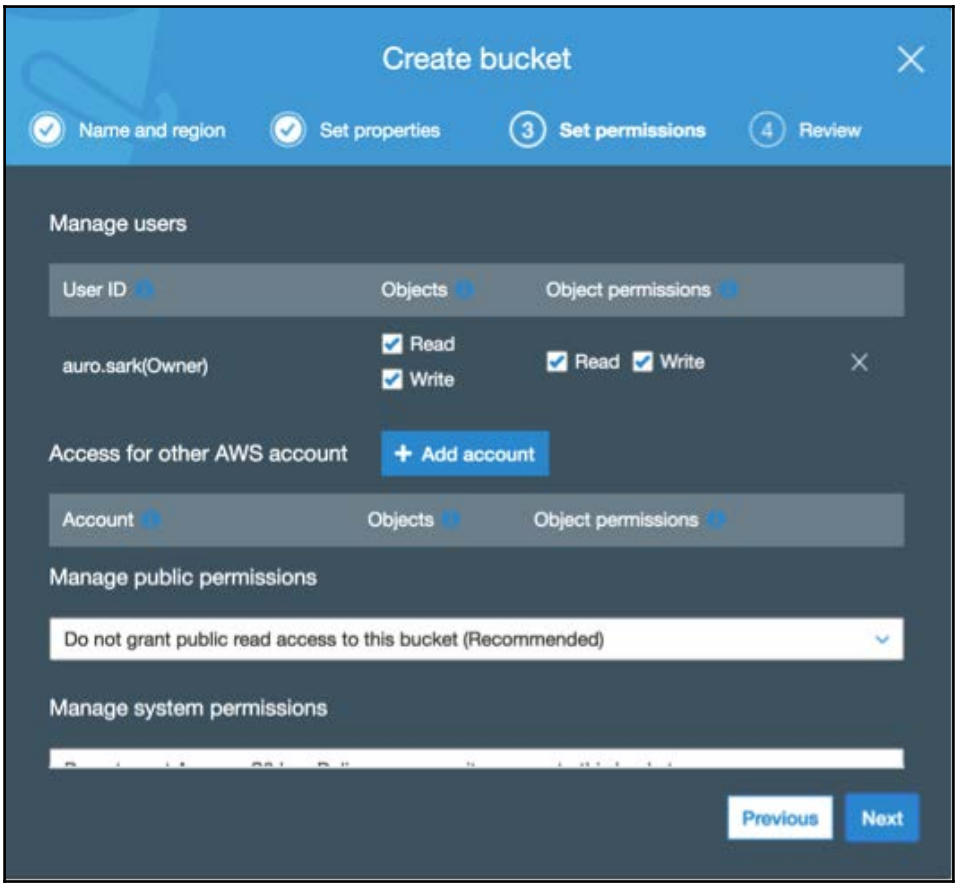
```
[hadoop@ip-172-31-23-43 ~]$ spark-shell
```

```
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
18/01/28 12:30:28 WARN Client: Neither spark.yarn.jars nor spark.yarn.archive is set, falling back to uploadin
g libraries under SPARK_HOME.
Spark context Web UI available at http://ip-172-31-23-43.us-west-2.compute.internal:4040
Spark context available as 'sc' (master = yarn, app id = application_1517139237616_0003).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.2.1
      /_/

Using Scala version 2.11.8 (OpenJDK 64-Bit Server VM, Java 1.8.0_151)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

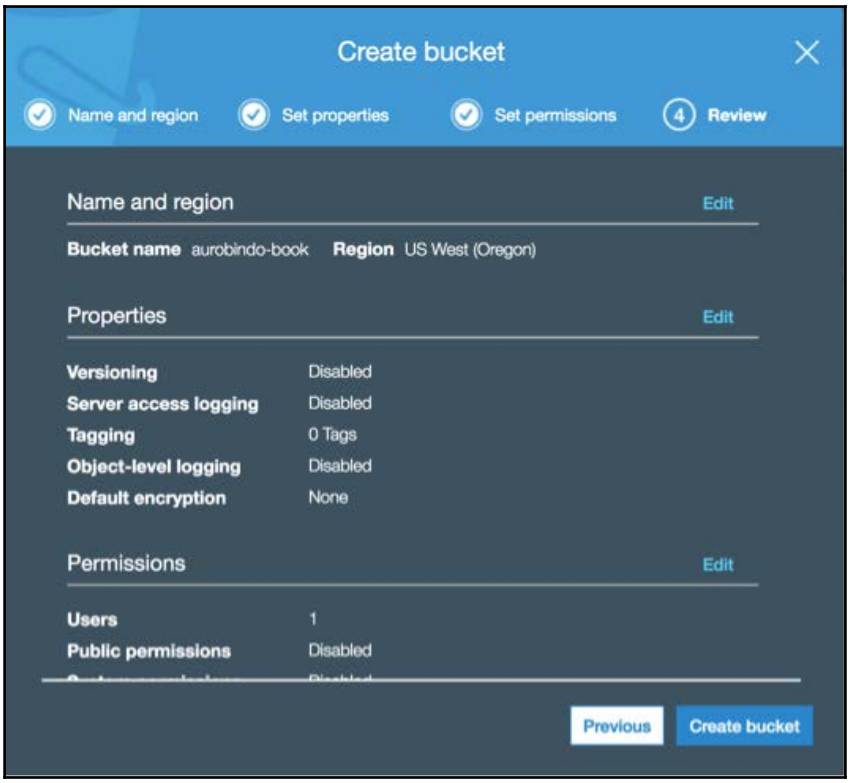
13. For illustration purposes, we will show a few processing steps in the Spark shell (using Scala). First, download a book (in the `.txt` format) from: `http://www.gutenberg.org/ebooks/35688`. We will use the downloaded file as input. Go to the S3 console to create an S3 bucket. Specify the **Bucket name** (in our example, we have named it `aurobindo-book`). Click on the **Next** button:



14. Leave the default selections unchanged and click on the **Next** button:



15. Review the Bucket parameters and click on the **Create bucket** button:



16. You should see the newly created S3 bucket listed on the S3 console as shown in the following screenshot:



17. Upload the input file to the newly created S3 bucket as shown here:

```
Aurobindos-MacBook-Pro-2:Downloads aurobindosarkar$ aws s3 mv
pg35688.txt s3://aurobindo-book/aliceinwonderland.txt
```

18. Go to the specific S3 bucket's contents to see the file listed:



19. Switch back to the Spark shell. Read the input file into a Spark RDD (Resilient Distributed Dataset) and display the first five lines (in the RDD) as shown next:

```
scala> val fileRDD = sc.textFile("s3://aurobindo-
book/aliceinwonderland.txt")

scala> fileRDD.take(5).foreach(println)
The Project Gutenberg EBook of Alice in Wonderland, by Alice
Gerstenberg
This eBook is for the use of anyone anywhere at no cost and with
almost no restrictions whatsoever. You may copy it, give it away or
re-use it under the terms of the Project Gutenberg License included
```

20. The logic used in the following code basically computes the word count for all the words in the book:

```
scala> val counts = fileRDD.flatMap(line =>
line.toLowerCase().replace(".", " ").replace(",", " ").split("
")).map(word => (word, 1L)).reduceByKey(_ + _)
```

21. We display the results for 10 words as shown here:

```
scala> counts.take(10).foreach(println)
 (goals,1)
 (brother,1)
 (lessons;,1)
 (dashes,3)
 (alice,369)
 (therefore,1)
 (rate,1)
 (kettles,1)
 (hay,3)
 (_picked_,1)
```

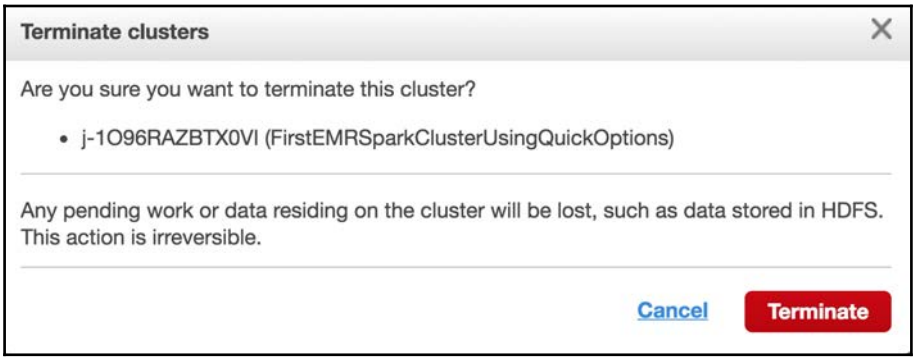
22. Next, we sort the list and show the top `10` words and their count:

```
scala> val sorted_counts = counts.collect().sortBy(wc => -wc._2)

scala> sorted_counts.take(10).foreach(println)
 (,4969)
 (the,834)
 (and,459)
 (you,441)
 (to,398)
 (a,386)
 (of,381)
 (alice,369)
 (i,328)
 (it,236)
```

23. We can store the sorted results in an S3 bucket as shown here:

```
scala>
sc.parallelize(sorted_counts).saveAsTextFile("s3://aurobindo-
book/wordcount-alice-in-wonderland")
```

24. Finally, we go back to the EMR console and terminate the cluster (by clicking on the Terminate button).



25. You should see a terminating message temporarily on the console. After a few minutes, the cluster is terminated:

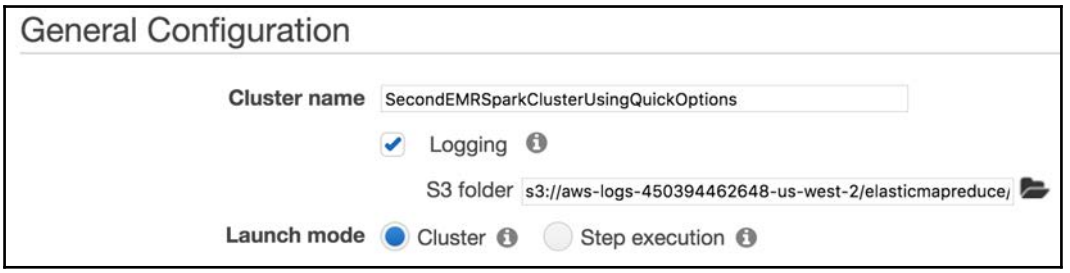


In our second example, we create an EMR cluster as we did in the first example. We want to get comfortable spinning up cluster, perform the computations required, and terminate the cluster immediately after.

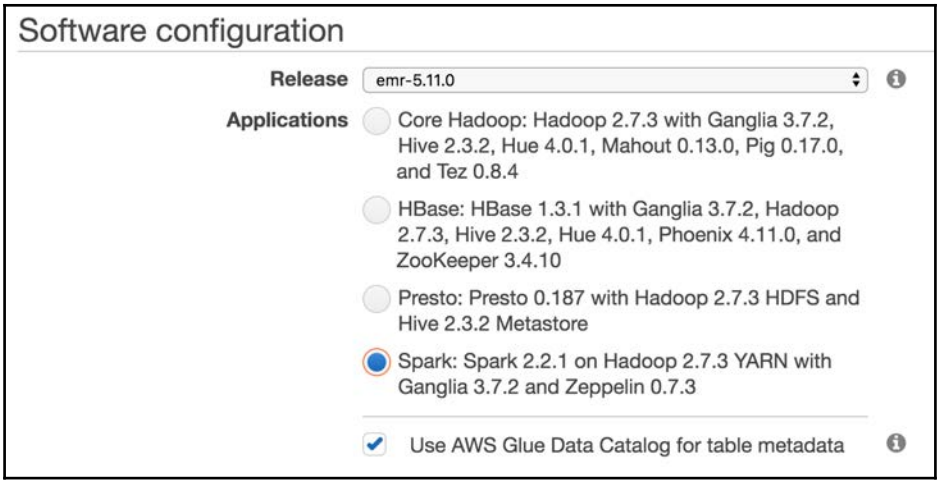
1. Use your favorite editor to create a Spark Python program (containing the following code) and name it `wordcount.py`. Upload the program to a S3 bucket:

```
from __future__ import print_function
from pyspark import SparkContext
import sys

if __name__ == "__main__":
   sc = SparkContext(appName="WordCount")
   text_file = sc.textFile(sys.argv[1])

# Compute the word count
counts = text_file.flatMap(lambda line: line.split(" ")).map(lambda
word: (word, 1)).reduceByKey(lambda a, b: a + b)

#Save the results to S3
   counts.saveAsTextFile(sys.argv[2])

   sc.stop()
```

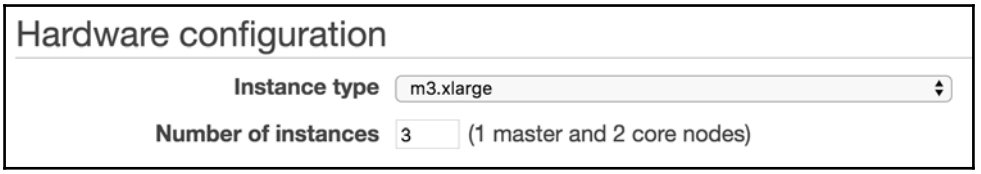
2. From the EMR console, create a new EMR cluster and name it `MSecondEMRSparkClusterUsingQuickOptions`:



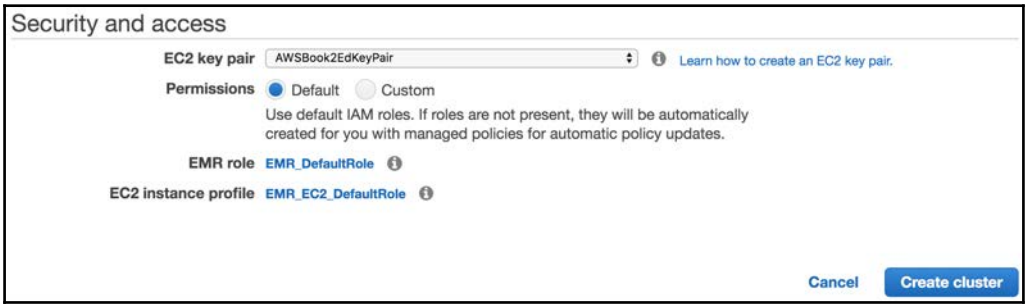
Choose Spark in the Software configuration section choose Application options as shown here:



Choose the default selections for Hardware configuration as shown:



Select the EC2 key pair as shown:



3. List all the EMR clusters using AWS CLI command. Note the ID value for the cluster:

```
Aurobindos-MacBook-Pro-2:Downloads aurobindosarkar$ aws emr list-
clusters --active

{    "Clusters": [
        {
            "Status": {
                "Timeline": {
                    "ReadyDateTime": 1517146171.077,
                    "CreationDateTime": 1517145843.445
                },
                "State": "WAITING",
                "StateChangeReason": {
                    "Message": "Cluster ready after last step
failed."
                }
            },
            "NormalizedInstanceHours": 96,
            "Id": "j-2996LIDHV5QLJ",
            "Name": "SecondEMRSparkClusterUsingQuickOptions"
        }
    ]
}
```

3. After your cluster is up and running, click on the **Add step** button (from the Steps tab):

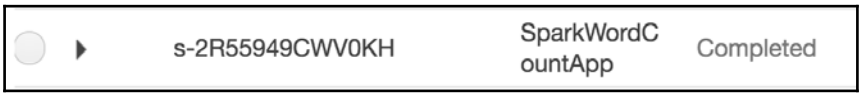


4. Specify the parameters in the **Add step** screen to include the step type to be Spark application, give a name for the application, choose **Cluster** as the **Deploy mode** parameter, specify the location of the `wordcount.py` program on S3 (s3://aurobindo-book/wordcount.py) , and specify the input and output S3 buckets as arguments to the program as shown here. Then, click on the **Add** button:

```
aws emr add-steps --cluster-id j-2996LIDHV5QLJ --steps
Type=spark,Name=SparkWordCountApp,Args=[--deploy-mode,cluster,--
master,yarn,--
conf,spark.yarn.submit.waitAppCompletion=false,s3://aurobindo-
book/wordcount.py,s3://aurobindo-
```

```
book/aliceinwonderland.txt,s3://aurobindo-
book/pywordcount],ActionOnFailure=CONTINUE
{
    "StepIds": [
        "s-2R55949CWV0KH"
    ]
}
```

5. You should see the newly created step in the **Steps** tab on the **Clusters** console. The Status field will change from Pending to Running to Completed:



6. Go to the S3 console to check the contents of the output. Click on the **pywordcount** bucket (output S3 bucket), download one of the files and view the contents downloaded file:

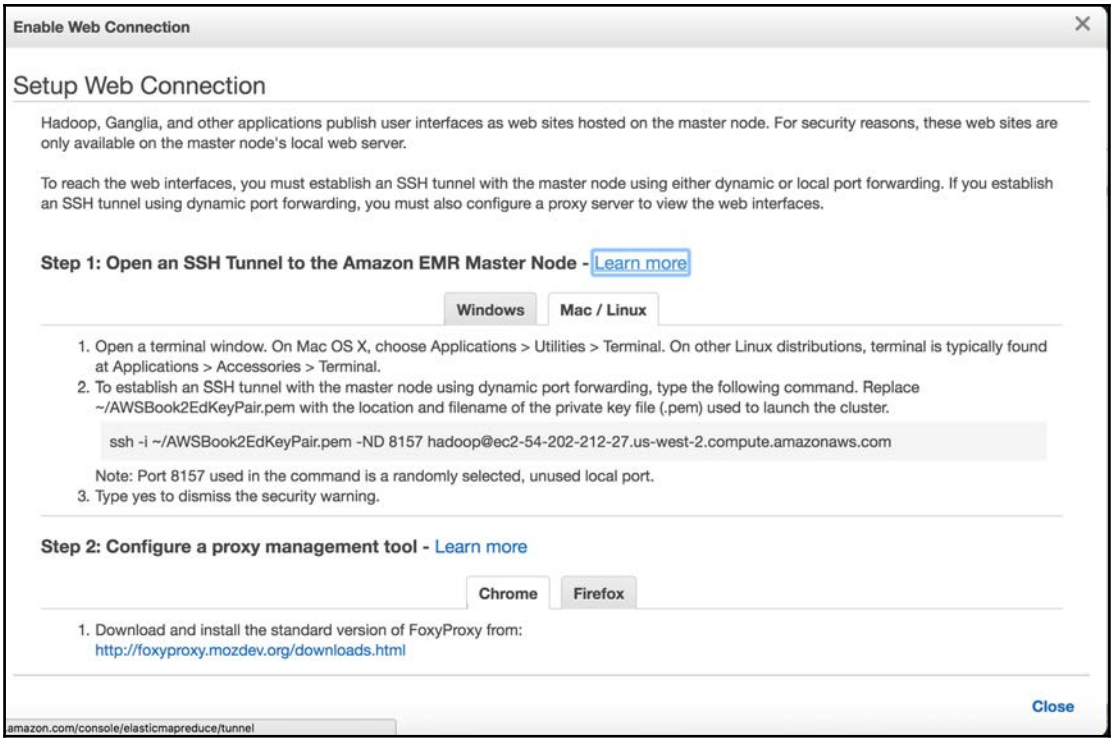
```
(,4969)
(the,834)
(and,459)
(you,441)
(to,398)
(a,386)
(of,381)
(alice,369)
(i,328)
(it,236)
(in,217)
(_],208)
(queen,164)
(with,142)
(is,120)
(that,118)
...
(there,30)
(think,30)
(up,30)
(said,29)
(did,29)
```

```
(down,28)
(gutenberg,28)
(been,28)
(never,28)
(time,28)
(way,28)
```

7. Apache Spark provides browser-based UIs for its jobs. In order to enable the Spark UI, click on the **Enable Web Connection** link (in the Summary tab) as shown here:



8. You should see the following screen with instructions of enabling Spark UI.





For more details on enabling Spark Web UIs, refer to the AWS documentation: `https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-spark-history.html`

Summary

In this chapter, we shifted our focus to designing big data applications on AWS. We explored the best practices for using EMR, designing streaming applications with Amazon Kinesis, serverless applications, machine learning, and predictive analytics applications using Amazon SageMaker.

In the next chapter, we will focus on the hands-on implementation of big data applications using AWS services such as Glue, Lambda, Kinesis, and SageMaker.

9
Implementing a Big Data Application

In this chapter, we will focus on the hands-on implementation of big data applications using AWS services. More specifically, we will implement typical use cases for ETL, serverless computing, streaming data, and machine learning using AWS services such as Kinesis, EMR, Apache Spark, SageMaker, and Glue.

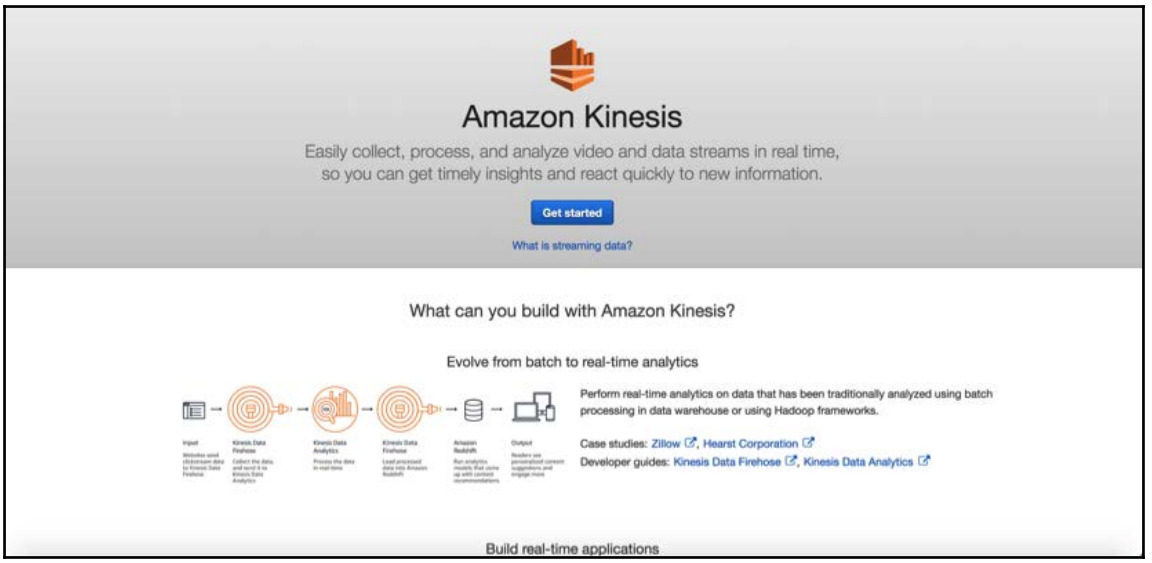
In this chapter, we will cover the following:

- Setting up an Amazon Kinesis Stream
- Creating an AWS Lambda function
- Using Amazon Kinesis Firehose
- Using AWS Glue and Amazon Athena
- Using Amazon SageMaker

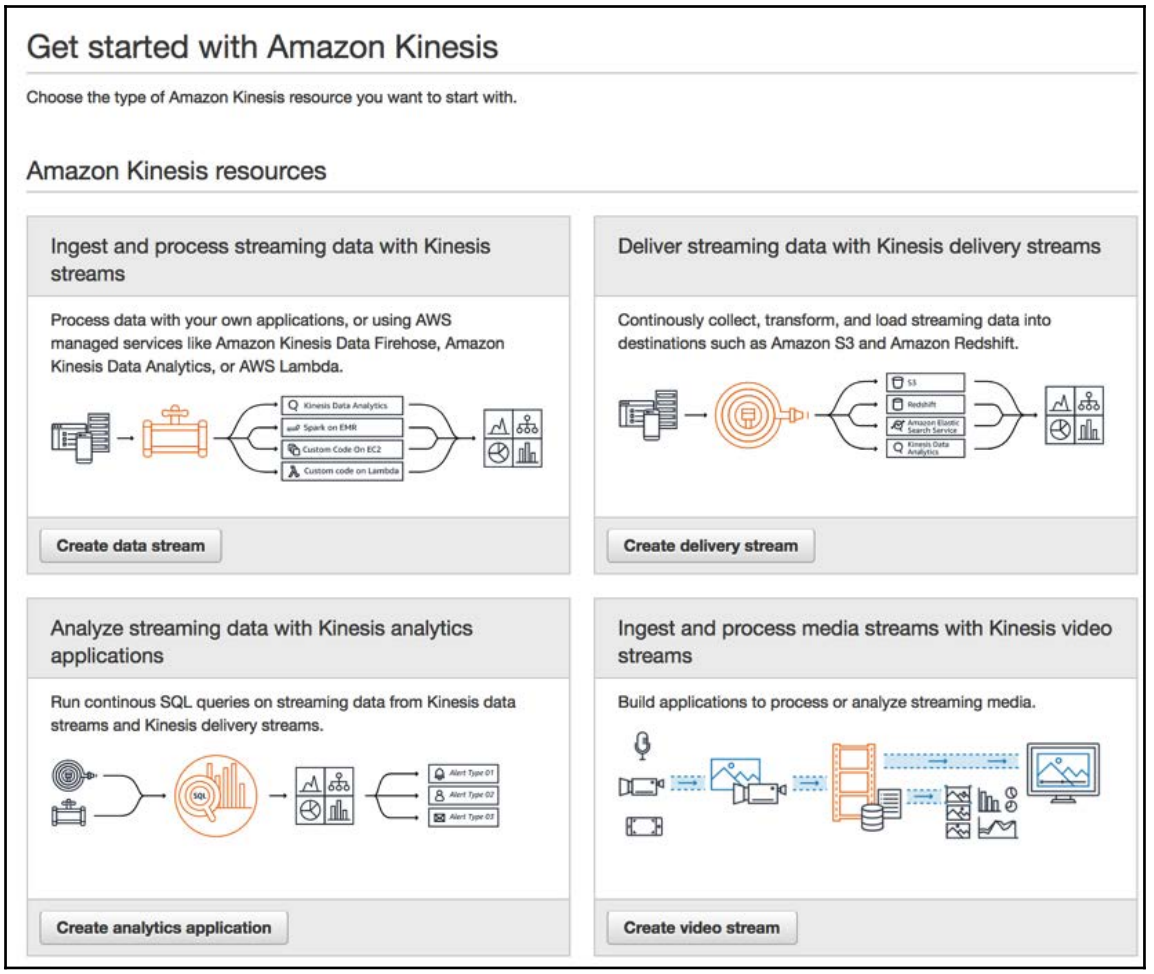
Setting up an Amazon Kinesis Stream

In this section, we will implement a Kinesis Stream and familiarize ourselves with a few useful Kinesis CLI commands.

1. Log into the AWS Management Console and go to **Amazon Kinesis Console**. Click on the **Get started** button:



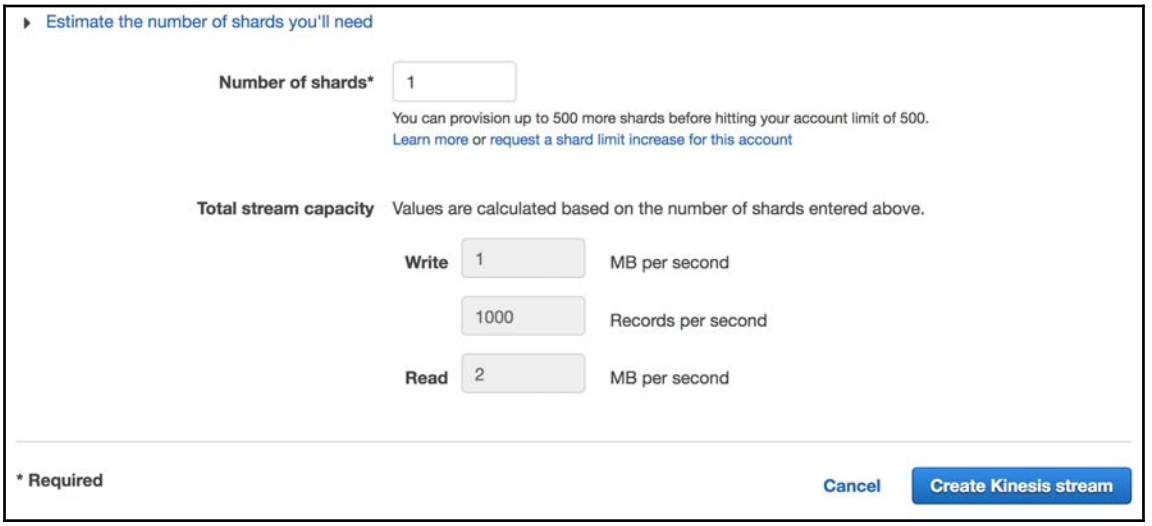
2. Click on the **Create data stream** button on the **Get started with Amazon Kinesis** screen:



3. Specify a name for the Kinesis stream as `KinesisTestStream`:



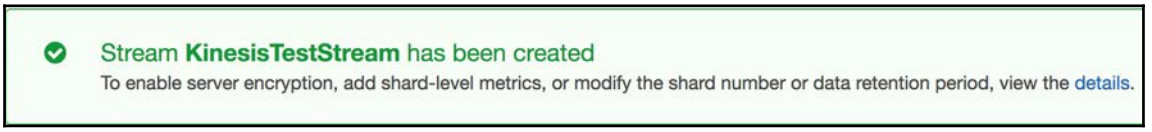
Specify the **Number of shards** as `1`. Click on the **Create Kinesis stream** button:



4. You should temporarily see the **Status** as **CREATING**, as shown in the following screenshot:



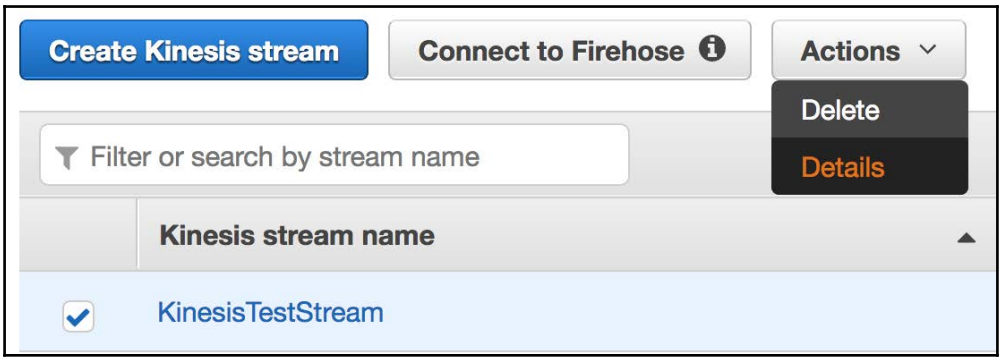
5. After the stream is created, you should see a success message, as shown in the following screenshot:



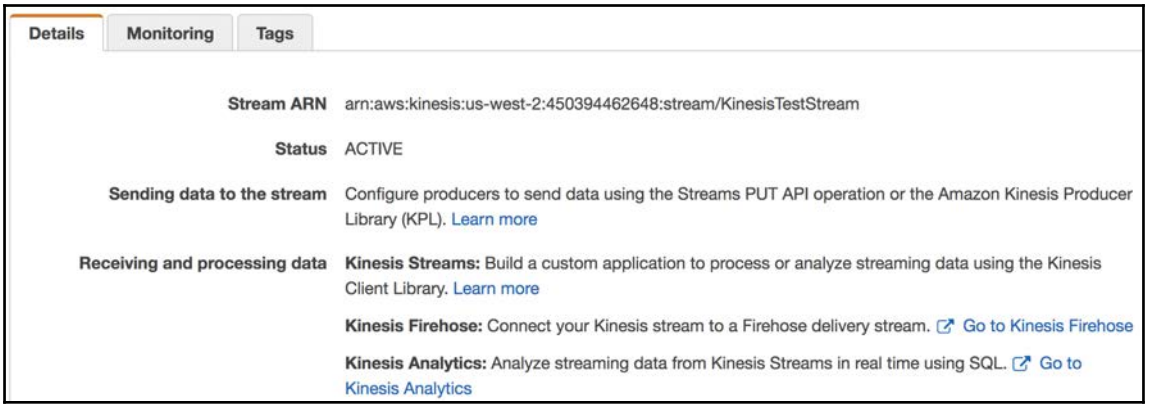
The Status of the Kinesis stream will change to ACTIVE.



6. Select the Kinesis stream and then click on **Details** from the **Actions** menu:



7. View the details of the Kinesis stream you just created in the Details tab:



8. Execute the describe-stream command to display the details of the Kinesis stream:

```
Aurobindos-MacBook-Pro-2:Downloads aurobindosarkar$ aws kinesis
describe-stream --stream-name KinesisTestStream
```

9. You can list all your Kinesis streams with the `list-streams` command:

```
Aurobindos-MacBook-Pro-2:Downloads aurobindosarkar$ aws kinesis
list-streams
```

10. Insert a test record with the `put-record` command:

```
Aurobindos-MacBook-Pro-2:Downloads aurobindosarkar$ aws kinesis
put-record --stream-name KinesisTestStream --partition-key 123 --
data "Test Event Record"
```

11. Use the `get-shard-iterator` to get the Shard Iterator and copy its value:

```
Aurobindos-MacBook-Pro-2:Downloads aurobindosarkar$ aws kinesis
get-shard-iterator --shard-id shardId-000000000000 --shard-
iterator-type TRIM_HORIZON --stream-name KinesisTestStream
{
    "ShardIterator":
"AAAAAAAAAAGBmIC9k7uah4A0bb0oCW2QfniIesq8snvGJjvlu6D6/UyejLbcjUZK/h
zRmSnJtTlQpVpInUoOvacIavV0f4/7X4UE9637qgQi9lc5UqrzXfXKs1Yl4H1Cmp97L
D1si3LqqClCsWebTGW8+YUlWFXKfm0kle270c/gcFUqHsxlqyV1XH7XnsY7JVF+0opx
vWNygX0Gce0ld4xmCYJ4RPNX"
}
```

12. Use the `get-records` command (with the shard iterator value) to display the records:

```
Aurobindos-MacBook-Pro-2:Downloads aurobindosarkar$ aws kinesis
get-records --shard-iterator
AAAAAAAAAAFn2BntwTpAyLoy9V1e1pQ9OiPz44MAMq+fXKPvG/S0eLoKaeuy/NDWq40
7234Oc1UgtS5rNOjwN+Zz0YSLDKgSmG/muEwi9ZAA+b91WxjZGatpMTHmizMIdYHkWv
LvltzFDbCd/ezoOeE3t2kWWFz0ospxacohJxWicFOnuTrAXymSpaUdWCL5junBCwQ78
h4t5v3E38P+zbYbh9OYgy4o
```

Creating an AWS Lambda function

In this section, we will create a Lambda function to process Kinesis events.



For a more detailed coverage of this topic, refer to the blog by *Sunil Dalal* at: `https://www.polyglotdeveloper.com/lambda/2017-07-05-Using-Lambda-as-Kinesis-events-processor/`. Refer to *Creating a .jar Deployment Package Using Maven and Eclipse IDE (Java)* for more details: `https://docs.aws.amazon.com/lambda/latest/dg/java-create-jar-pkg-maven-and-eclipse.html`.

1. Create a maven project. Create an `example` package and key in the following Java code in a class (`ProcessKinesisEvents`):

```
package example;
```

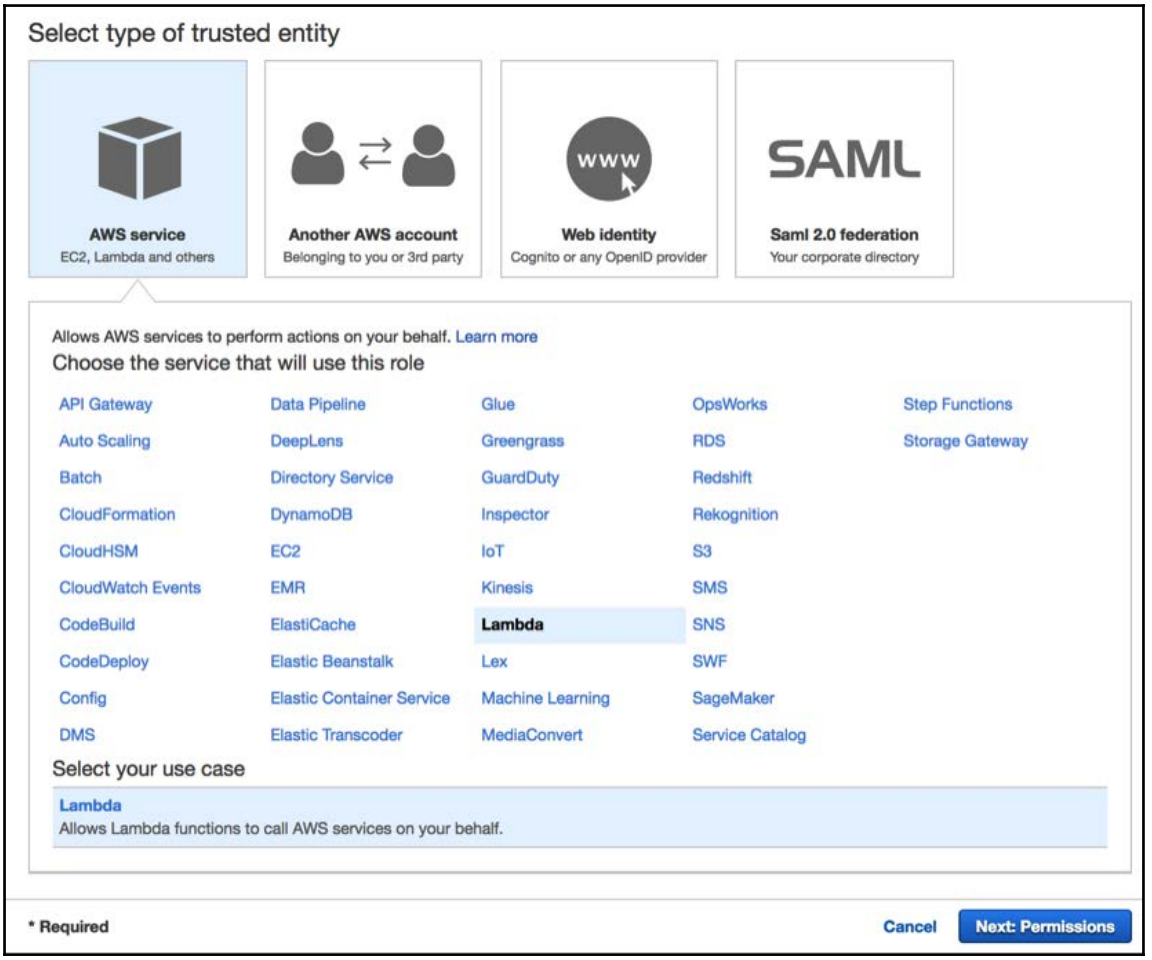
```
import java.io.IOException;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;
import
com.amazonaws.services.lambda.runtime.events.KinesisEvent.KinesisEv
entRecord;
public class ProcessKinesisEvents {
    public String handleRequest(KinesisEvent event) throws
IOException {
       System.out.println("Record Size - " +
event.getRecords().size());
       for(KinesisEventRecord rec : event.getRecords()) {
         System.out.println(new
String(rec.getKinesis().getSequenceNumber()));
         System.out.println(new
String(rec.getKinesis().getData().array()));
       }
       return "success";
    }
}
```

2. The `pom.xml` file is listed as follows for reference:

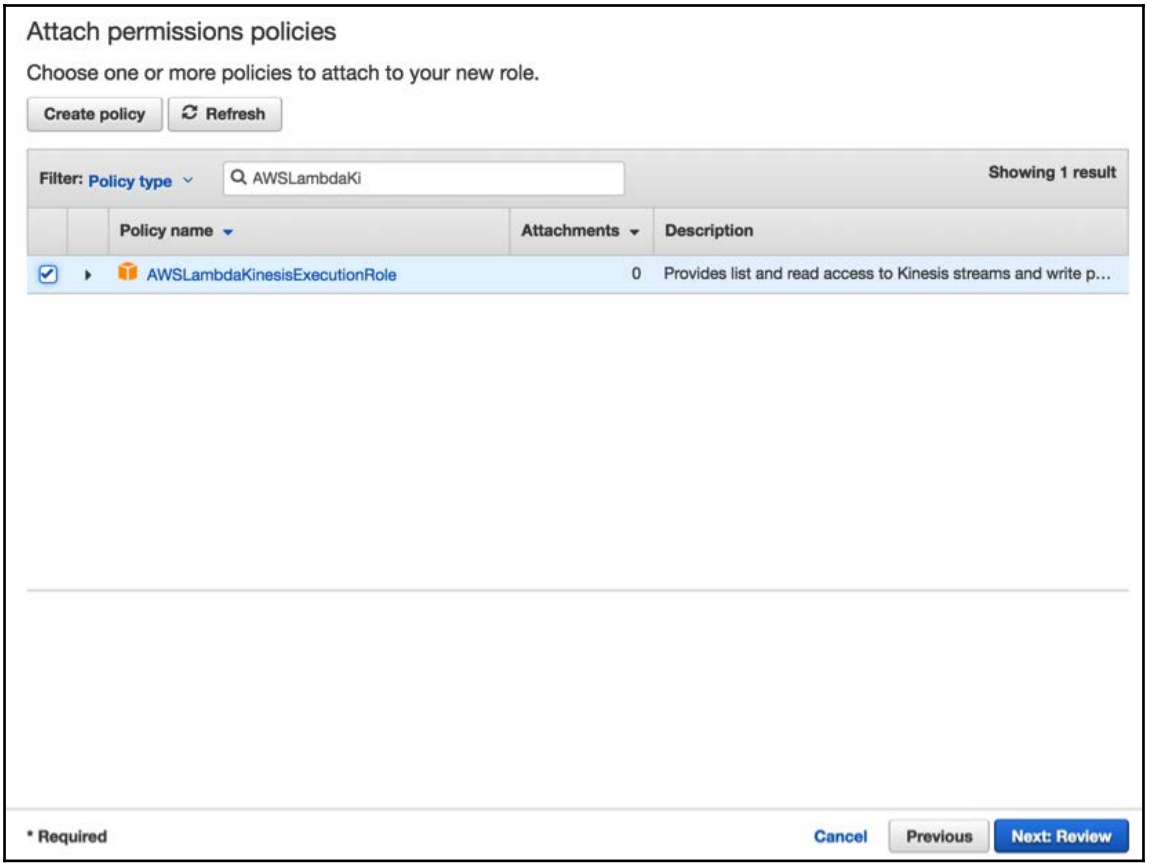
```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>doc-examples</groupId>
<artifactId>lambda-java-example</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>lambda-java-example</name>
<description>AWS Lambda Java Example</description>
<dependencies>
  <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-core</artifactId>
      <version>1.1.0</version>
  </dependency>
  <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-events</artifactId>
      <version>1.3.0</version>
  </dependency>
</dependencies>
<build>
  <plugins>
      <plugin>
```

```
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-shade-plugin</artifactId>
            <version>2.3</version>
        </plugin>
    </plugins>
</build>
</project>
```

3. Select **Lambda** from the IAM console:



4. Search for **AWSLambdaKinesisExecutionRole** and attach the permissions policy to the AWS Lambda role:



5. Use the `create-function` command as shown to create the Lambda function:

```
aws lambda create-function \
>  --region us-west-2 \
>  --function-name ProcessKinesisEvents \
>  --zip-file
fileb:///Users/aurobindosarkar/Downloads/IWebApps/workspace/lambda-
java-example/target/lambda-java-example-0.0.1-SNAPSHOT.jar \
>  --role arn:aws:iam::450394462648:role/lambda-kinesis-execution-
role \
>  --handler example.ProcessKinesisEvents::handleRequest \
>  --runtime java8
{
    "TracingConfig": {
        "Mode": "PassThrough"
    },
```

```
    "CodeSha256": "jQQ9jo18W57ngH7HFDDv4S0DF9EYNFkYs2NoUQBFz1Q=",
    "FunctionName": "ProcessKinesisEvents",
    "CodeSize": 7763705,
    "MemorySize": 128,
    "FunctionArn": "arn:aws:lambda:us-
west-2:450394462648:function:ProcessKinesisEvents",
    "Version": "$LATEST",
    "Role": "arn:aws:iam::450394462648:role/lambda-kinesis-
execution-role",
    "Timeout": 3,
    "LastModified": "2018-01-28T18:52:25.963+0000",
    "Handler": "example.ProcessKinesisEvents::handleRequest",
    "Runtime": "java8",
    "Description": ""
}
```

The newly created function will be listed in the Lambda **Functions** console:

	Function name	Description	Runtime	Code size
○	ProcessKinesisEvents		Java 8	7.4 MB

6. Create an input file (`input.txt`) with the following records to test the Lambda function:

```
{
   "Records": [
       {
           "kinesis": {
               "partitionKey": "partitionKey-3",
               "kinesisSchemaVersion": "1.0",
               "data": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0IDEyMy4=",
               "sequenceNumber":
"49545115243490985018280067714973144582180062593244200961"
           },
           "eventSource": "aws:kinesis",
           "eventID":
"shardId-000000000000:49545115243490985018280067714973144582180062593244200961",
           "invokeIdentityArn": "arn:aws:iam::account-
id:role/testLEBRole",
           "eventVersion": "1.0",
           "eventName": "aws:kinesis:record",
           "eventSourceARN": "arn:aws:kinesis:us-
west-2:35667example:stream/examplestream",
```

```
            "awsRegion": "us-west-2"
        }
    ]
}
```

7. Invoke the Lambda function. Specify the `invocation-type` as `Event`, function name as `ProcessKinesisEvents`, `region`, `input`, and `output` files as shown:

```
Aurobindos-MacBook-Pro-2:Downloads aurobindosarkar$ aws lambda
invoke \
> --invocation-type Event \
> --function-name ProcessKinesisEvents \
> --region us-west-2 \
> --payload file:///Users/aurobindosarkar/Downloads/input.txt \
> outputfile.txt
```

8. For a synchronous response, specify `invocation-type` as `RequestResponse` in the preceding command.
9. Add an event source in AWS Lambda so that your Lambda function can start polling the Amazon Kinesis stream:

```
Aurobindos-MacBook-Pro-2:Downloads aurobindosarkar$ aws lambda
create-event-source-mapping \
> --region us-west-2 \
> --function-name ProcessKinesisEvents \
> --event-source arn:aws:kinesis:us-
west-2:450394462648:stream/KinesisTestStream \
> --batch-size 100 \
> --starting-position TRIM_HORIZON
```

10. Using the following AWS CLI command, add event records to your Amazon Kinesis stream. You can run the same command more than once to add multiple records to the stream:

```
Aurobindos-MacBook-Pro-2:Downloads aurobindosarkar$ aws kinesis
put-record \
> --stream-name KinesisTestStream \
> --data "Test Record 1" \
> --partition-key shardId-000000000000 \
> --region us-west-2
```

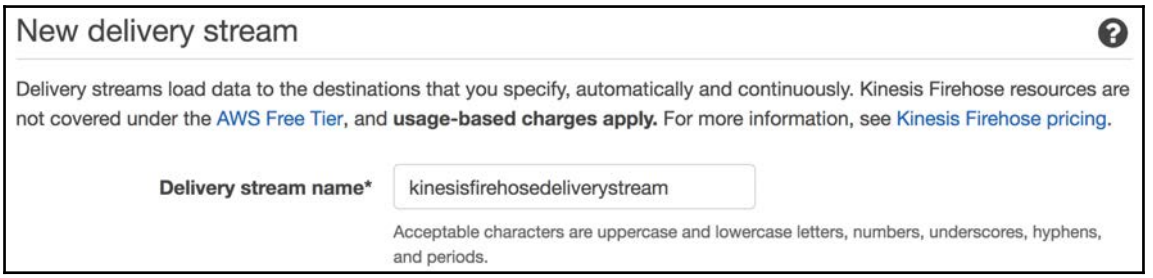
11. Validate that the Lambda function and process the records by going to CloudWatch logs:

```
Aurobindos-MacBook-Pro-2:Downloads aurobindosarkar$ aws logs
describe-log-streams --log-group-name
'/aws/lambda/ProcessKinesisEvents' --region us-west-2 --order-by
LastEventTime --descending
```

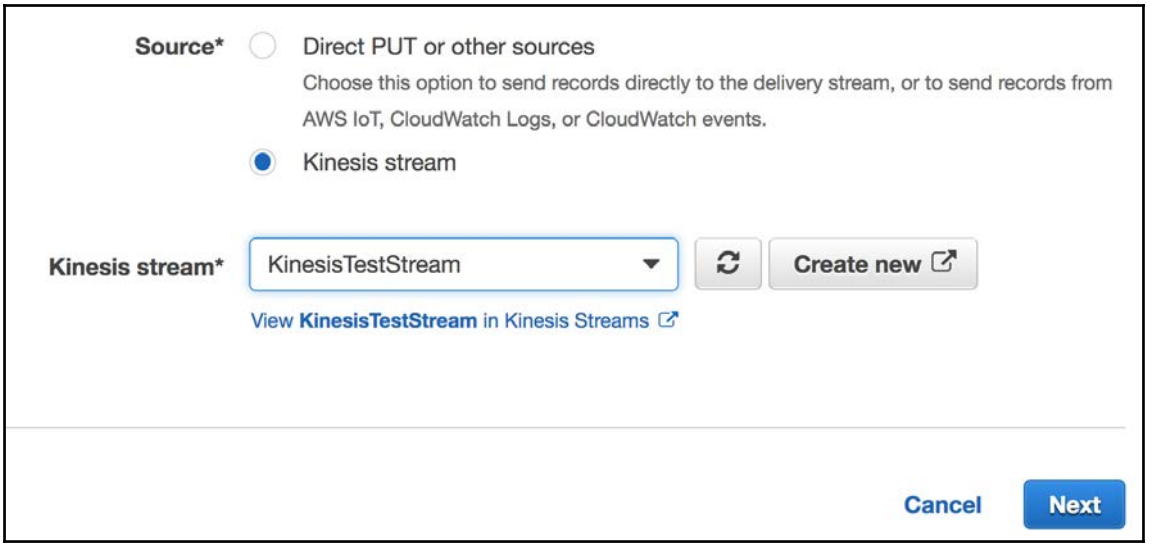
Using Amazon Kinesis Firehose

In this section, we will implement a delivery stream using Kinesis Firehose. The application reads data from a Kinesis Stream and stores it in a S3 bucket.

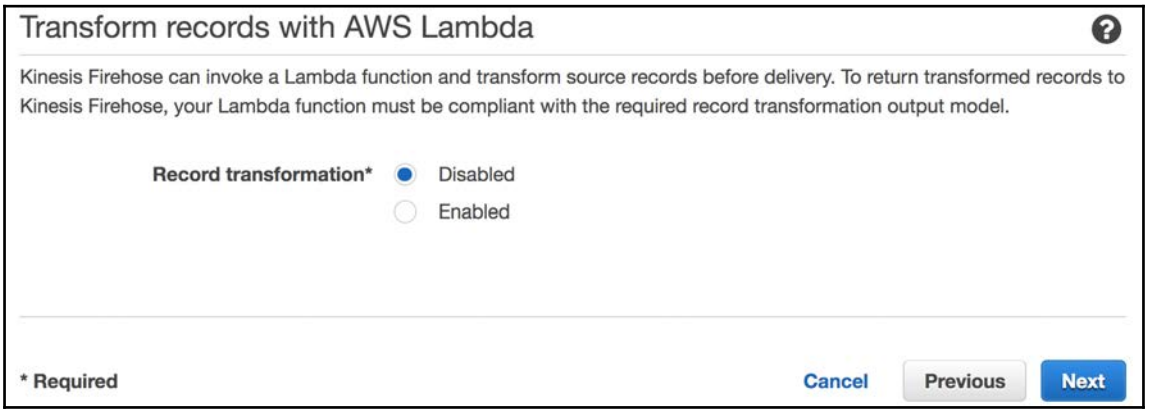
1. Log in to the Management Console and go to the **Kinesis Firehose** console. Click on the **Create delivery stream** button. Specify a name (`kinesisfirehosedeliverystream`) for the **Delivery stream** name:



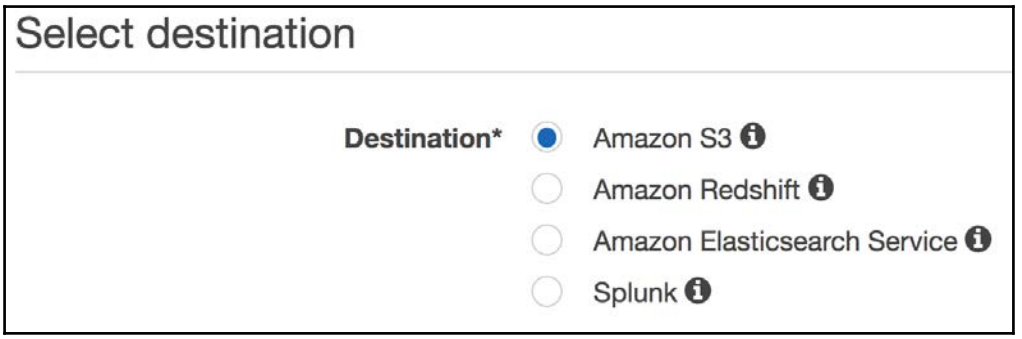
2. Choose the source as Kinesis stream and select the Kinesis stream we created in a previous section (`KinesisTestStream`) from the dropdown list:



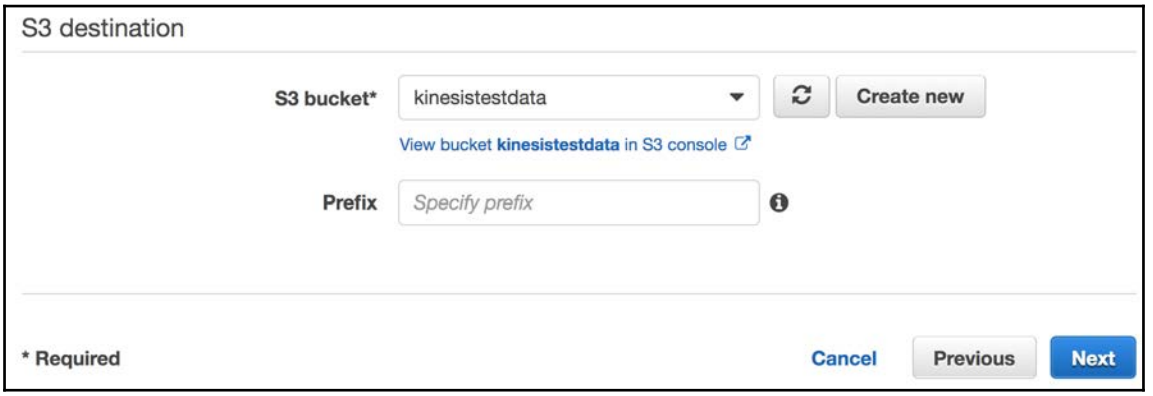
2. To keep our example simple, we will not do any Lambda-based processing of the incoming data. So select the **Disabled** option for the **Record transformation** parameter. Click on the **Next** button:



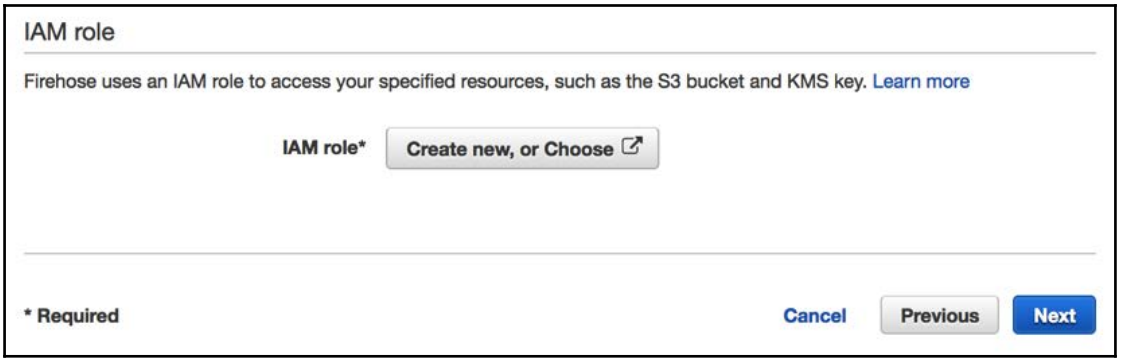
3. Create an Amazon S3 bucket called `kinesistestdata` from the S3 console. Select **Amazon S3** as the **Destination** for streaming data to be stored by Kinesis Firehose:



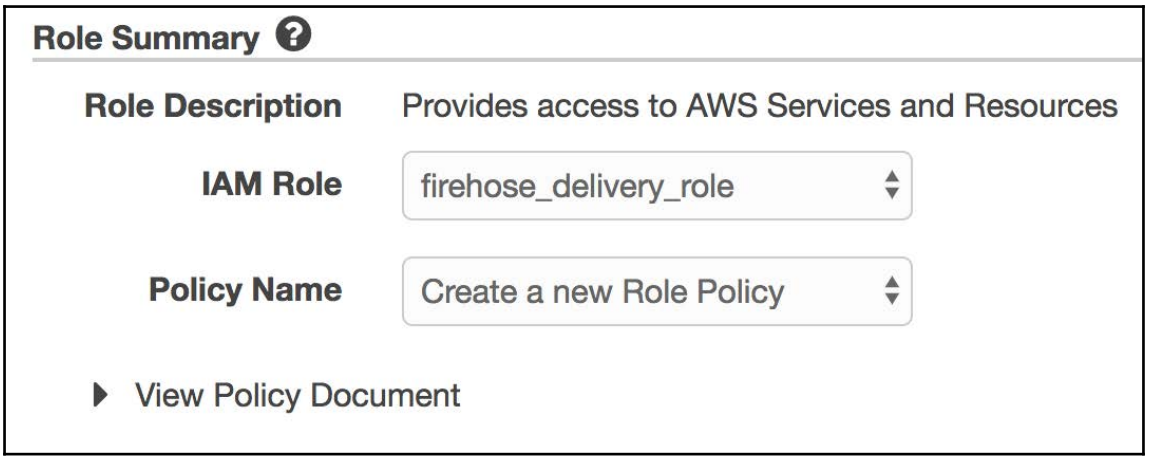
4. Select `kinesistestdata` from the S3 bucket dropdown list. Click on the **Next** button:



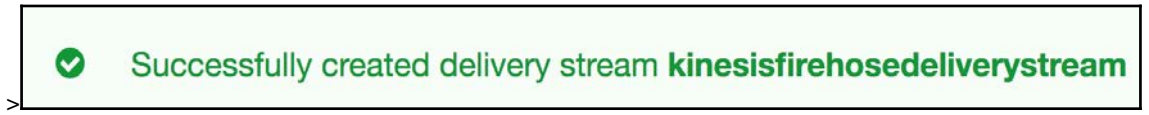
5. Click on the **Create new or Choose** button:



6. Select **firehose_delivery-role. Click on the Allow button**:



7. Click on the **Next** button.
8. Review all the details of the delivery stream and then click on the **Create delivery stream** button:
9. You should see the success message as shown in the following screenshot:



>

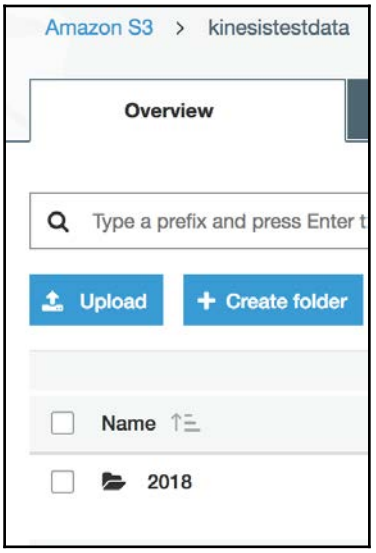
The newly created Kinesis Firehose delivery stream should be listed:

	Name	Status	Created	Source
○	kinesisfirehosedeliverystream	Active	2018-01-29T00:54+0530	KinesisTestStr...

10. Insert a test record into the Kinesis stream `KinesisTestStream` with the `put-record` command:

```
Aurobindos-MacBook-Pro-2:Downloads aurobindosarkar$ aws kinesis
put-record --stream-name KinesisTestStream --partition-key 123 --
data "Test Event Record"
```

11. Go to the S3 console and you should see your test record data in the destination bucket (**kinesistestdata**):

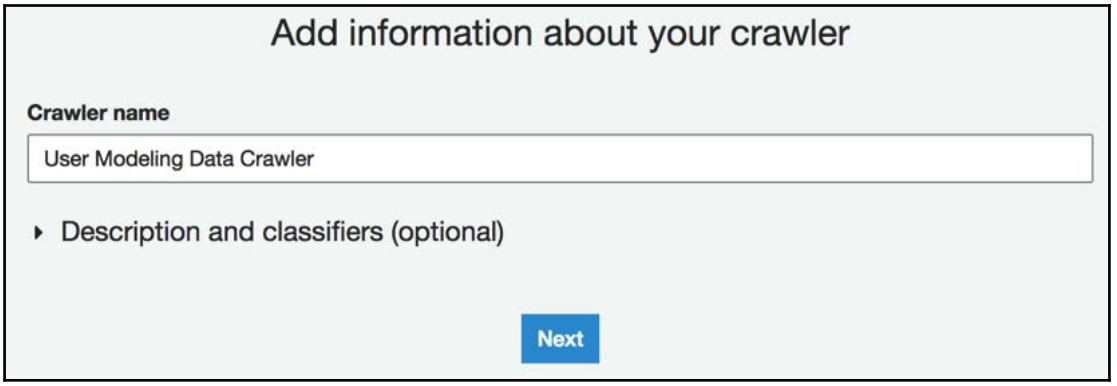


Using AWS Glue and Amazon Athena

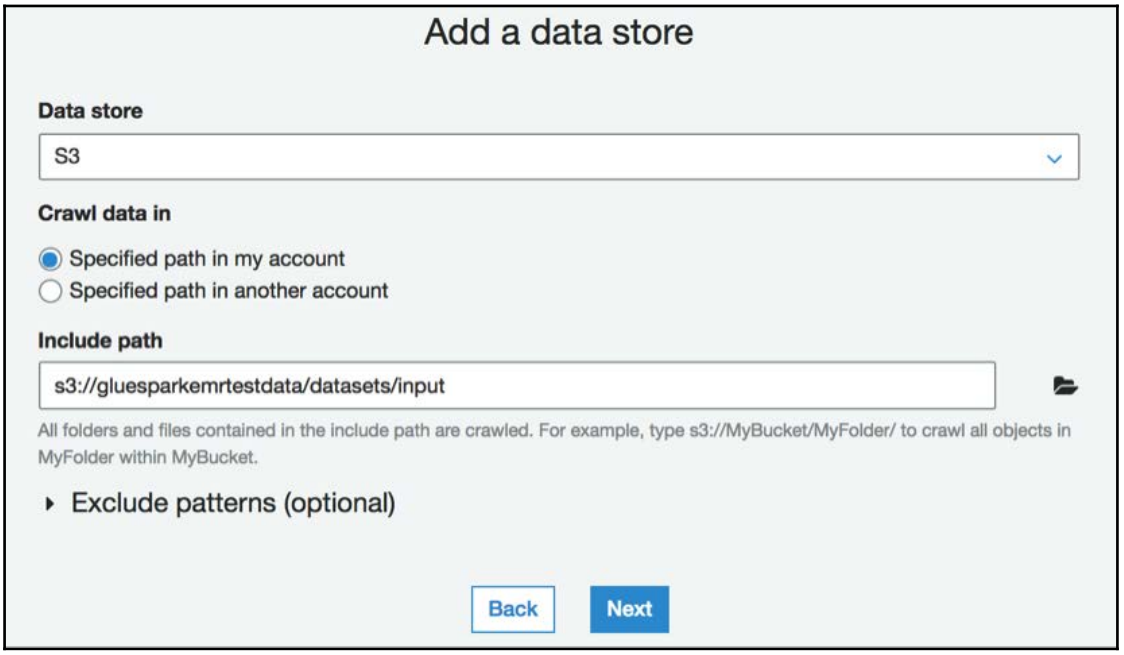
In this section, we will use AWS Glue to create a crawler, an ETL job, and a job that runs KMeans clustering algorithm on the input data.

We use a publicly available dataset about the students' knowledge status on a subject. The dataset and the field descriptions are available for download from the UCI site: `https://archive.ics.uci.edu/ml/datasets/User+Knowledge+Modeling`

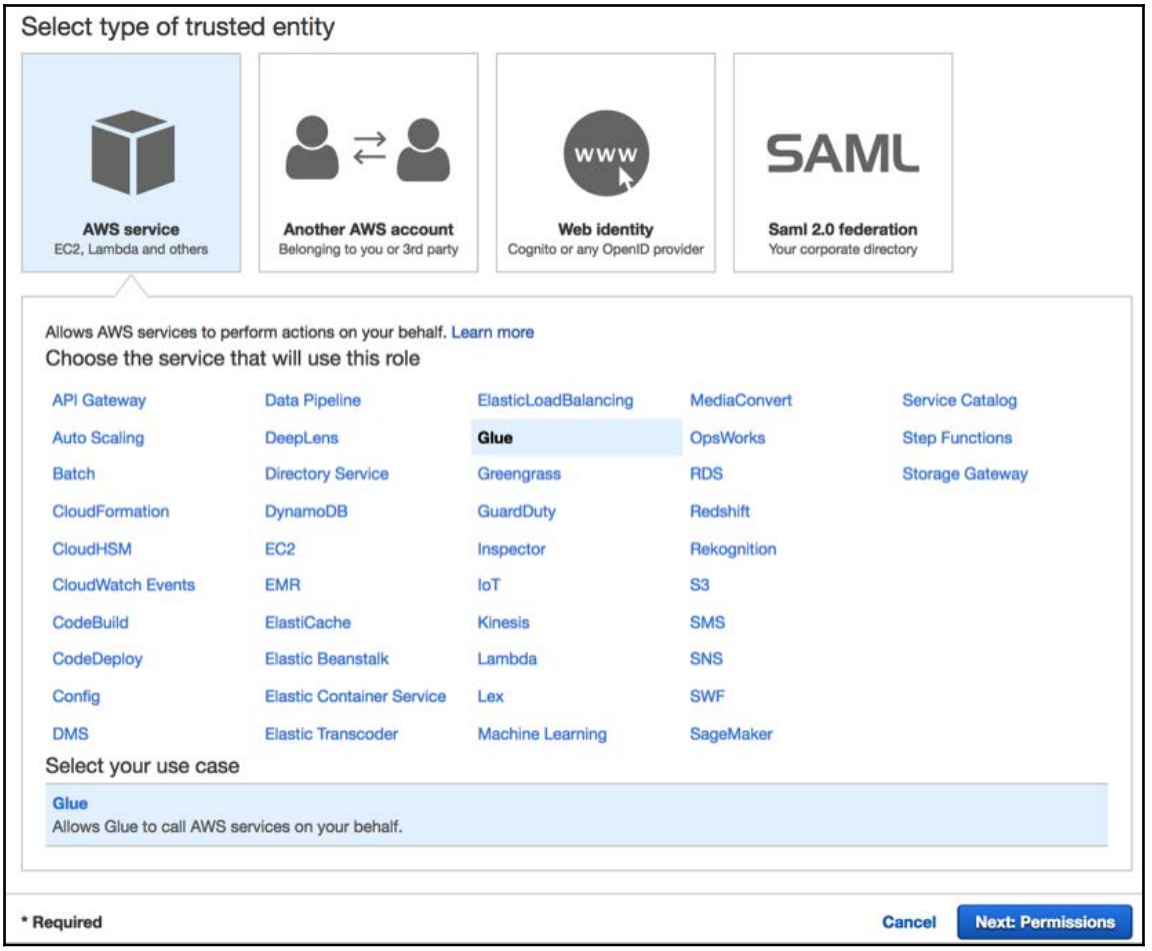
1. Log in to the AWS Management Console and go to the Glue console. Click on the **Add crawler** button.
2. Specify the **Crawler name** as `User Modeling Data Crawler` as shown here. Click on the **Next** button:



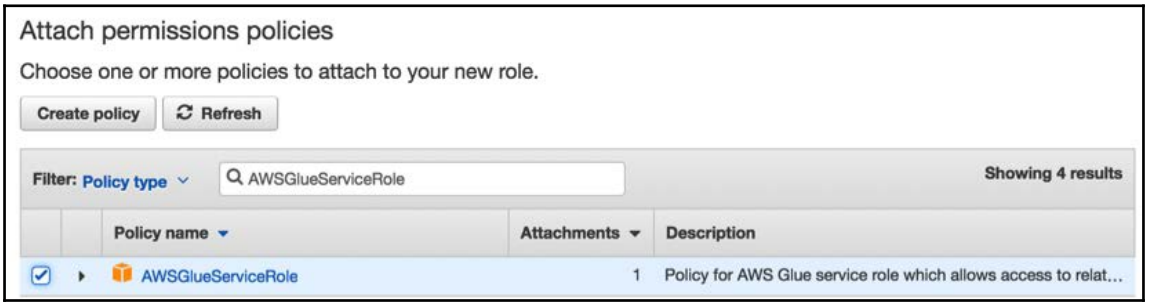
3. In the **Add a data store** screen, select **S3** as the **Data store**, and select the **Specified path in my account** option. Specify the path for the S3 bucket containing the input data. Click on the **Next** button:



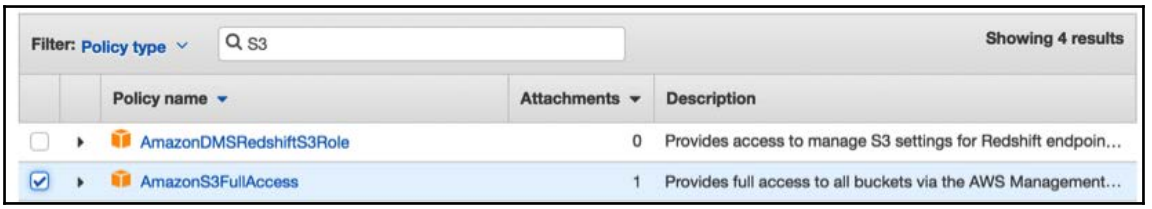
4. Select **No** on the **Add another data store** and click on the **Next** button.
5. On the IAM console, select the **Glue service** and click on the **Next: Permissions** button:



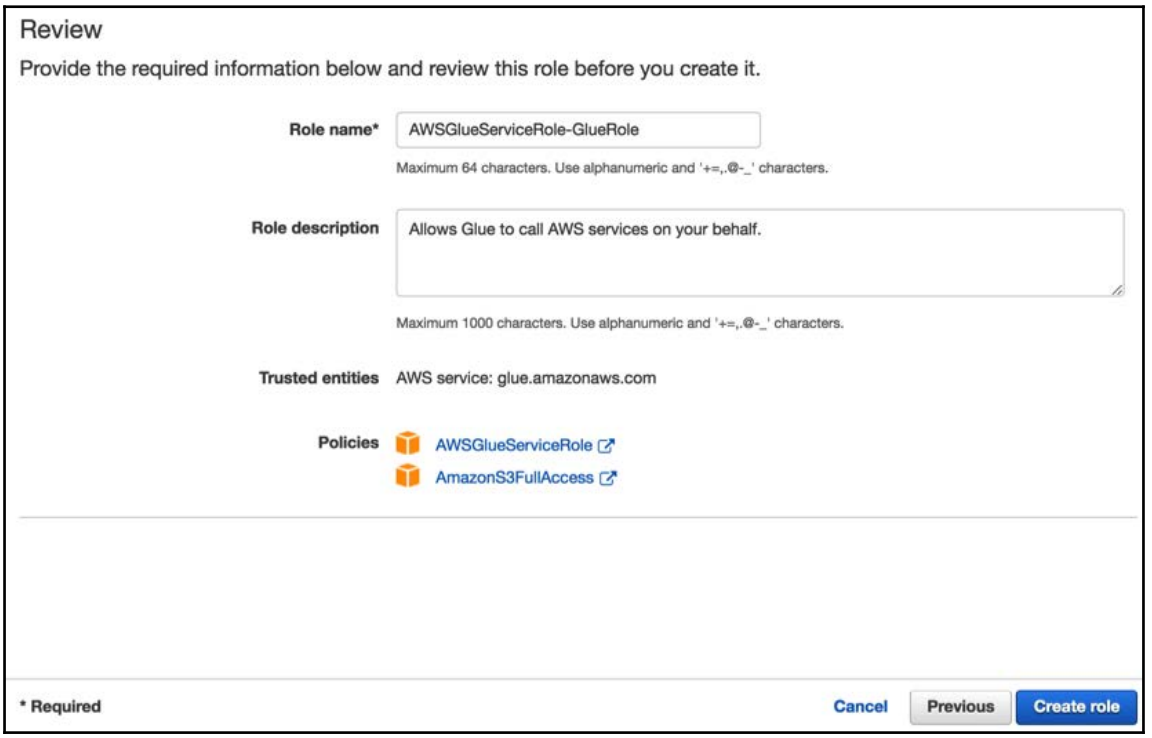
6. Next, we attach the appropriate policies to the role. Filter for **AWSGlueServiceRole** and select it:



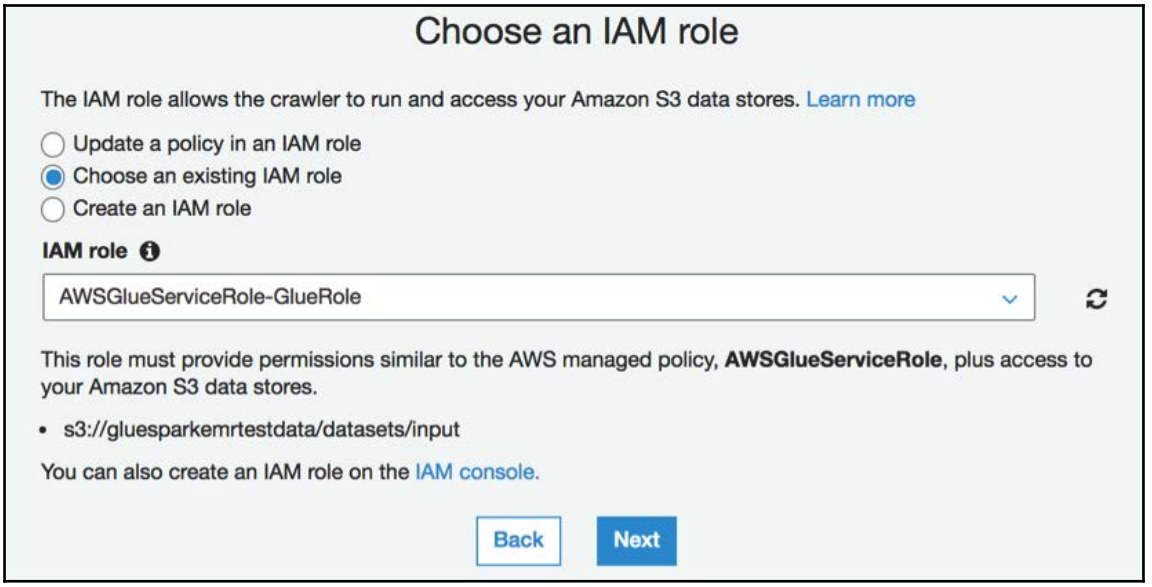
7. Filter for **AmazonS3FullAccess** policy and select it:



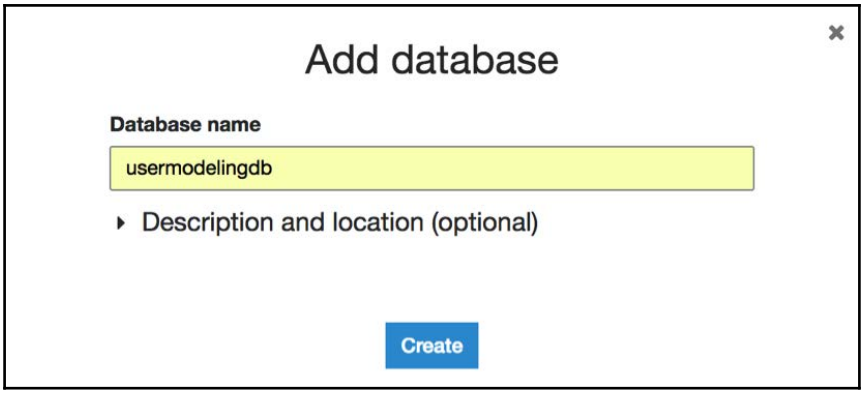
8. Review the role for the Glue service and click on the **Create role** button:



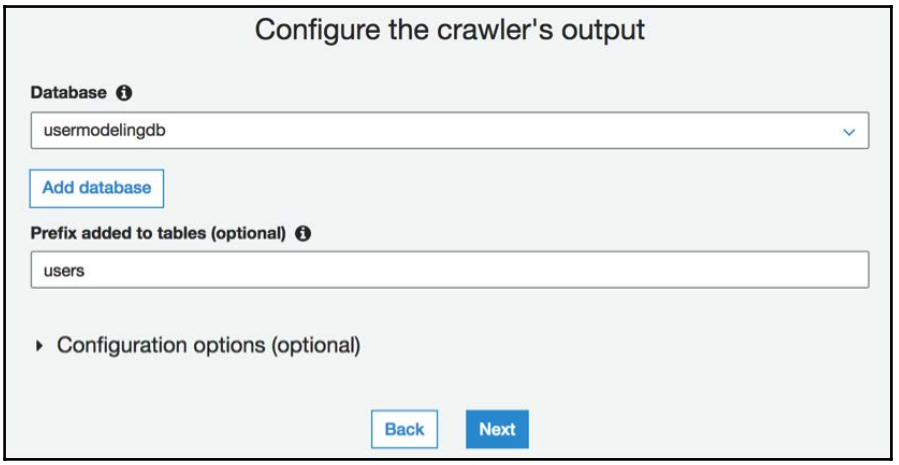
9. Return to the crawler wizard and select the newly created IAM role. Click on the **Next** button:



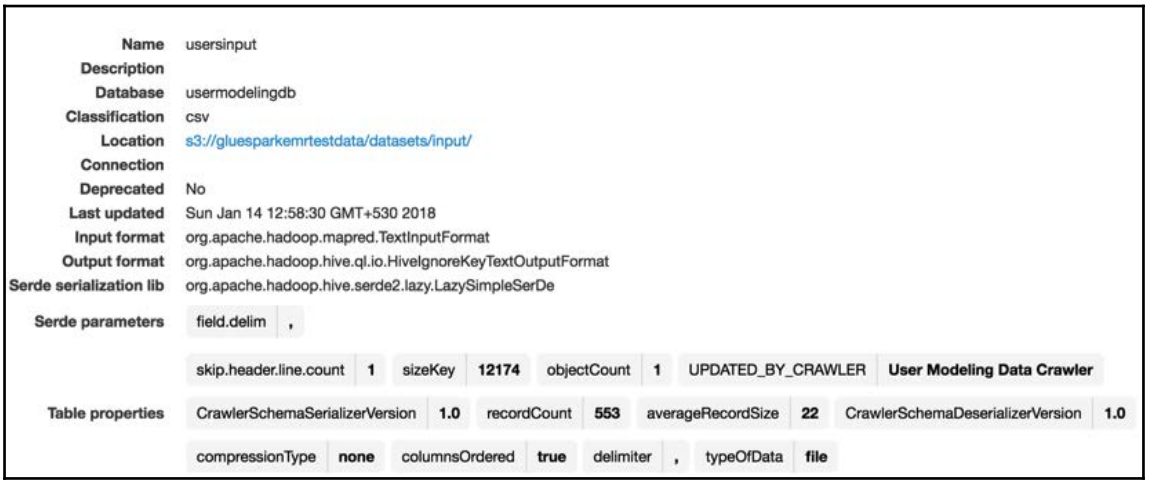
12. Select the **Frequency parameter** as **Run on demand** and click on the **Next** button.
13. Next, we configure a database for the crawler's output. As we do not have a database configured, click on the **Add database** button.
14. Specify the name of the database as `usermodelingdb` and click on the **Create** button:



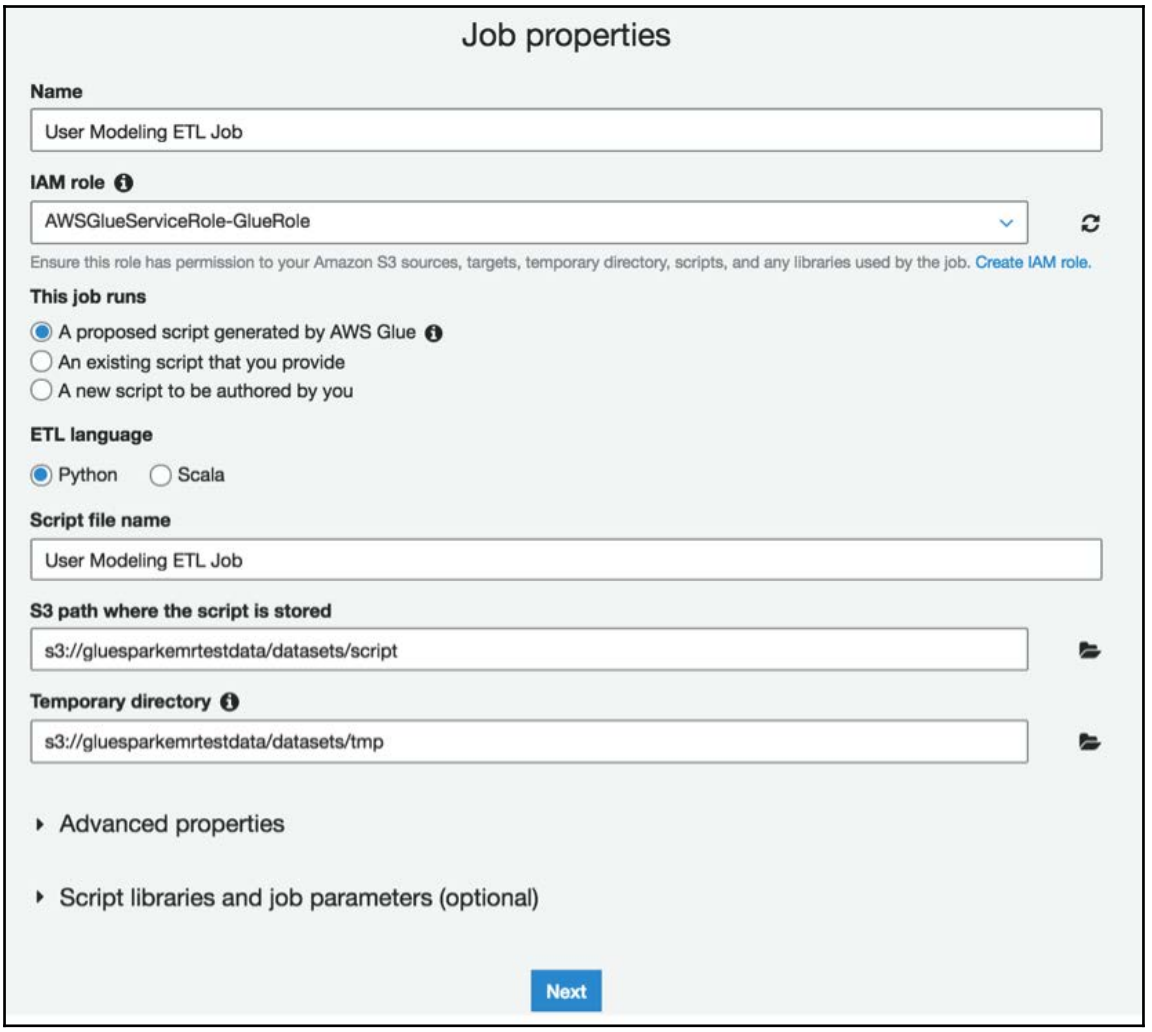
13. Specify a prefix for the tables (as users) and click on the **Next** button:



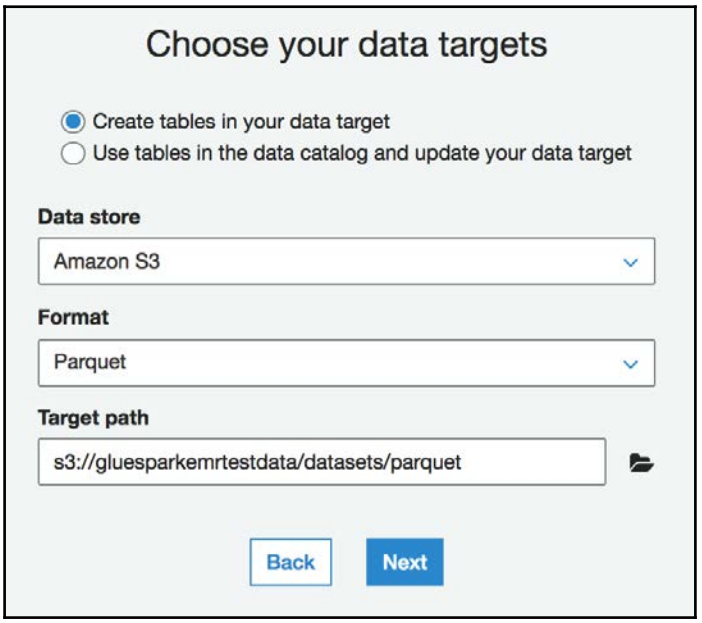
14. Review all the details for the crawler and click on the **Finish** button
15. After the crawler is created, you should see the message shown here. Click on the hyperlink **Run it now?** to execute the crawler:
16. After the crawler finishes running, click on the **Tables** link to see the tables listed. Click on the newly created table (usersinput). You should see the following details. Notice that the header line is skipped and 553 records from our input file are inserted. Additionally, verify that the inferred schema is accurate:



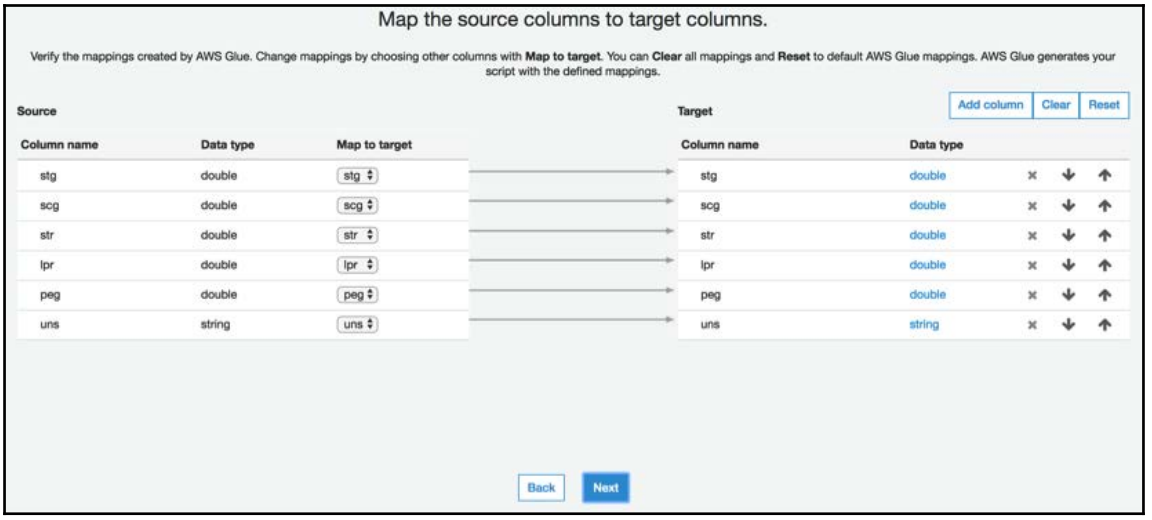
19. Next, we create an ETL job using Glue. Click on the **Jobs** link and then click on the **Add job** button.
20. Next, we specify the properties of the ETL job, including the **Name** (`User Modeling ETL Job`), **IAM role** (select the **AWSGlueServiceRole-GlueRole** we created earlier), select the option for using the proposed script generated by AWS Glue, select the language as **Python**, specify a name for the script file, and S3 locations for the script and temporary directory (create these folders from the S3 console). Click on the **Next** button:



22. Choose **usersinput** table as the data source and click on the **Next** button.
23. Select **Amazon S3** as the **Data store**, **Format** as **Parquet**, and the S3 **Target path** (create the S3 folder from the S3 console). Click on the **Next** button:

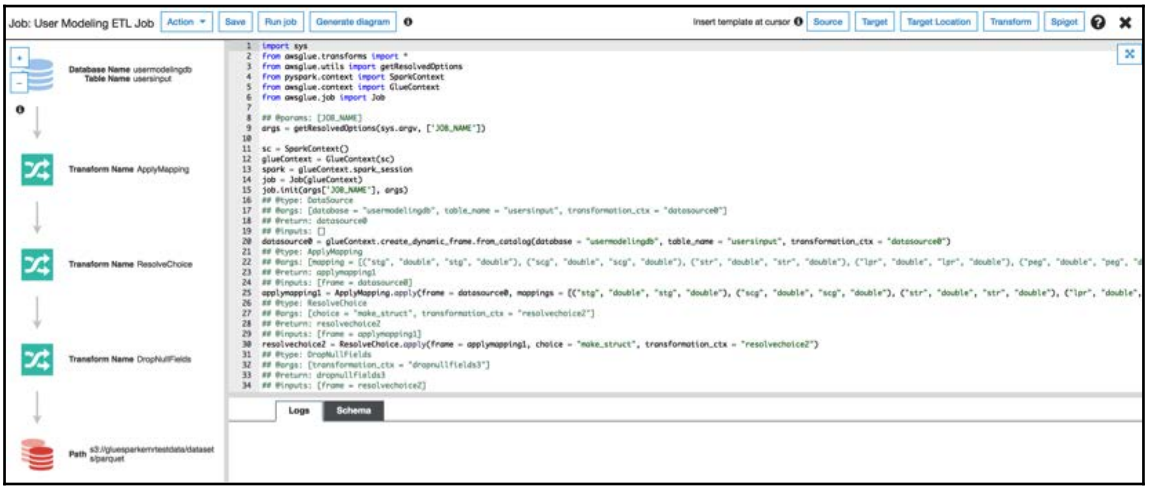


24. On this screen, you can map the source columns to the target columns. Additionally, you can drop columns in the target, if they are not relevant to your analysis. We will leave the mapping as-is as we want all of the input data in the target parquet file. Click on the **Next** button:

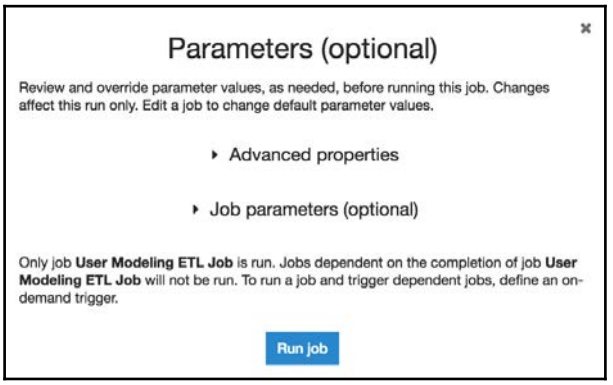


25. Review the **Job properties** and click on the **Finish** button.

26. You should see the code generated by Glue. In the left pane, there is a figure that shows you the flow visually. We can make changes to the generated script, if required. Click on the **Run job** button in the top menu:



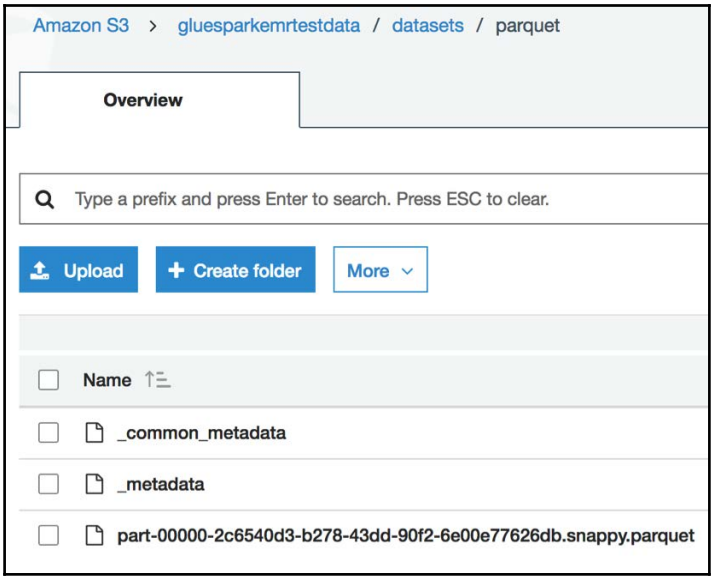
27. Click on the **Run job** button. We do not have any further parameters to specify for our example:



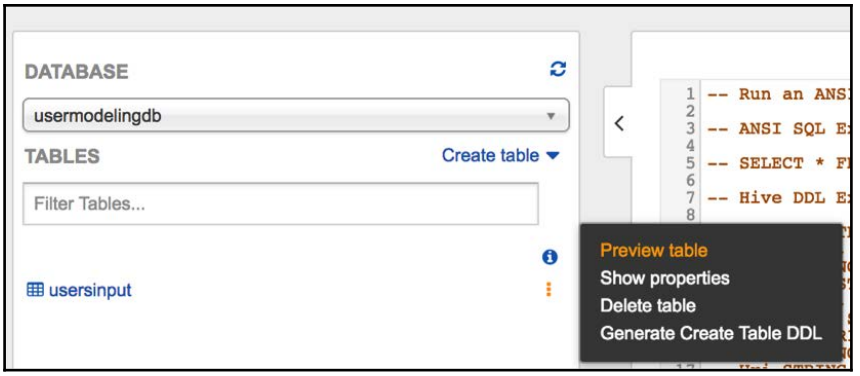
28. You should see the **Run job** button disabled as the job has started:



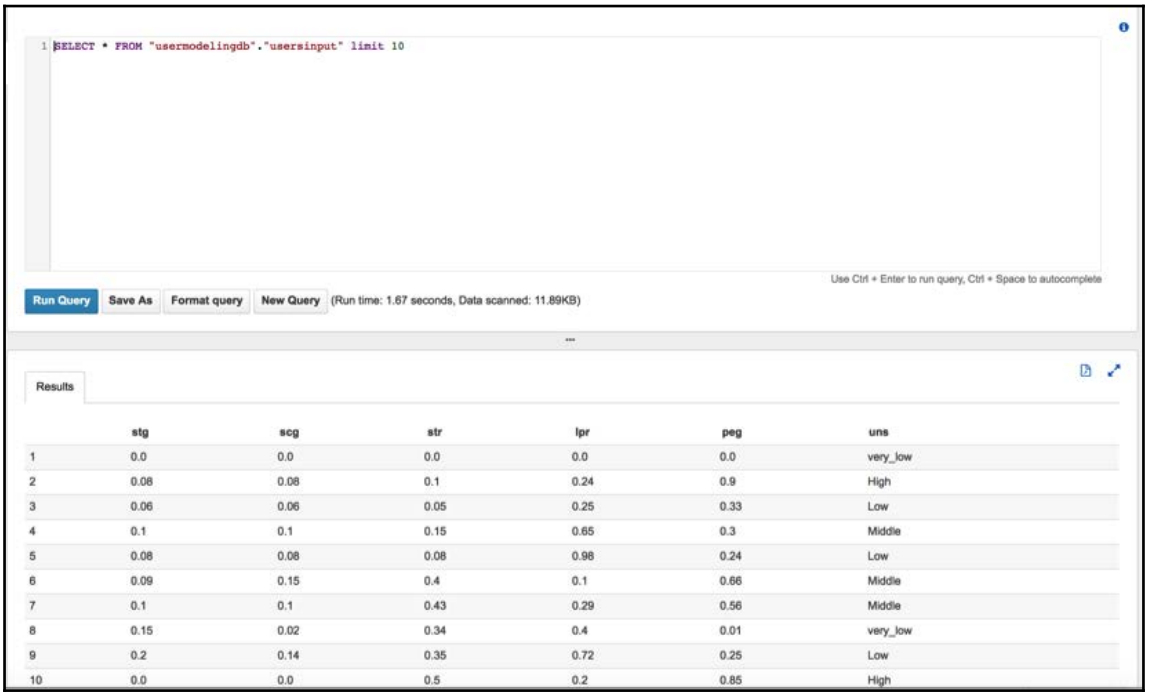
29. Switch to the **Jobs** console to see your submitted job listed. Select the job to expand the display. You should see the **Run status** as **Running.** Upon completion of the job, the status changes to **Succeeded.**
30. Go to the S3 console to confirm that the parquet files have been created in the target folder:



33. Next, we use Athena to check the contents of the source table. Start the Athena console and click on the **Query Editor** link.
34. Select the database and click on the three dots next to the source table to display the menu. Click on the **Preview table** option:



35. The SQL for querying contents of the table is executed and the rows are displayed as shown in the following screenshot:

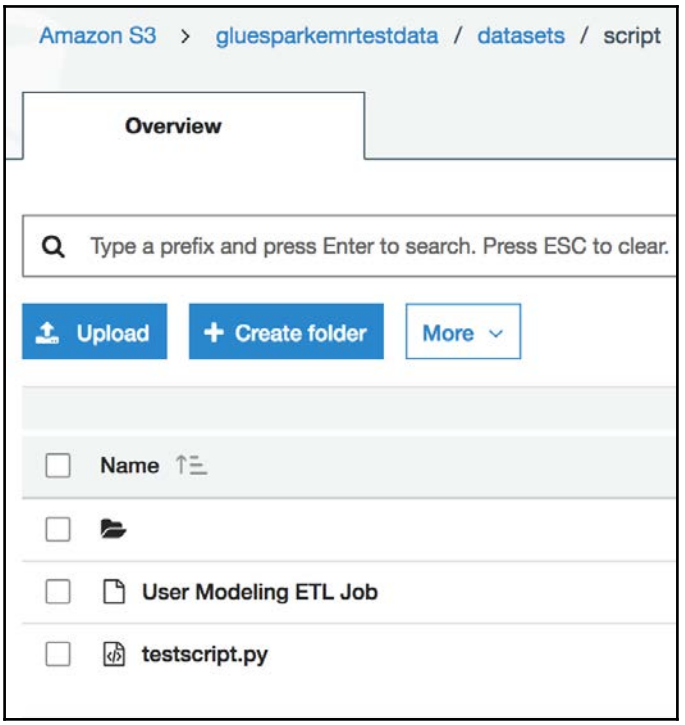


	stg	scg	str	lpr	peg	uns
1	0.0	0.0	0.0	0.0	0.0	very_low
2	0.08	0.08	0.1	0.24	0.9	High
3	0.06	0.06	0.05	0.25	0.33	Low
4	0.1	0.1	0.15	0.65	0.3	Middle
5	0.08	0.08	0.08	0.98	0.24	Low
6	0.09	0.15	0.4	0.1	0.66	Middle
7	0.1	0.1	0.43	0.29	0.56	Middle
8	0.15	0.02	0.34	0.4	0.01	very_low
9	0.2	0.14	0.35	0.72	0.25	Low
10	0.0	0.0	0.5	0.2	0.85	High

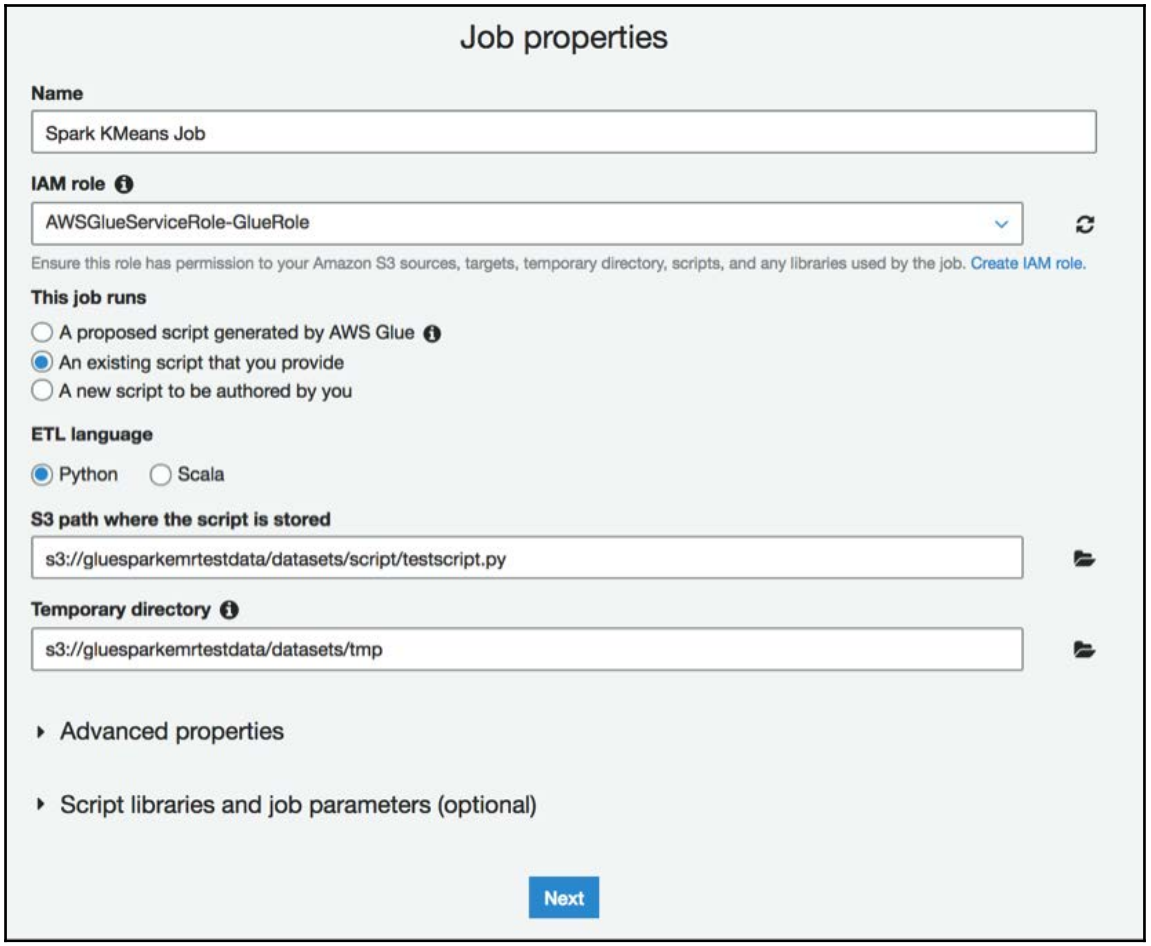
36. Create a Python-Spark-Glue script file (`testscript.py`) containing the following code:

```
import sys
from awsglue.utils import getResolvedOptions
from awsglue.context import GlueContext
from awsglue.job import Job
from awsglue.transforms import SelectFields
from awsglue.dynamicframe import DynamicFrame
from pyspark.context import SparkContext
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.clustering import KMeans

args = getResolvedOptions(sys.argv, ['JOB_NAME'])
destination = "s3://gluesparkemrtestdata/datasets/parquet/results/"
namespace = "usermodelingdb"
tablename = "usersinput"

sc = SparkContext()
glueContext = GlueContext(sc)
job = Job(glueContext)
job.init(args['JOB_NAME'], args)

#Load table and select fields
datasource0 =
glueContext.create_dynamic_frame.from_catalog(name_space =
namespace, table_name = tablename)
SelectFields0 = SelectFields.apply(frame = datasource0,
paths=["stg","scg","str","lpr","peg","uns" ])
DataFrame0 = DynamicFrame.toDF(SelectFields0)

#Select features and convert to Spark ML required format
features = ["stg","scg","str","lpr","peg"]
assembler =
VectorAssembler(inputCols=features,outputCol='features')
assembled_df = assembler.transform(DataFrame0)

#Fit and Run Kmeans
kmeans = KMeans(k=2, seed=1)
model = kmeans.fit(assembled_df)
transformed = model.transform(assembled_df)

#Save data to destination
transformed.write.mode('overwrite').parquet(destination)
job.commit()
```

37. Upload the script file to the S3 script folder using the S3 console:

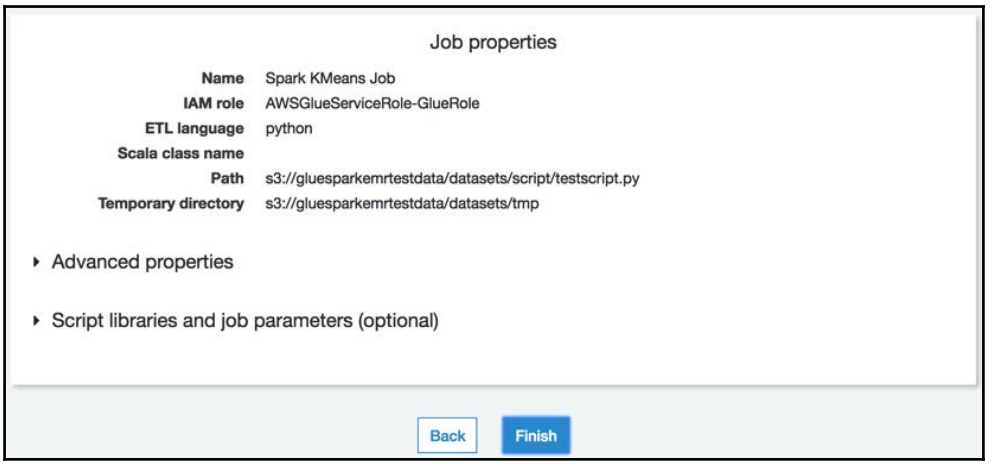


38. Create a new job and specify the **Name** as `Spark KMeans Job`. This time select the option **An existing script that you provide**. Furthermore, select **Python** as the language, and specify the S3 paths for the `testscript.py` file and the temporary directory. Click on **Next**:

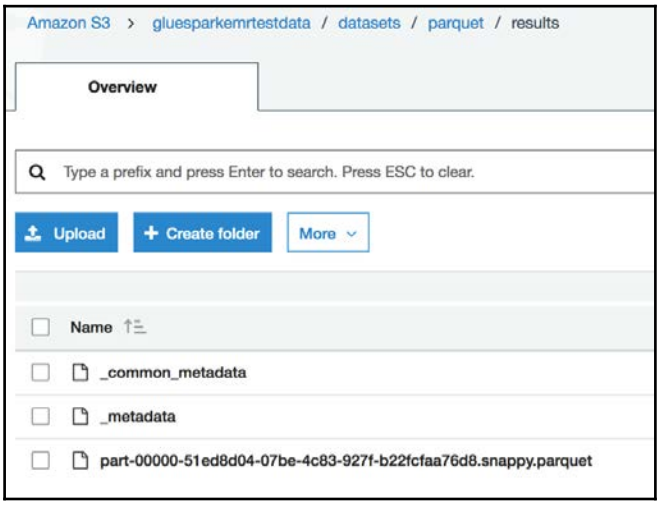


39. Click on the **Next** button on the **Connections** screen.

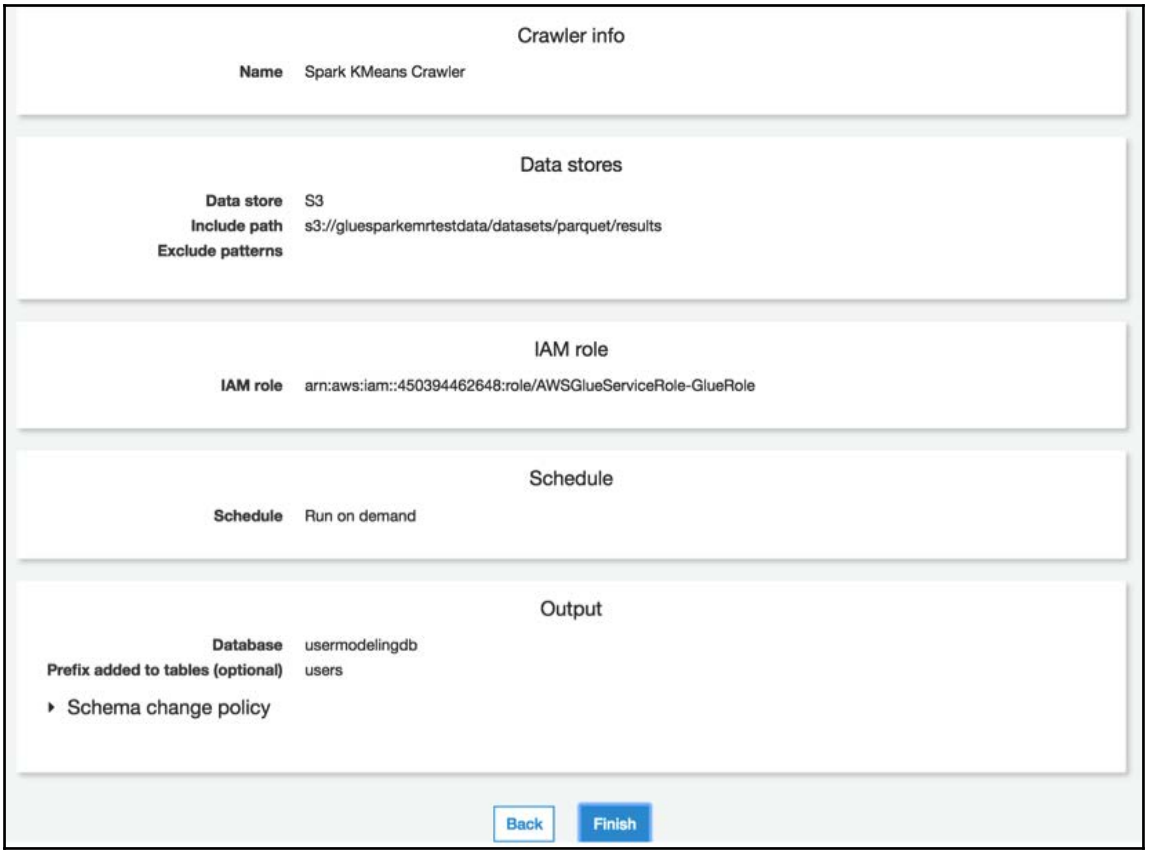
40. Review the **Job properties** and click on the **Finish** button:



41. Follow the steps for running the job as shown in the previous steps of this exercise.
42. After the job has completed successfully, check the S3 results folder to confirm that the output files are present:



43. Next, we will create another crawler to parse the results file. Follow the steps listed for creating a crawler as shown earlier in this exercise. We display the final review screen for the crawler for reference purposes:



44. After the crawler finishes processing, you should see another table added to your database (**usersresults**).
45. Verify the contents of the newly created table (userresults) using Athena as shown previously. Note that a new column called prediction is added by the K-Means algorithm.

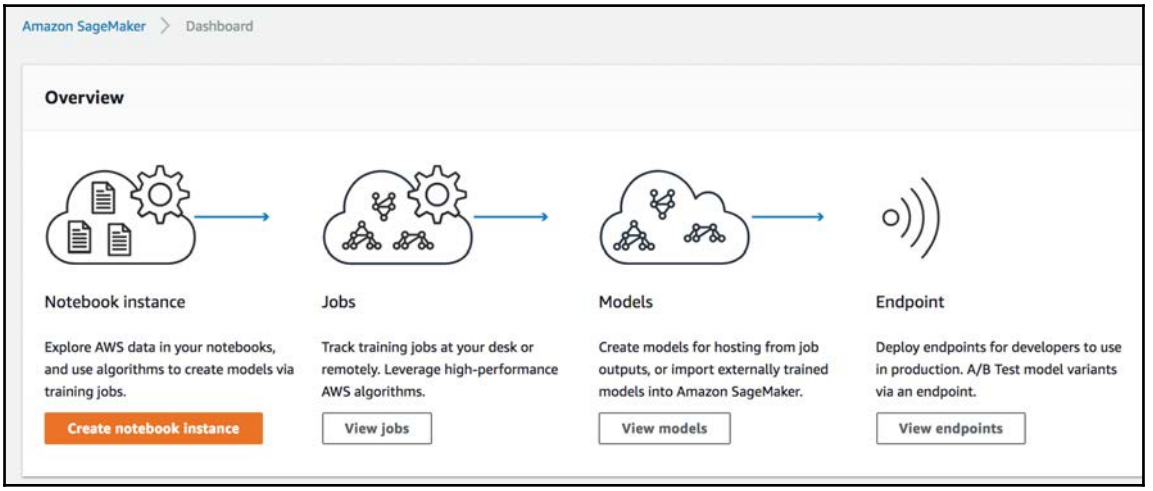
Using Amazon SageMaker

In this section, we will demonstrate setting up an Amazon SageMaker notebook instance. Run a sample machine learning job and create an endpoint to host the model.

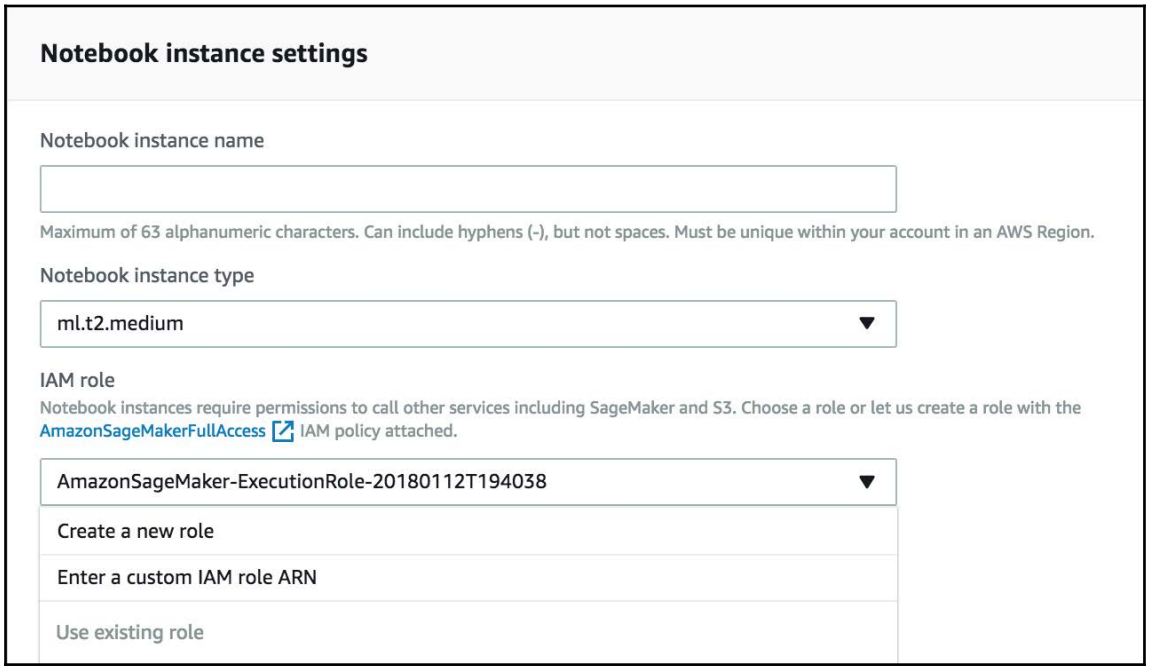


Refer to the detailed comments and explanations in the sample Python notebook used in this section at: `https://github.com/awslabs/amazon-sagemaker-examples/blob/master/introduction_to_applying_machine_learning/breast_cancer_prediction/Breast%20Cancer%20Prediction.ipynb`.

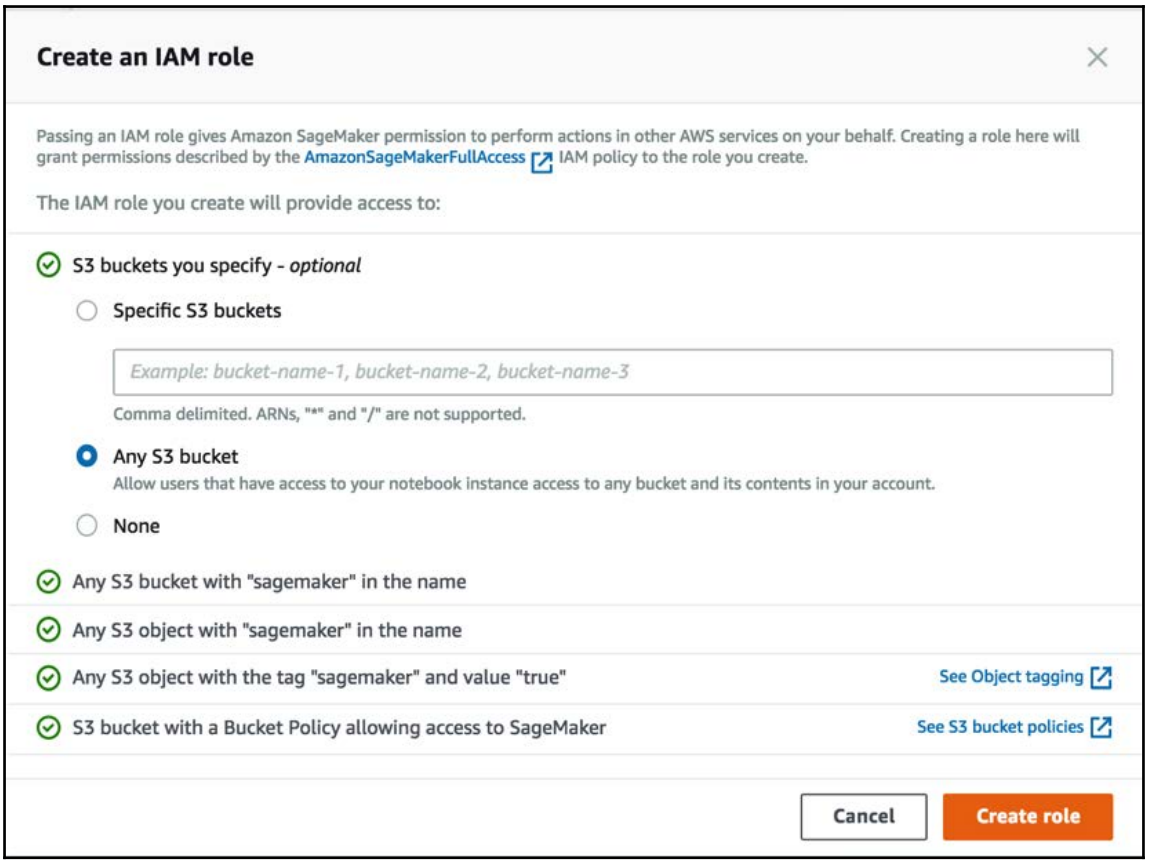
1. Log in to AWS Management Console and go to the **Amazon SageMaker** console. Click on the **Create notebook instance** button:



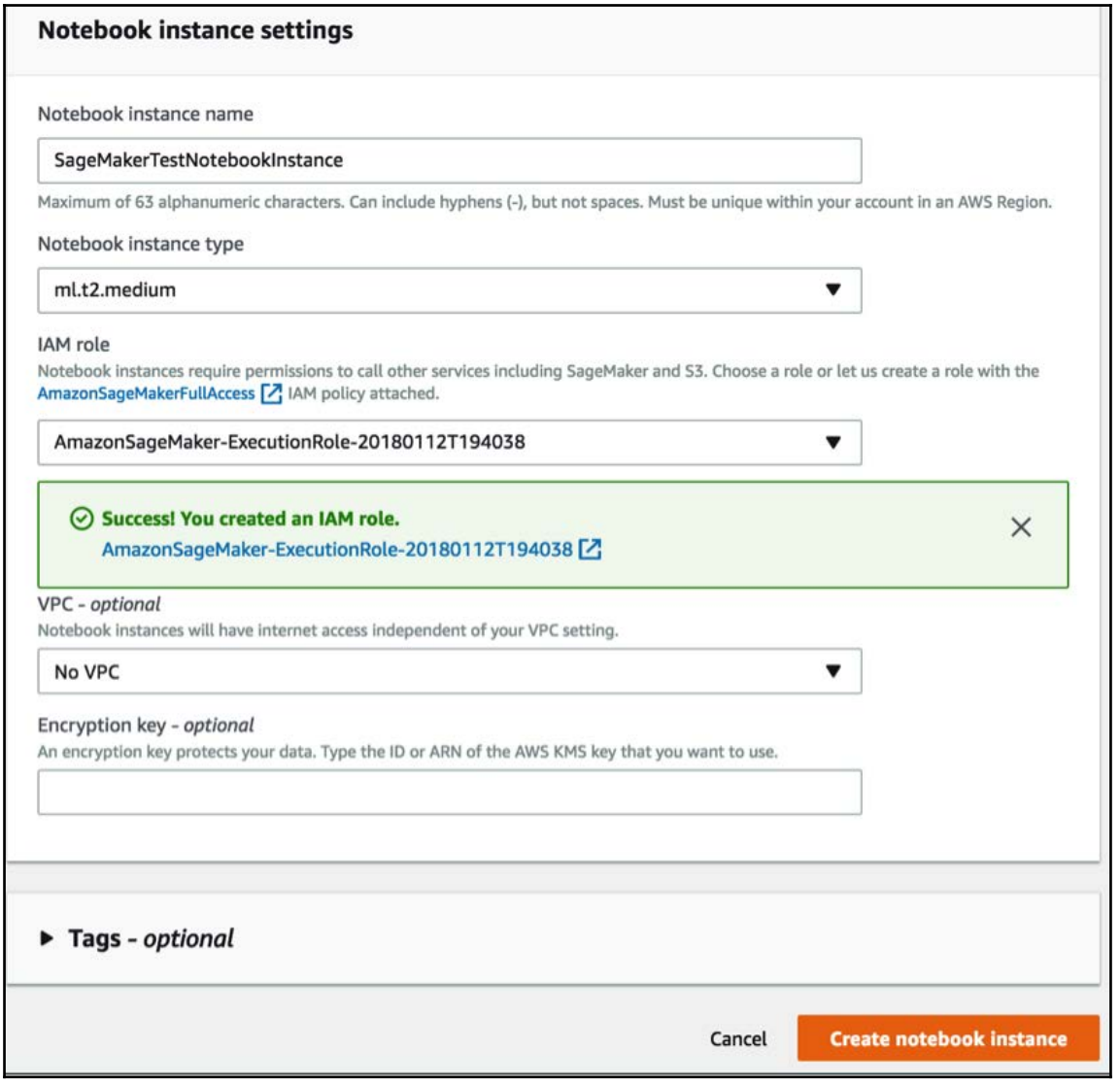
2. On the **Notebook instance settings**, we will create an Amazon SageMaker execution role. Click on the **IAM role** drop-down list and select the **Create a new role** option:



3. Select the **Any S3 bucket** option and click on the **Create role** button:



4. Specify the **Notebook instance name** (as `SageMakerTestNotebookInstance`), select the **Notebook Instance type**, and select the **No VPC** option. Click on the **Create notebook instance** button:

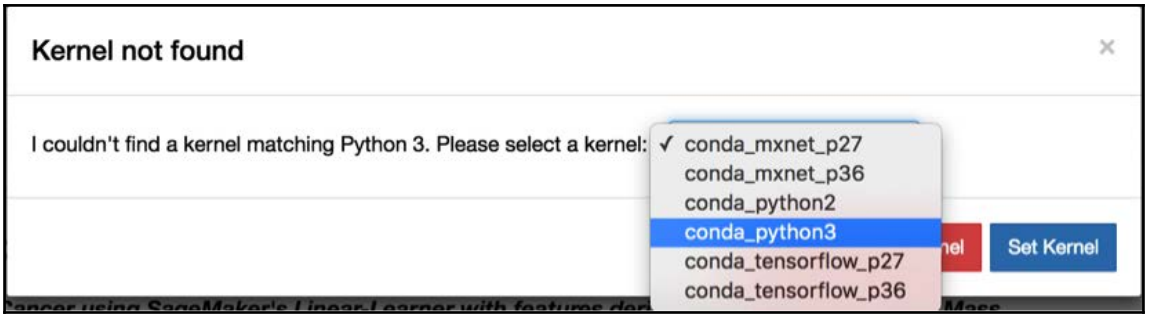


5. After the notebook instance is created, you should see a success message. The newly created notebook instance listed on the **SageMaker Notebook instances** screen. The status should change from **Pending** to **InService.**

6. Click on the **SageMaker notebook instance** to see the details for it. Click on the **Open** button to open the Jupyter notebook:



8. Navigate through the samples provided: **/sample-notebooks/introduction_to_applying_machine_learning/breast_cancer_prediction.**
9. Click on the **Breast Cancer Prediction.ipynb** file. Select **conda_python3** from the kernel drop-down menu:



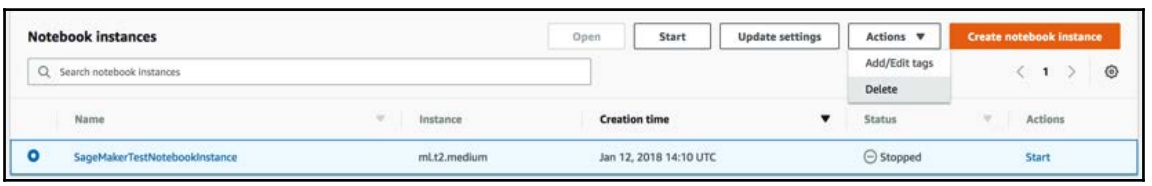
10. Create a S3 bucket (from the S3 console) and name it `sagemakertestdata`.
11. Change the bucket name to the newly created bucket:

```
In [ ]: import os
        import boto3
        import re
        from sagemaker import get_execution_role

        role = get_execution_role()

        bucket = 'sagemakertestdata'# enter your s3 bucket where you will copy data and model artifacts
        prefix = 'sagemaker/breast_cancer_prediction' # place to upload training files within the bucket
```

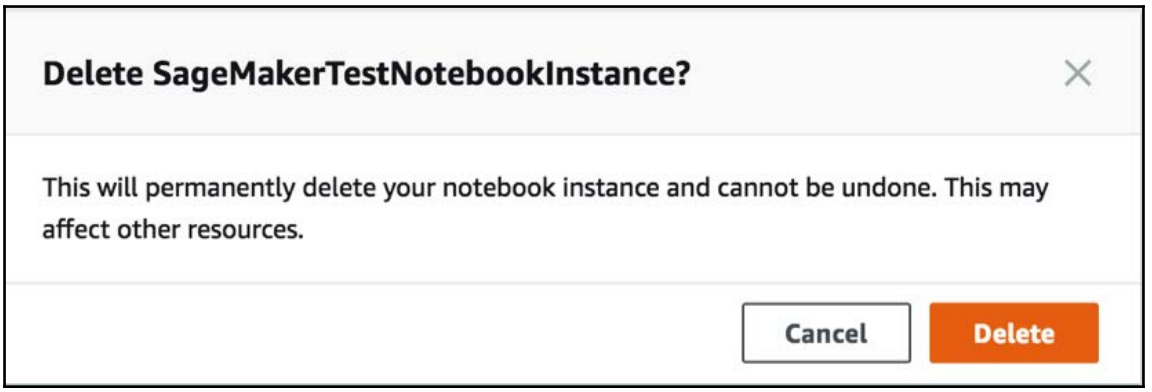
12. Execute the code in the notebook cell by cell. The code creates a training job that can be seen on the SageMaker Jobs console. While the training job is running, you will see the **Status** as **inProgress.** The **Status** changes to **Completed** when the training is completed.
13. After training the linear algorithm on our data, we set up a model that can be hosted later. After we set up the model, we configure and create the hosting endpoints. You can view the endpoint configuration on the SageMaker Endpoint configuration console. Wait for the **Status** to change to **inService.**
14. The notebook contains code for testing the model using the endpoint. Finally, execute the following cell in the notebook to delete the endpoint:

```
Run the cell below to delete endpoint once you are done.

In [17]: sm.delete_endpoint(EndpointName=linear_endpoint)

Out[17]: {'ResponseMetadata': {'HTTPHeaders': {'connection': 'keep-alive',
    'content-length': '0',
    'content-type': 'application/x-amz-json-1.1',
    'date': 'Fri, 12 Jan 2018 15:26:26 GMT',
    'x-amzn-requestid': 'a9dcea45-8098-4b65-ab48-e668ca32296e'},
   'HTTPStatusCode': 200,
   'RequestId': 'a9dcea45-8098-4b65-ab48-e668ca32296e',
   'RetryAttempts': 0}}
```

16. Confirm that the endpoint is deleted (from the SageMaker Endpoint console).
17. From the **Actions** menu on the SageMaker **Notebook instances** console, choose the **Delete** option:



18. Click on the **Delete** button to confirm:



Summary

In this chapter, we presented a series of hands-on exercises for implementing the key components of a big data application. We included Amazon Kinesis, AWS Lambda functions, Amazon Glue and Athena, and Amazon SageMaker for machine learning applications.

In the next chapter, we will explore the deployment big data applications on AWS using Amazon CloudFormation templates and AWS SAM (Serverless Application Model) for serverless applications.

10
Deploying a Big Data System

In `Chapter 7`, *Deploying to Production and Going Live*, we explored the deployment of applications in AWS environments, and in `Chapter 9`, *Implementing a Big Data Application*, we presented a hands-on session in Amazon SageMaker that also deployed the machine learning model. In this chapter, we will shift our focus to the deployment of big data applications using AWS services. More specifically, we will implement typical deployment use cases for big data and serverless computing applications using AWS services such as CloudFormation, Lambda, Serverless Application Model (SAM), Cloud9, and CodeDeploy.

In this chapter, we will cover the following topics:

- Using CloudFormation templates
- Authoring and deploying serverless applications
- Understanding AWS SAM
- Using AWS Serverless Application Repository

Using CloudFormation templates

CloudFormation is used to define, provision, and manage a collection of related AWS resources. This collection is referred to as a CloudFormation stack. The stack's template can be defined in YAML or JSON. Hence, the input to CloudFormation is a YAML file and the output is the provisioned AWS resources.

There are many sample CloudFormation templates available from AWS. You will probably never have to create a CloudFormation template from scratch. You can always download a sample and modify it to meet your requirements. These samples include templates for big data applications, including streaming applications, machine learning and deep learning applications, serverless applications, and many more.



Access the GitHub repository for AWS labs for sample CloudFormation templates at: `https://github.com/awslabs`.

In this section, we will use a sample CloudFormation template to create a data lake solution.

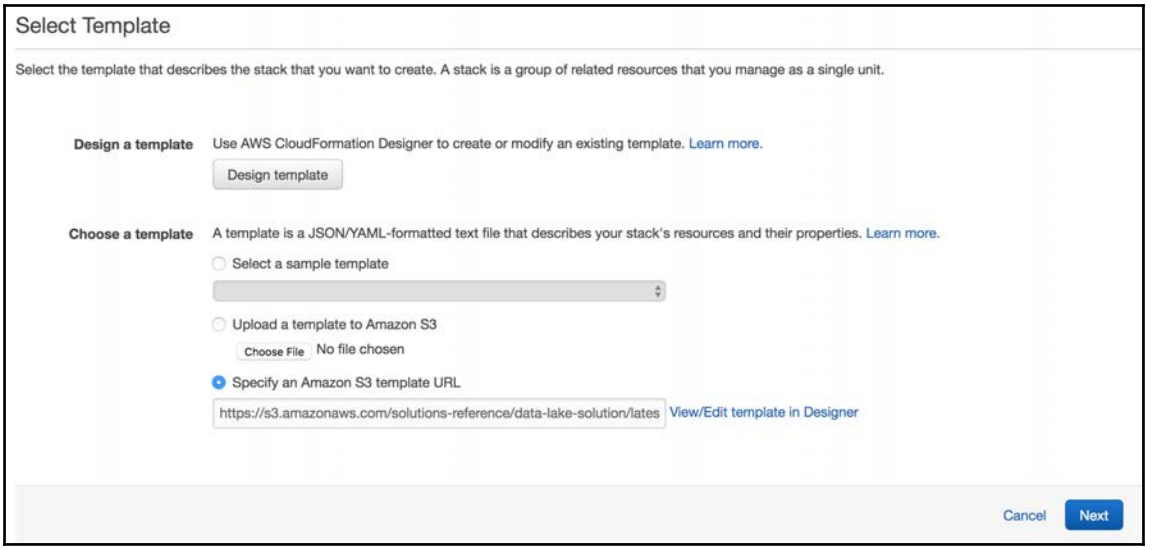


For more detailed instructions on the data lake CloudFormation example presented here, refer to: `https://github.com/awslabs/aws-data-lake-solution`.

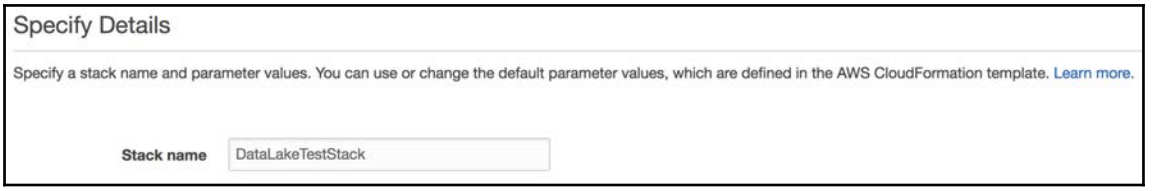
Creating a data lake using a CloudFormation template

The following are the steps to create a data lake using a CloudFormation template:

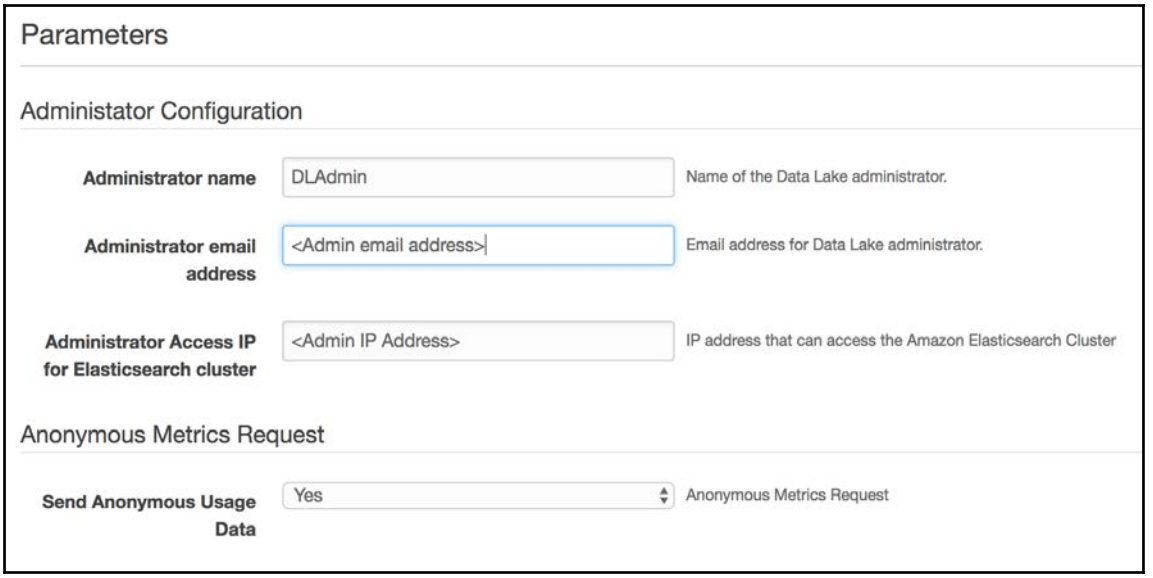
1. Follow the instructions on the GitHub page for this example. Copy the link to the S3 location for the template file. Change the region (on the Management Console to point to your region, if required). Click on the **Next** button:



2. Specify a **Stack name** as `DataLakeTestStack`, which is shown as follows:



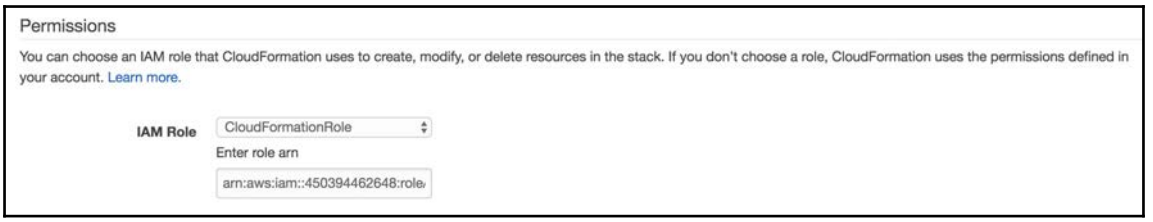
3. Specify the **Administrator Configuration** details, including name, email address, and access IP for the ElasticSearch cluster (for test purposes this can be `0.0.0.0\0` for now). You can choose the **Send Anonymous Usage Data** parameter as **Yes**. Send anonymous usage data to AWS. This data is used by AWS to better understand how customers use this solution and related services and products:



4. Specify tags for the various resources in your stack. For now, we leave them blank:



5. Leave the **IAM Role** as **CloudFormationRole** as shown:



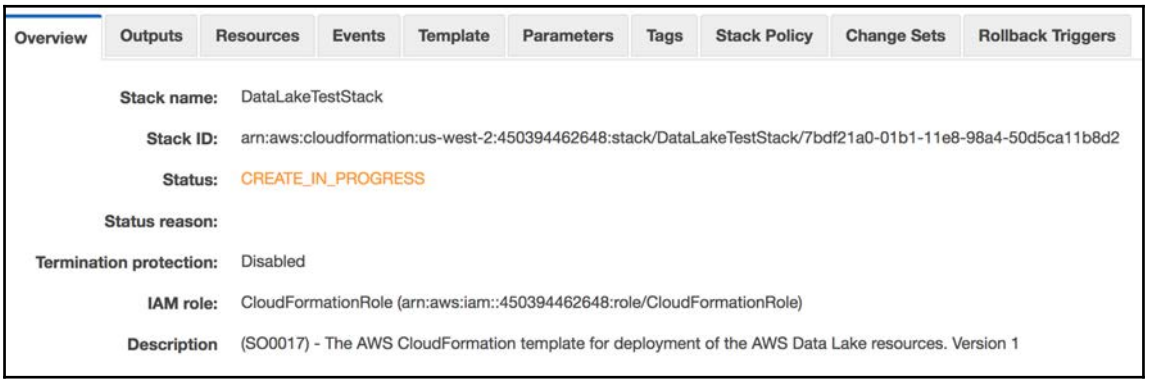
6. Review all the details, and check the acknowledgment and click on the **Create** button:



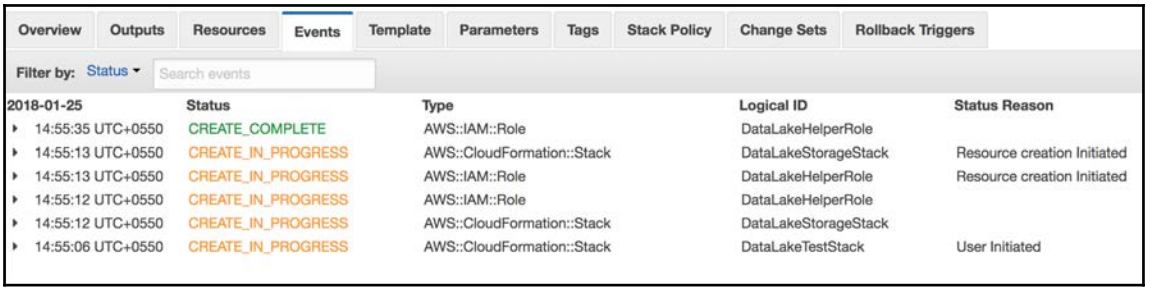
7. At this stage, you should see the **CREATE_IN_PROGRESS** for your stack (and the nested stacks within) on the CloudFormation console:



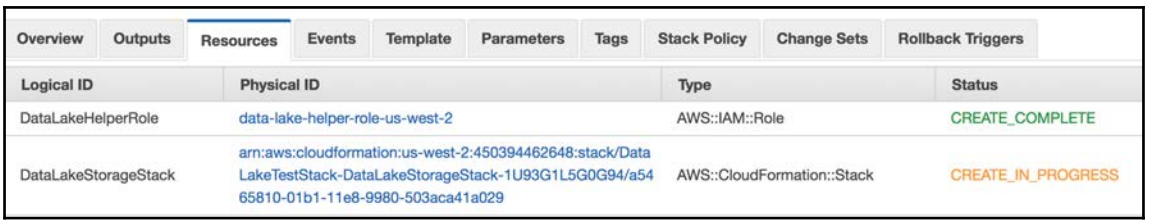
8. Select a stack (**DataLakeTestStack**) from the list to see more detailed information. Click on the **Overview** tab:



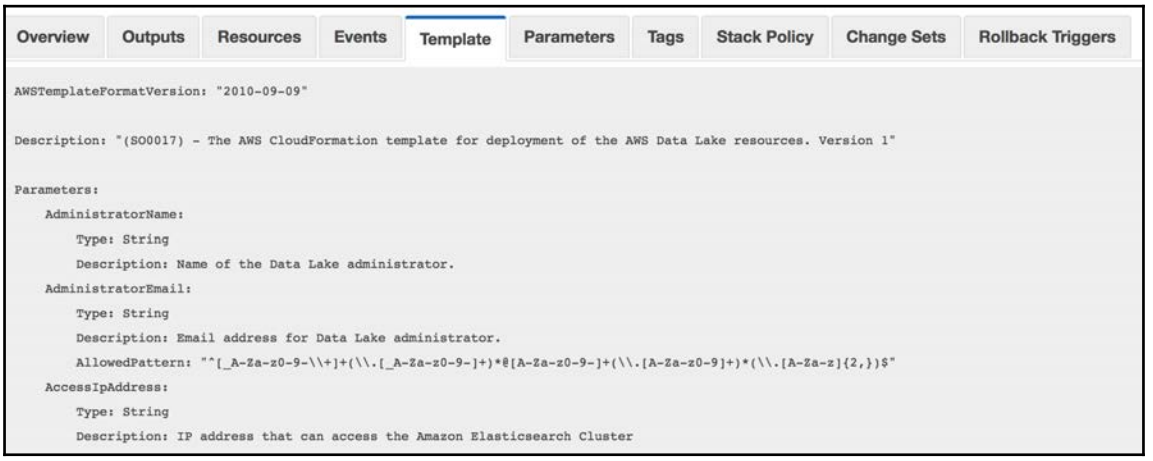
9. Click on the **Events** tab to see the statuses as the creation events execute:



10. The **Resources** tab displays the AWS resources and their current status:



11. If you want to view the CloudFormation template and then click on the **Template** tab:



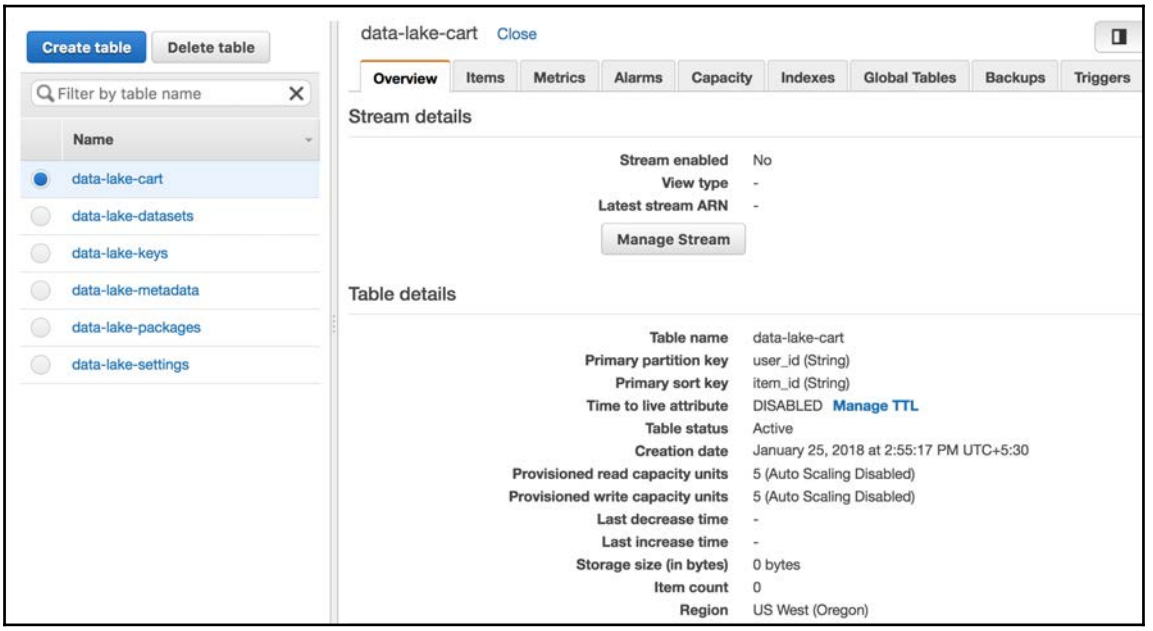
12. After 20-25 minutes, you should see the **CREATE_COMPLETE** message against your stack (and the nested stacks):



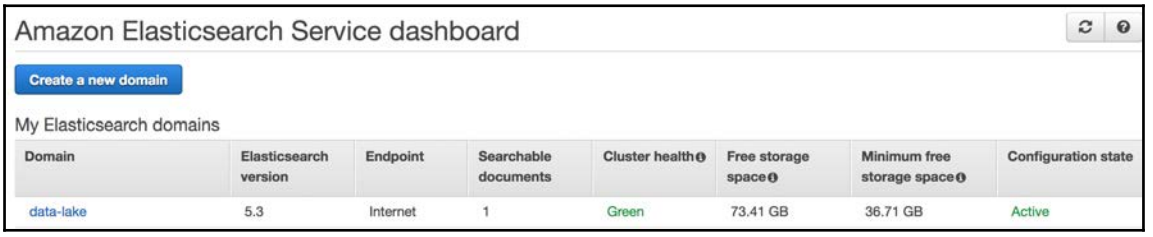
13. Go to the DynamoDB console to verify that the tables were created successfully:

Name	Status	Partition key	Sort key
data-lake-cart	Active	user_id (String)	item_id (String)
data-lake-datasets	Active	package_id (String)	dataset_id (String)
data-lake-keys	Active	access_key_id (String)	user_id (String)
data-lake-metadata	Active	package_id (String)	metadata_id (String)
data-lake-packages	Active	package_id (String)	-
data-lake-settings	Active	setting_id (String)	-

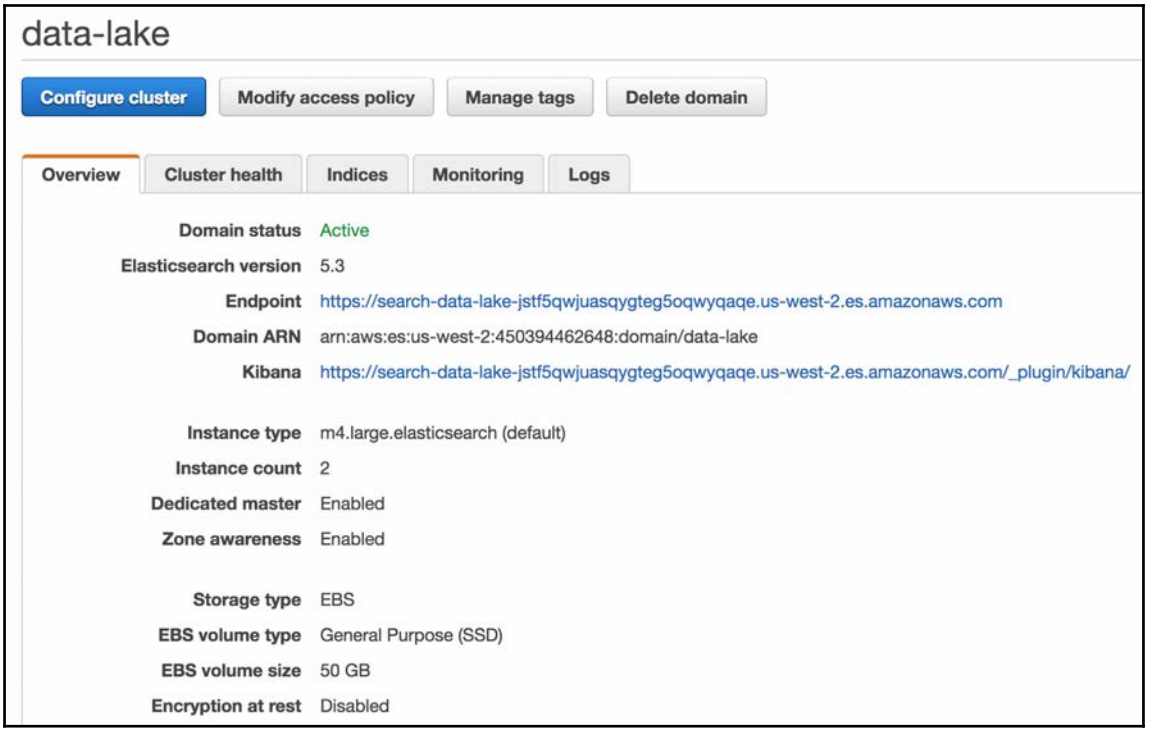
14. You can view the details of a specific DynamoDB table by selecting it from the list:



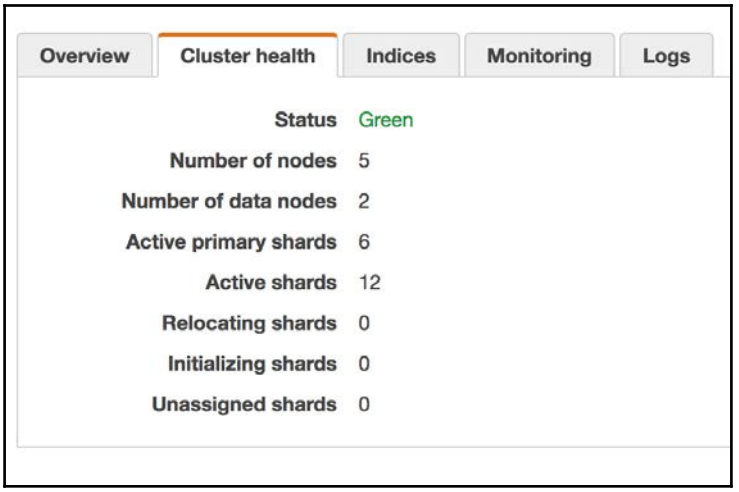
15. Similarly, you can go to the **Amazon Elasticsearch Service** dashboard to verify the creation of the Elasticsearch domain:



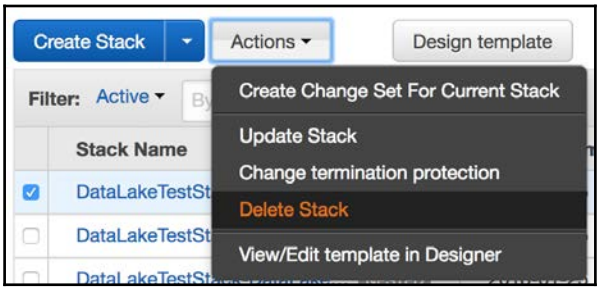
16. Click on the domain name to verify the details of the Elasticsearch cluster:



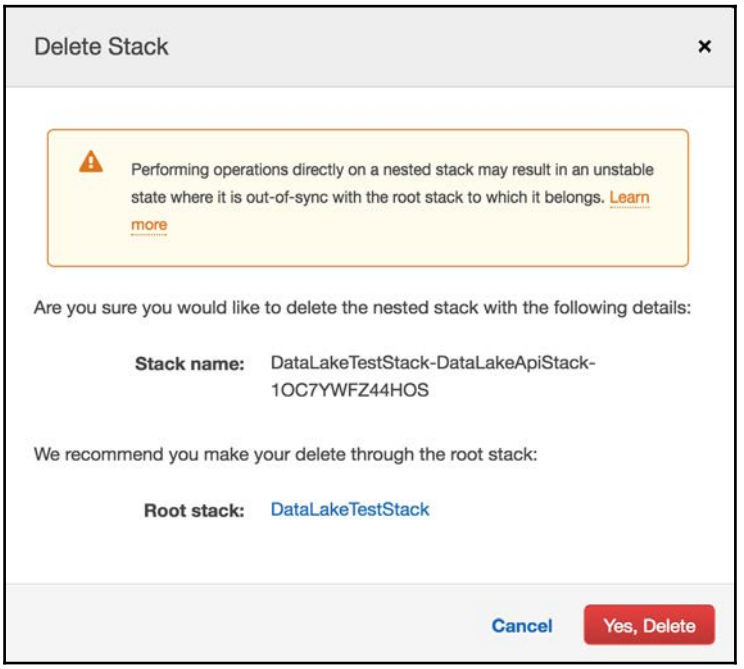
17. Click on the **Cluster health** tab to ensure that the **Status** is **Green**:



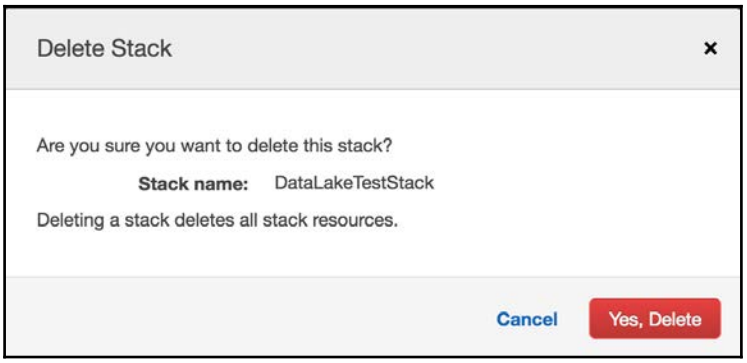
18. At this stage, you have a fully functional data lake provisioned using a CloudFormation template. You should have received an email with the user ID and temporary password in the Administrator's mailbox. Use the credentials to sign in to your data lake and explore it. As it is expensive to keep this stack running, select the **Delete Stack** option from the **Actions** menu:



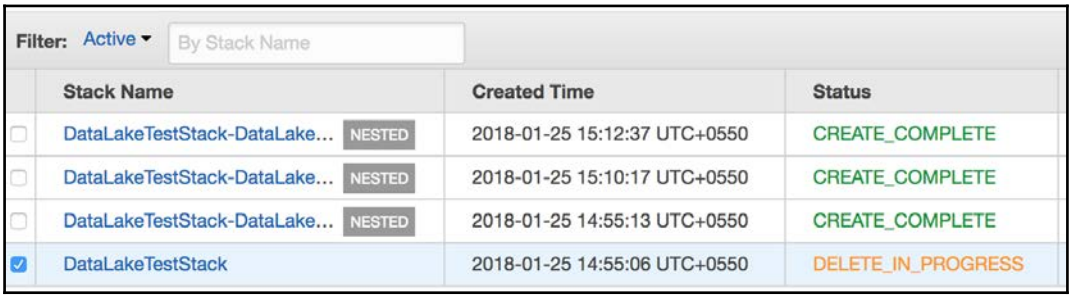
19. Here, we get a warning because we tried to delete a nested stack. As we don't want to be in an unstable state with respect to the root stack, we cancel out of this screen:



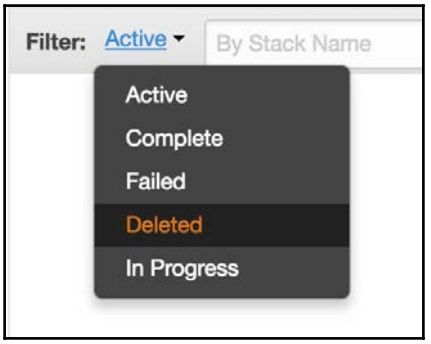
20. Select the root stack (**DataLakeTestStack**) and choose **Delete Stack** option from the **Actions** menu. Click on the **Yes, Delete** button:



21. You should see the **Status** for the stacks change to **DELETE_IN_PROGRESS** as shown:



22. After the stack is deleted, select the **Deleted** filter to see the details of stacks you deleted:



23. You will see the root stack and the nested stack listed with the **Status** as **DELETE_COMPLETE**:

Stack Name	Created Time	Status
DataLakeTestStack-DataLakeApiStack-...	2018-01-25 15:12:37 UTC+0550	DELETE_COMPLETE
DataLakeTestStack-DataLakeServicesS...	2018-01-25 15:10:17 UTC+0550	DELETE_COMPLETE
DataLakeTestStack-DataLakeStorageSt...	2018-01-25 14:55:13 UTC+0550	DELETE_COMPLETE
DataLakeTestStack	2018-01-25 14:55:06 UTC+0550	DELETE_COMPLETE

24. You can quickly verify that the DynamoDB tables, Elasticsearch Domains, and other resources have been deleted by going to their respective consoles. The following screen shows that the DynamoDB tables no longer exist:



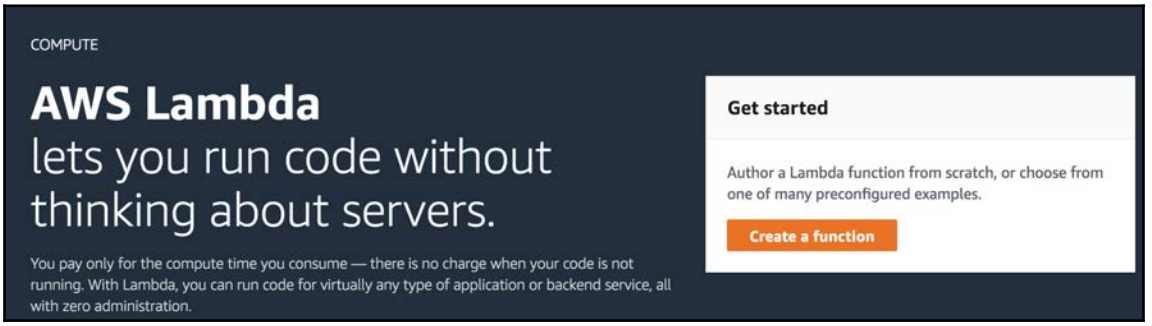
Authoring and deploying serverless applications

There are several choices for authoring and deploying serverless applications. We can use different tools for different use cases. For example, you can use the Lambda console for quick creation and iteration of simple apps. It is easy to use and has a built-in dev environment. If you have a more complex application, then you can define those with SAM and take advantage of tools built on top of SAM such as SAM Local for testing and debugging. You can plug SAM local into the IDE of your choice or use Cloud9 that is optimized for serverless applications and has SAM Local built-in. For incrementally rolling out new versions into production, you can build on SAM for CI/CD capabilities, including canary deployments.

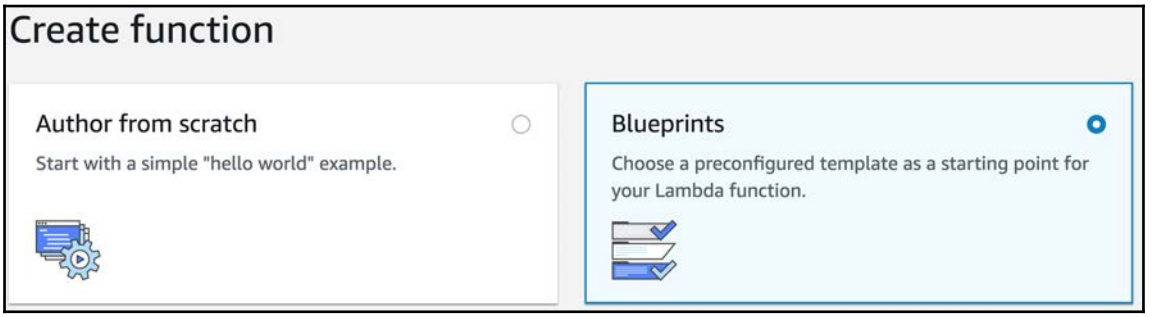
The Lambda console now has the Cloud9 editor, which is optimized for serverless applications for quick author-test-debug iterations in serverless application development. It gives you an IDE-like experience. Realistically, this application lifecycle includes code check-in (after authoring and testing the code is completed) that kicks off the automated deployment process. This process picks up the code from the code repository, packages it, and builds the application. It then tests and deploys it in the production environment.

Using AWS Lambda console to author and test Lambda functions:

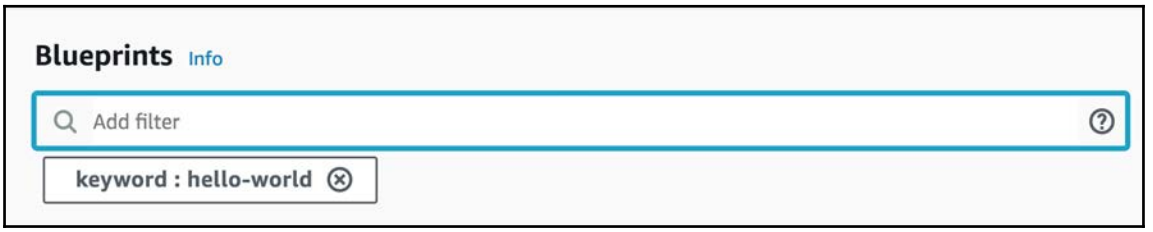
1. Log in to the Management Console and go to the Lambda console. Click on **Create a function** button:



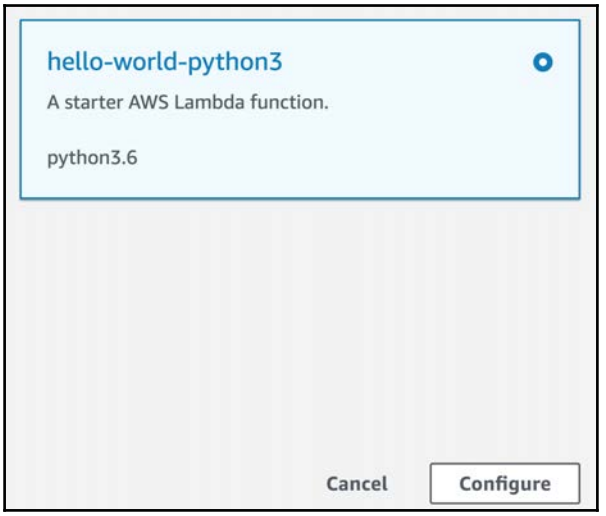
2. We have shown the process of creating a Lambda function from scratch in `Chapter 9`, *Implementing a Big Data Application*. Here, we will select a pre-configured template for a Lambda function from the available blueprints:



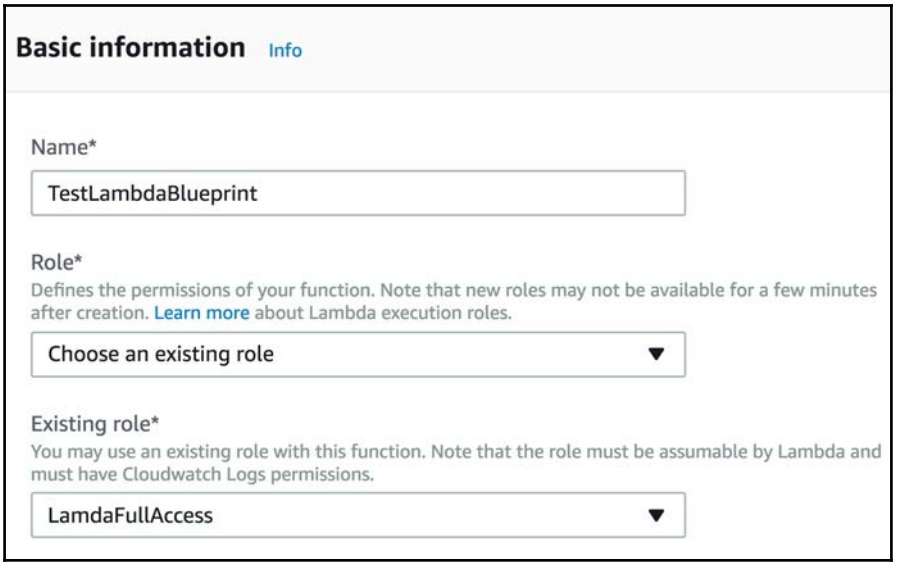
3. Search for the blueprint by entering `hello-world` in the filter field:



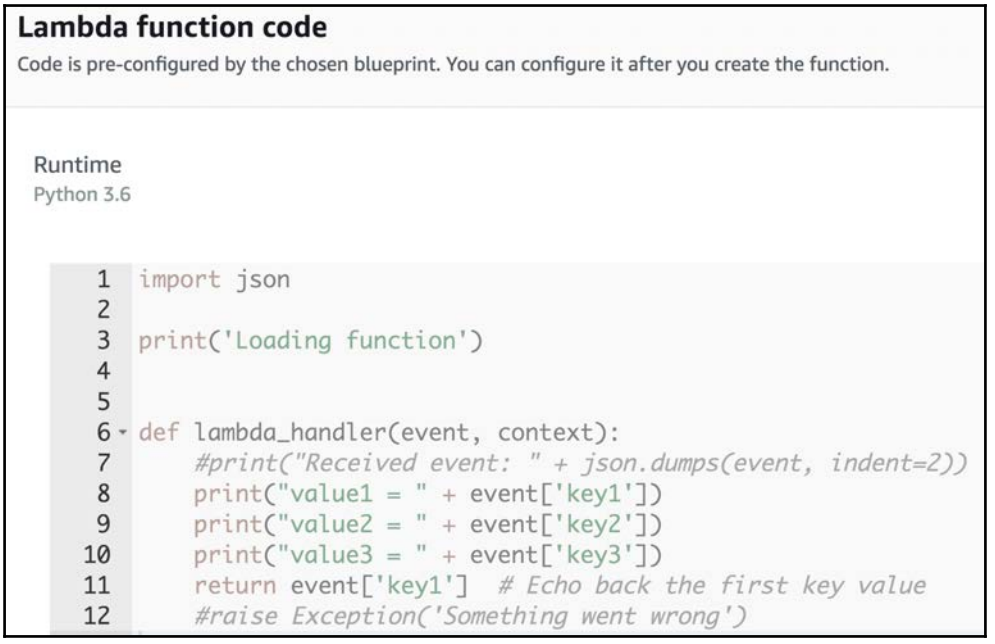
4. For simplicity, we choose **hello-world-python3** as the blueprint here. Check on the radio button and click on the **Configure** button:



5. Provide a name for the Lambda function (`TestLambdaBlueprint`). For **Role** select the **Choose an existing role** option and then select **LambdaFullAccess** as the **Existing role** (you should be more selective in your production environment to restrict access to only the resources required by this function):



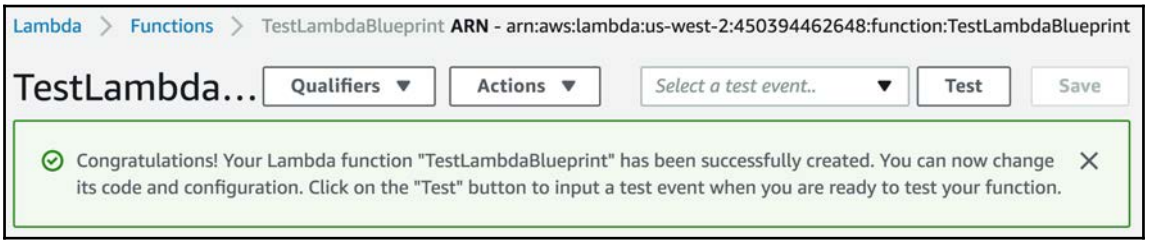
6. Scroll down to see the Lambda function code:



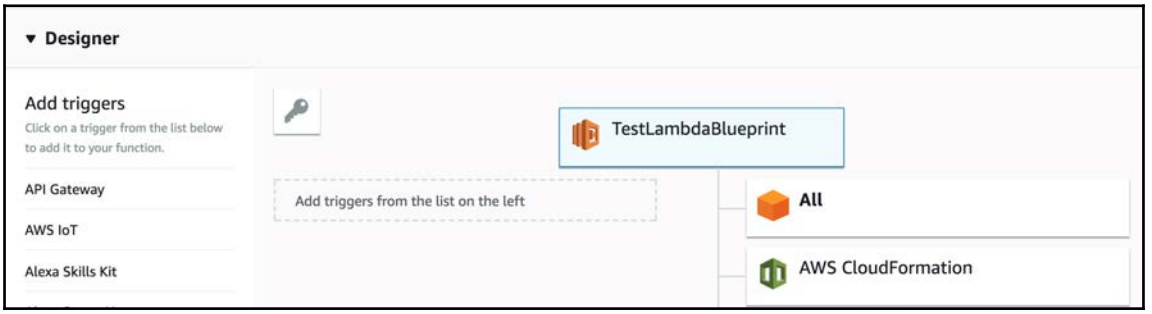
7. Click on the **Create function** button:



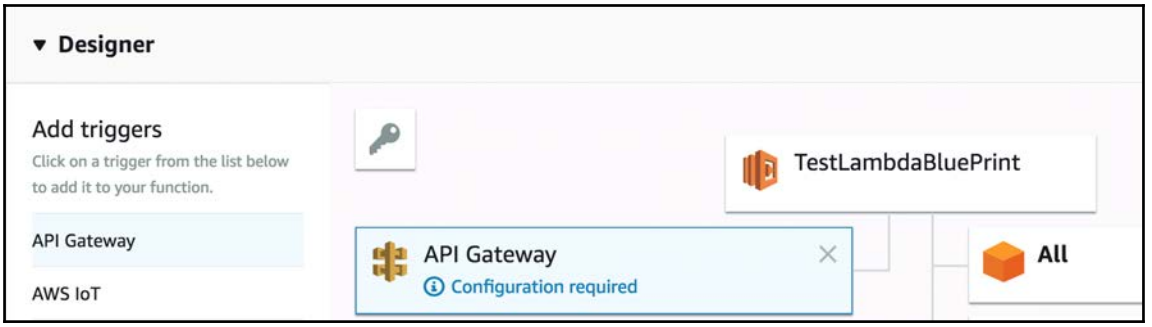
8. You should see the message as shown to confirm the successful creation of the Lambda function:



9. The view you land on contains function graph or the designer view. It is a visualization of what flows into your Lambda function and what flows out of it. You can see the event sources that are configured to trigger the function and also the downstream resources the function has permissions to access. The function is displayed in the middle:



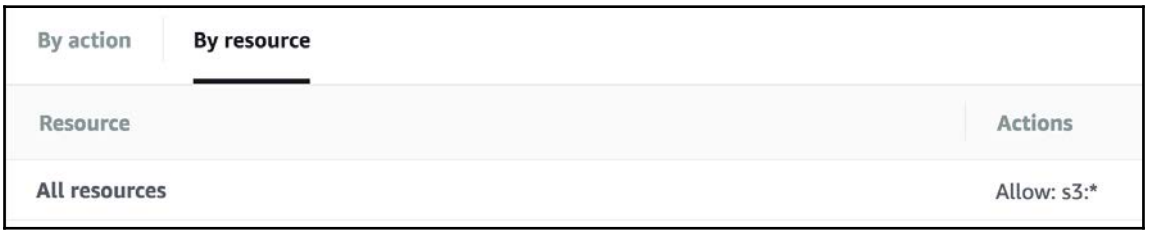
10. On the left of the function box, you can see the list of triggers that are added to the function. We can click on a trigger from the list located in the left pane and that particular resource is added as a trigger. Click on **API Gateway** in the list of triggers:



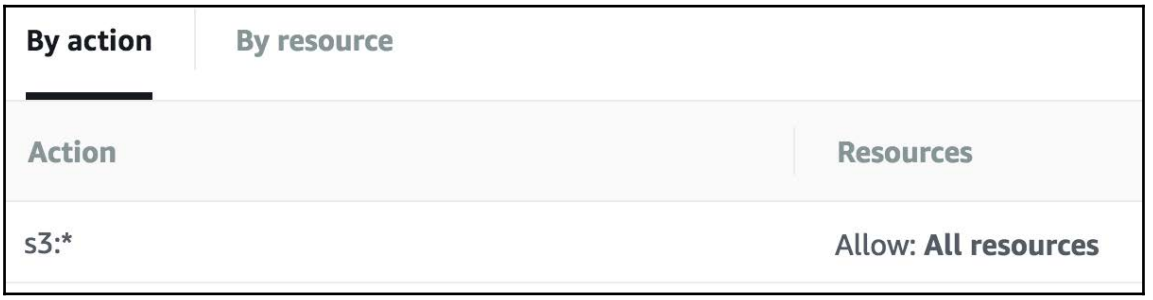
11. If you scroll down further at this stage, you will see that the panel below also changes accordingly to allow you to configure the newly added trigger as shown:



12. On the right of the function box are the resources the function can access. The CloudWatch logs are added by default to all functions, so that your function can emit logs to CloudWatch. By clicking on the right, you can see the details by resources and actions the function can perform, on which resources. Click on S3 to see the resource and permitted actions on them:

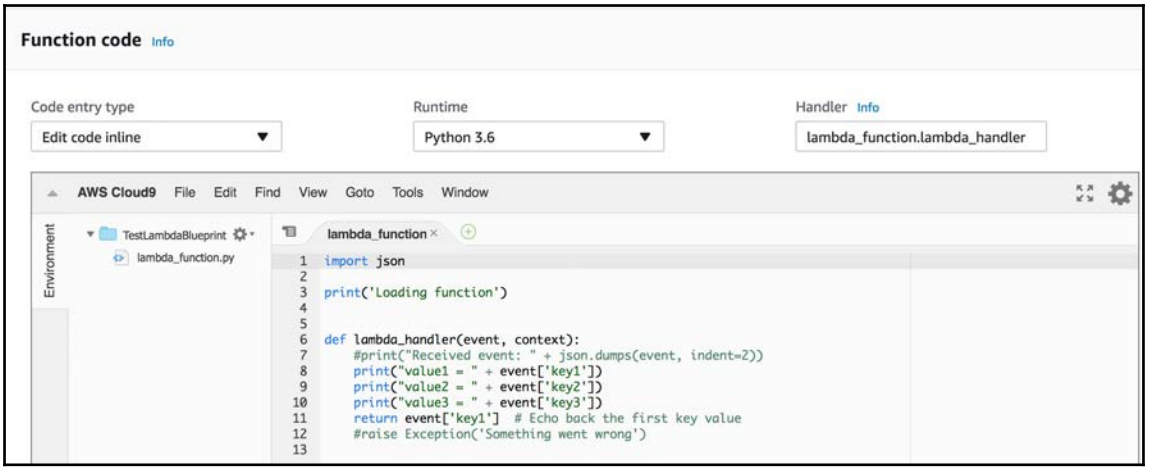


13. Click on the **By action** link to see the details by permitted actions against the resources:

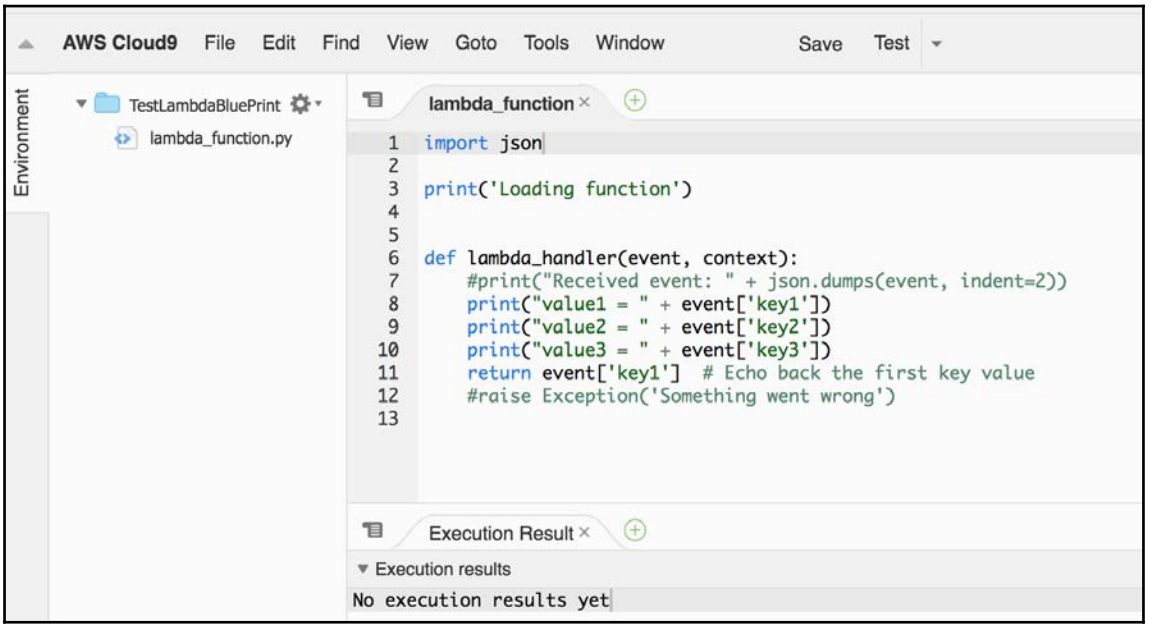


14. Scroll further down to see the source code of the function. You can see all the files in the deployment package in the left pane:

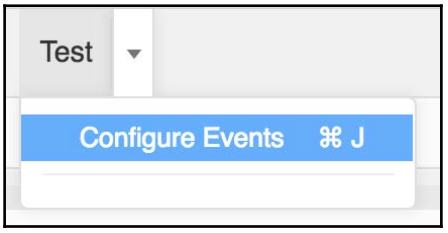
>



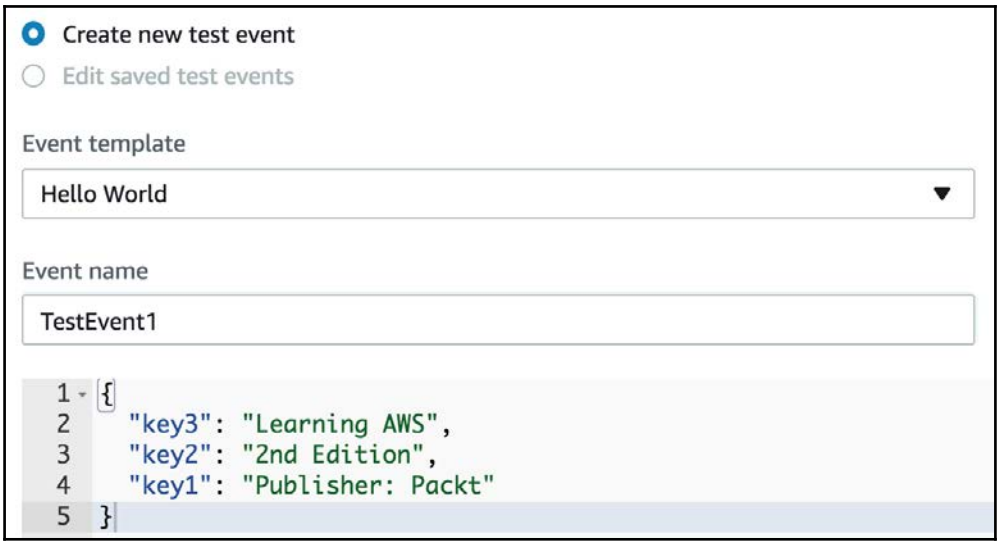
15. Press *Cmd+Shift+F* on Mac (or *Ctrl+Shift+F* on Windows) to go into a full-screen mode for an IDE-like experience:



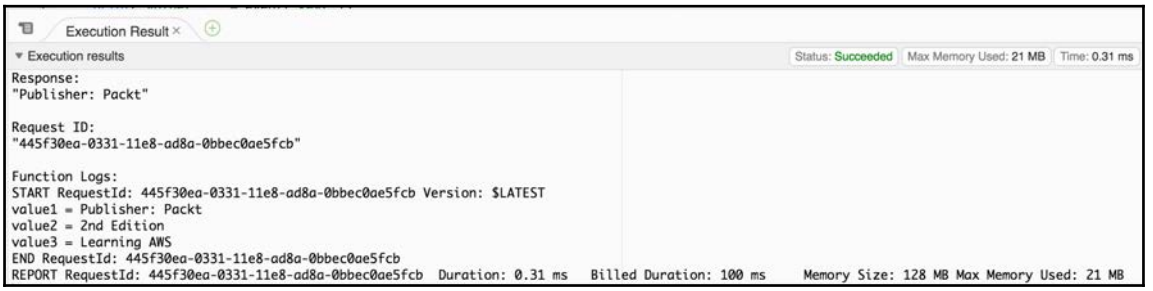
16. Next, we will define a test event for testing our function. You can create multiple test events and persist them. Click on **Test** and select **Configure Events** from the drop-down menu:



17. Create the Test Event as shown:



18. Click on **Test** to execute the Lambda function. You will see the response right below the code window. Some important metrics are also displayed as shown:



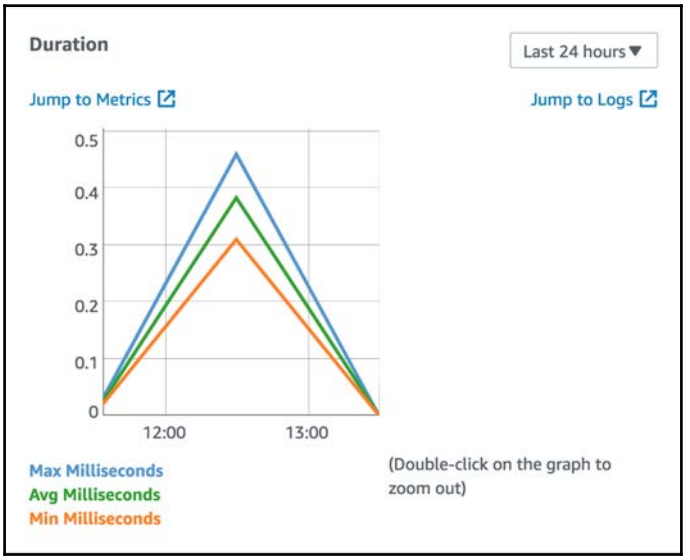
19. Scroll back up to the **Designer** section and click on the **Monitoring** tab:



20. In **Monitoring** view, there are interactive graphs that allow you to zoom into and out off any time frame and jump to the logs or the metrics for that time frame:



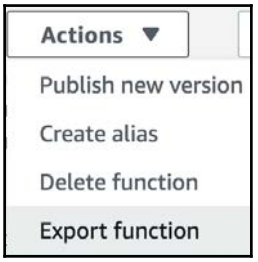
21. In the **Duration** pane, select the period during which you executed the Test events. The view changes as shown:



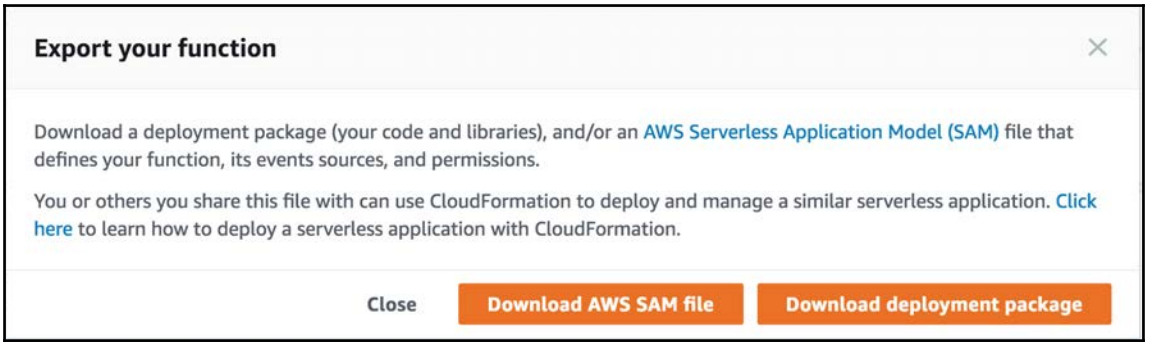
22. Click on the **Jump to Logs** link to view the CloudWatch logs for that time period:

Time (UTC +00:00)	Message	Show in stream
2018-01-27		
	No older events found for the selected date range. Adjust the date range.	
07:11:17	Loading function	2018/01/27/[$LATEST]d2bfd731...
07:11:17	START RequestId: 445f30ea-0331-11e8-ad8a-0bbec0ae5fcb Version: $LATEST	2018/01/27/[$LATEST]d2bfd731...
07:11:17	value1 = Publisher: Packt	2018/01/27/[$LATEST]d2bfd731...
07:11:17	value2 = 2nd Edition	2018/01/27/[$LATEST]d2bfd731...
07:11:17	value3 = Learning AWS	2018/01/27/[$LATEST]d2bfd731...
07:11:17	END RequestId: 445f30ea-0331-11e8-ad8a-0bbec0ae5fcb	2018/01/27/[$LATEST]d2bfd731...
07:11:17	REPORT RequestId: 445f30ea-0331-11e8-ad8a-0bbec0ae5fcb Duration: 0.31 ms Billed Duratio	2018/01/27/[$LATEST]d2bfd731...

23. Next, from the **Actions** menu, select the **Export** function option:



24. Click on the **Download AWS SAM** file button and then on the **Download deployment** package button. The deployment package essentially contains your function's source code in this example. We will discuss the contents of the SAM file in a later section:



25. From the **Actions** menu, select the **Delete function** option:



26. Click on **Delete** to confirm the deletion:



27. You should see the following message after the function has been deleted successfully:



Understanding AWS SAM

Typically, the application development process is more complex than that with the environment containing code repositories. Most applications comprise dozens of resources and several Lambda functions. For such applications, using Lambda console is not a good option. Also, though CloudFormation is a powerful resource, the issue is that it was built years ago, and it is optimized for infrastructure and less so for serverless.

SAM (Serverless Application Model) is an open specification (Apache 2.0) realized as a CloudFormation extension optimized for serverless. It is essentially a CloudFormation template under the covers. Hence, SAM can support anything that CloudFormation supports. You can define any CloudFormation resource within your SAM template and it will work just fine.

Understanding the SAM template

The following is a YAML source for the SAM template we downloaded (from the export function option) for the Lambda function defined in the previous section:

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Description: A starter AWS Lambda function.
Resources:
    TestLambdaBluePrint:
        Type: 'AWS::Serverless::Function'
        Properties:
            Handler: lambda_function.lambda_handler
            Runtime: python3.6
            CodeUri: .
            Description: A starter AWS Lambda function.
            MemorySize: 128
            Timeout: 3
            Role: 'arn:aws:iam::450394462648:role/LamdaFullAccess'
            Tags:
                'lambda-console:blueprint': hello-world-python3
```

You have a Lambda function defined here. Below that you are defining a couple of Lambda function properties, including a link to the code, the handler, the runtime, and some IAM policies that the function is going to assume. Below that we would define the event source (we did not select a specific event source in our example). If we were using the API Gateway, then we need not explicitly define the API here – the event source definition is sufficient for CloudFormation to provision that API in your account.

The Transform line tells CloudFormation to take this template and transform it into a full-blown CloudFormation template under the covers. CloudFormation will use this template to provision the resources in the AWS account. Typically, the resources that get provisioned here include the Lambda function, a couple of IAM resources, some API Gateway resources, and so on. The full-blown CloudFormation template that is generated is complicated and voluminous compared to the SAM.

Introducing SAM Local

Testing and Debugging present a unique set of challenges in serverless applications. Firstly, serverless means that you don't have to manage any servers. The underlying infrastructure is abstracted away from you and is managed by AWS. So how do you make sure that the code that runs successfully in the dev environment will actually run successfully in Lambda as well. Ideally, we need to test in an environment that resembles Lambda in terms of the OS, libraries, and runtime. Additionally, it would be great if the limits (memory, timeout) could be enforced locally. Also, mimic responses and logs locally. Lastly, it would be great to generate test events that are syntactically accurate and different for each trigger. This is challenging because Lambda has over fifteen different event sources, currently and each source emits its own specific event pattern to Lambda. For example, for an S3 event source, we would have to retrieve the event, save it, and then customize it for our testing needs. This process can be time consuming, especially if you have to do this for multiple event sources. The tool that helps you do all this is SAM Local.

SAM Local is an open source CLI tool for local testing of serverless apps that ensures that your application runs as expected in the production Lambda environment. It leverages Docker images to mimic Lambda's execution environment and enforces configuration limits (same memory and timeout limits).

If you are using the API Gateway, then you can access your API on the local machine. SAM Local emulates the Lambda functions and APIs. Additionally, the event generator helps you generate event payloads for common Lambda triggers. The responses and logs are also as it would be in the Lambda environment. You can pipe the output to stdout and stderr and you can also persist the logs for further inspection. Additionally, SAM Local exposes a debugger port so you can debug your applications using any IDE.

Some of the other useful SAM Local commands help you validate the SAM template against the specification on your local machine. This provides quick feedback locally instead of discovering issues during deployment. The start API command, starts up a local http server and makes all your APIs available to you on your local machine. The invoke command allows you to locally execute your function.

All of this SAM Local functionality is available out of the box in AWS Cloud9. It provides an easy-to-use UI to strip away the complexity of SAM Local commands.

Developing serverless applications using AWS Cloud9

AWS Cloud9 is a cloud-based dev environment used to write, test, and debug your serverless applications with just a browser. It is optimized for serverless and is fully integrated with the Lambda environment. You can import functions from the Lambda environment into your Cloud9 deploy functions back into the Lambda environment.

It comes with native support for SAM, so you can generate the templates easily. It comes prepackaged with all the tools you would expect from IDEs for the language of your choice. A customizable IDE can set themes and shortcuts as you expect from a local dev environment.



For a detailed coverage of AWS Cloud9, its features and tutorials, refer to: `https://aws.amazon.com/cloud9/`.

You can invoke the function locally (or remotely in the actual Lambda environment) using SAM Local under the covers. It comes with integrated debugger with features you would expect from an IDE debugger. You can deploy the function back to the Lambda environment, and invoke it remotely. In this case, the response is received from the actual Lambda environment.

Automating serverless application deployments

If the application is built on top of SAM, then all of the tooling is already in place for automated deployments. You can orchestrate this with CodePipeline, which can essentially pull your code (from the code repository), pull down the required packages, create the deployment package, and build the application. You can test the application using third-party tools that are natively supported by CodePipeline.



For more details on serverless application developer tooling, refer to: `http://aws.amazon.com/serverless/developer-tools`.

You can use the power of SAM to deploy in multiple environments. The problem here is that there is an immediate switch from the old to the new version of the code. And that is not how we usually deploy in production (see blue-green deployment in `Chapter 7`, *Deploying to Production and Going Live*). Usually, we incrementally roll out new versions to help reduce deployment risks.

However, you can do safe deployments with SAM. Lambda aliases now enable traffic shifting that is you can point an alias to two different versions and shift traffic between them using preassigned weights. Alias is essentially a pointer to a version. With weighted aliases, you can actually publish a new version and add it to the alias with percentages defined for each version. You can divert 5% of the traffic to the new version while 95% continues using the earlier version.

You also automate the deployment process using CodeDeploy. It comes with preconfigured canary and linear deployments (that is, you can linearly increase traffic as time progresses). It provides guardrails for safe deployments, for example, autoalarm-based rollbacks; pre- and post-traffic validation hooks (that is, test hooks as Lambda functions and can run before or after your traffic has shifted to the new version). You can easily monitor the deployment using the CodeDeploy console.

As SAM is natively supported, you just need to add a few lines to the SAM template for safe deployments. You can define function configurations for all the functions in the SAM template: tags, runtime, and other essential things that are common across all your functions. This is useful to define the deployment strategy for all the functions in the application at one place rather than for each function, separately. Next, you have to include a line for creating the alias (that autopublishes a version with each deployment). Lastly, you can define your deployment preferences with respect to the kind of deployment you would like to execute. For example, shift 10% of the traffic to the new version, if all goes well, then after 10 minutes shift all the traffic to the new version.

For test hooks and alarms, you can define pre- and post-traffic hooks (lambda functions that run as tests), they succeed or fail and accordingly they communicate back to CodeDeploy (so it knows how to react). The alarms can also be defined easily using the alarms construct within SAM. If any of the alarms are triggered during deployment, then that is going to prompt a rollback to the earlier version. It's easy to monitor through the CodeDeploy console.

Using AWS Serverless Application Repository

Developers and customers are facing challenges in finding, cloning, building, and deploying serverless apps. Typically, we need to get help from developers in the community, look at code samples, copy-and-paste the code, ensure they build, ensure they function properly, and ensure the roles and permissions are set up appropriately. Having to do all that is complicated for developers new to the the serverless paradigm.

There are several options to get started with serverless applications. Earlier, these options included AWS blueprints. However, the blueprints are a limited set that AWS creates and curates. It is not open to third parties and typically the functions have a limited scope in terms of the resources while serverless applications are a lot more than that (scalable, available, fault-tolerant, and secure). They are not linked to GitHub (so you can't go look at the source code and make changes). Hence, these blueprints are characterized by the limited selection, scope, and language choices.

At the same time, many serverless examples are shared on GitHub. GitHub provides a massive selection, but the developer has to find the right serverless apps, ensure source code quality, and they also have to recreate the build environment. GitHub is not connected to the authoring workflow for Lambda functions. There are essentially no hooks from GitHub to AWS that helps set up the right permissions and roles.

GitHub provides great community experience and blueprints provide a good authoring experience, but we need a solution that provides the pluses of both. The solution is the recently announced AWS Serverless Application Repository. Though the offerings are limited currently, it will definitely pick up in the weeks and months ahead.

The AWS Serverless Application Repository is a collection of serverless applications published by developers, companies, and partners in the serverless community. It helps you easily discover, deploy, and publish serverless applications. For example, you can deploy published code samples, mobile backends, web backends, or complete applications to get started with AWS serverless compute in minutes. The repository enables sharing between developers, helps companies to connect with customers, and enable customers to move faster with the serverless ecosystem. You can share the applications privately within your AWS account (or organizations and departments) or publicly.

Each application is packaged with an AWS SAM template that defines the AWS resources used. Publicly shared applications also include a link to the application's source code. Currently, there is no additional charge to use the Serverless Application Repository—you only pay for the AWS resources used in the applications you deploy. You can also use the Serverless Application Repository to publish your own applications and share them within your team, other teams across your organization, or with the community at large.

The AWS Serverless Application Repository is currently available in preview. To deploy and publish applications in the Serverless Application Repository, you will need to sign up for the preview.

Summary

In this chapter, we used AWS services such as CloudFormation, Lambda, SAM, Cloud9, and CodeDeploy for deploying big data and serverless applications. We showed the use of the CloudFormation template and the process of authoring and deploying serverless applications.

In this book, we started our coverage of AWS cloud by describing basic cloud concepts. Then, we presented a detailed example, and focused our attention on design and implementation of typical non-functional requirements, including scalability, high-availability, and security. Finally, in the last three chapters, we shifted our focus to the design, implementation, and deployment of the next generation of applications being deployed on the cloud, including streaming applications, machine learning, and serverless applications.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Mastering AWS Lambda
Yohan Wadia, Udita Gupta

ISBN: 978-1-78646-769-0

- Understand the hype, significance, and business benefits of Serverless computing and applications
- Plunge into the Serverless world of AWS Lambda and master its core components and how it works
- Find out how to effectively and efficiently design, develop, and test Lambda functions using Node.js, along with some keen coding insights and best practices
- Explore best practices to effectively monitor and troubleshoot Serverless applications using AWS CloudWatch and other third-party services in the form of Datadog and Loggly
- Quickly design and develop Serverless applications by leveraging AWS Lambda, DynamoDB, and API Gateway using the Serverless Application Framework (SAF) and other AWS services such as Step Functions
- Explore a rich variety of real-world Serverless use cases with Lambda and see how you can apply it to your environments



AWS Automation Cookbook
Nikit Swaraj

ISBN: 978-1-78839-492-5

- Build a sample Maven and NodeJS Application using CodeBuild
- Deploy the application in EC2/Auto Scaling and see how CodePipeline helps you integrate AWS services
- Build a highly scalable and fault tolerant CI/CD pipeline
- Achieve the CI/CD of a microservice architecture application in AWS ECS using CodePipeline, CodeBuild, ECR, and CloudFormation
- Automate the provisioning of your infrastructure using CloudFormation and Ansible
- Automate daily tasks and audit compliance using AWS Lambda
- Deploy microservices applications on Kubernetes using Jenkins Pipeline 2.0

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

A

access controls
  controlling 210
  signed cookies 210
  signed URLs 210
access token 289
Amazon analytics-related services
  about 76
  Amazon Elastic MapReduce (EMR) 77
  Amazon Kinesis 77
Amazon Athena
  using 325, 326, 328, 329, 333, 341
Amazon CloudFront 74
Amazon CloudWatch
  about 74, 134
  using 134
Amazon compute-related services
  about 68
  Amazon EC2 68
  Amazon EC2 container service 69
Amazon database-related services
  about 70
  Amazon DynamoDB 71
  Amazon ElastiCache 72
  Amazon Redshift 71
  RDS 71
Amazon DynamoDB 71
Amazon EBS 70
Amazon EC2 68
Amazon EC2 container service 69
Amazon Elastic Load Balancing 73
Amazon Elastic MapReduce (EMR)
  about 77, 281
  auto scaling, using 279
  costs, lowering 279
  EC2 Spot, using 279
  use cases 277, 278
  using, best practices 276
Amazon ElastiCache 72
Amazon Glacier 70
Amazon Kinesis Analytics
  used, for analyzing data streaming 144
Amazon Kinesis Firehose
  using 320, 321, 322, 324
Amazon Kinesis Stream
  setting up 310, 312, 314
Amazon Kinesis
  about 77
  Data Analytics 274
  Data Firehose 275
  Data Streams 274
  used, for streaming data analysis 274
Amazon Machine Learning 77
Amazon management tools
  about 74
  AWS CloudFormation 74
  AWS CloudTrail 75
  CloudWatch 74
Amazon messaging-related services
  about 72
  Amazon Pinpoint 73
  Amazon SES 72
  Amazon SNS 72
  Amazon SQS 72
Amazon networking and content delivery services
  Amazon Virtual Private Cloud (Amazon VPC) 73
  Elastic Load Balancing 73
Amazon Pinpoint 73
Amazon Redshift 71
Amazon Relational Database Service (RDS) 71, 94
Amazon Route 53 74
Amazon S3 70, 281

Amazon S3 storage classes
  Amazon Glacier 88
  Amazon S3 Standard 88
  Amazon S3 Standard - Infrequent Access 88
Amazon SageMaker
  features 283
  key algorithm 283
  using 341, 344, 346, 347, 348
  using, for distributed machine learning 282
Amazon security, identity, and compliance services
  about 75
  AWS Certificate Manager 76
  AWS Directory Service 76
  AWS WAF 76
  IAM 75
  KMS 76
Amazon Simple Email Service (SES) 72
Amazon Simple Notification Service (SNS) 72
Amazon Simple Queue Service (Amazon SQS) 72
Amazon storage-related services
  about 69
  Amazon EBS 70
  Amazon Glacier 70
  Amazon S3 70
Amazon Web Services (AWS)
  about 15
  account, creating 16, 17, 19, 21
  account, setting up 15
  management console, exploring 21, 23, 24, 25
  reference link 78
  using, for disaster recovery 186
Apache Spark 48, 78
application development environment
  about 91
  production environments 92
  purpose 91
  QA/test environment 92
  staging environment 92
application security
  about 223
  data, securing at rest 230
  ELB, configuring for SSL 225, 228, 230
  self-signed certificates, generating 224
  transport security, implementing 224
architecture
  evolving, against loads 137
  half a million to a million users, scaling 139, 140
  million to ten million users, scaling 141
  one to half a million users, scaling from 137, 138, 139
Auto Scaling
  AMI, creating 146
  construction 146
  ELB, creating 149, 153, 156, 157, 158
  group, creating 164, 165, 166, 168, 170, 172, 174
  groups, scaling 176
  groups, testing 175
  launch configuration, configuration 158, 160, 162, 163
  setting up 146
availability objectives
  defining 178
availability zone (AZ) 190
Availability Zone Redundancy 182
Availability Zones 182
AWS API activity
  tracking, with CloudTrail 212
AWS Certificate Manager (ACM) 76, 209
AWS Cloud construction
  about 95
  EC2 Instance Key Pairs, creating 97
  EC2 Instance, creating 101, 109
  Elastic IPs (EIP), creating 109
  RDS, configuring 112, 116, 121, 123, 126
  roles, creating 98, 101
  Security Groups, creating 95
  software stack, installing 126, 129
  software stack, verifying 126, 129
AWS Cloud9
  about 374
  reference link 374
  used, for developing serverless applications 374
AWS cloud
  Amazon S3 storage, using 88
  auto scale, using 84, 86
  AWS services, using 90
  cost lowering, strategies 81
  costs, analyzing 81
  costs, managing 79

costs, monitoring 81
costs, optimizing 79, 80, 89
costs-related objectives, setting 79
database utilization, optimizing 89
EC2 Instance, selecting 82, 84
queues, using 91
reserved instances, using 86
spot instances, using 87, 88
unused instances, turning off 84
AWS CloudFormation 74
AWS CloudFront
using, for content distribution 134
AWS CloudTrail 75, 212
AWS CloudWatch
reference link 104
AWS components
about 68
AI-related service 77
Amazon analytics-related services 76
Amazon compute-related services 68
Amazon database-related services 70
Amazon machine learning service 77
Amazon management tools 74
Amazon messaging-related services 72
Amazon security, identity, and compliance services 75
amazon services 78
Amazon storage-related services 69
AWS networking and content delivery services 73
used, in big data applications 274
AWS Direct Connect 74
AWS Directory Service 76
AWS ELB
used, for scaling without service interruptions 134
AWS environments
creating, with CloudFormation 238
managing, with CloudFormation 238
AWS Glue
about 277, 281
crawlers 277
Data Catalog 277
ETL service 277
using 325, 326, 328, 329, 333, 341
AWS high availability architecture
about 189
AZ 190
EC2 instances 191
ELB 191
RDS 191
S3 191
VPC 192
AWS Identity and Access Management (IAM)
about 75, 214
roles 215
used, for securing infrastructure 214
AWS infrastructure services
Amazon CloudWatch, using 134
AWS CloudFront, using for content distribution 134
data services, scaling 135
EC2 container service, using 136
ELB, used for scaling without service interruptions 134
leveraging, for scalability 133
scaling 136
AWS infrastructure
AWS Cloud construction 95
Cloud deployment architecture 93
setting up 92
AWS Key Management Service (KMS)
about 76
issues, managing 217
keys, creating 218, 219, 220, 222
keys, using 222
using 217
AWS Lambda function
creating 314, 316, 317, 319
AWS Lambda
about 69, 142
using 142
AWS management tools
reference link 75
AWS networking and content delivery services
about 73
Amazon CloudFront 74
Amazon Route 53 74
Amazon VPC 73
AWS Direct Connect 74

AWS production deployment architecture
  about 250
  bastion host 258
  bastion subnet 258
  private subnets 252, 254, 256
  security groups 259, 261
  VPC subnets 252
AWS SAM
  about 372
  SAM Local 373
  SAM template 372
AWS Security Token Service
  reference link 207
AWS security
  AWS API activity, tracking with CloudTrail 212
  considerations, while using CloudFront 208
  Identity Lifecycle Management, implementing 211
  implementing, best practices 206
  security analysis, logging 212
  security configuration, auditing 213
  security configuration, reviewing 213
  third-party security solutions, using 212
AWS serverless application repository
  using 376, 377
AWS serverless options
  examples 275
AWS services and products
  reference link 93
AWS services
  URL 15
  used, for implementing large-scale API-based architecture 142
  using, for out-of-the-box scalability 132
AWS solutions
  used, for archiving 247, 248
  using, for backup 247, 248
AWS WAF 76

B

bastion host 258
big data applications
  about 272
  AWS components, used 274
  EMR cluster, securing 284
  security overview 284
  serverless applications, securing 286
Business Process Execution Language (BPEL) 38

C

Certification Authorities (CAs) 224
Classless Inter-Domain Routing (CIDR) 255
cloud applications design principles
  about 39
  eventual consistency, designing for 47
  failure, designing for 43, 44
  parallel processing, designing for 44
  performance, designing for 45
  scale, designing 39, 40, 42
cloud computing
  about 7
  costs, estimating 53, 55
cloud infrastructure
  automating 42
cloud-based machine learning models
  deploying 53
cloud-based machine learning pipelines
  deploying 51, 53
cloud-based multitier architecture 28, 29
cloud-based service models
  IaaS 9
  PaaS 9
  SaaS 9
cloud-based workloads
  about 13
  on-premise applications, migrating to cloud 13
cloud-native applications
  about 15
  building 15
CloudFormation templates
  used, for creating data lake 350, 353, 355, 358, 360
  using 350
CloudFormation
  extending 246
  setting up 261, 264, 266
  stacks, updating 242, 244, 245
  templates, creating 240
  used, for building DevOps pipeline 241
  used, for creating AWS environments 238

used, for managing AWS environments 238
CloudFront
about 209
access control options 210
application, securing 211
features 208
options, for TLS 209
using, for AWS security considerations 208
Web Application Firewall 210
CloudTrail
AWS API activity, tracking 212
CloudWatch
setting up 269, 270
using, for monitoring 246
Content Delivery Network (CDN) 45
content
distribution, with AWS Cloud 134
Cross-Origin Resource Sharing (CORS) 88

D

data extensibility
requisites, addressing 34, 38
data lake
creating, with CloudFormation template 350, 353, 355, 358, 360
reference link 350
data security
requisites, addressing 32, 34
data
S3 console, using for server-side encryption 230
securing, on RDS 235
securing, on S3 230
dataset
reference link 325
deployments, at scale
managing 237
dequeue operation 40
development environment
application structure 64
application, executing 61, 63
setting up 58
war file, building for deployment 63
DevOps pipeline
building, with CloudFormation 241
disaster recovery (DR)
about 177
AWS, using for 186
backup strategy, using 187
Multi-Site architecture, using 188
Pilot Light architecture, using 187
restore strategy, using 187
strategy, testing 188
warm standby architecture, using 187
distributed machine learning
Amazon SageMaker, using 282
best practices 280
data wrangling 282
deployment 282
experimentation 282

E

e-commerce web application
about 56
development environment, setting up 58
functional requisites 56
non-functional requisites 57
EC2 instance
about 94, 191
reference link 94
Eclipse with Maven plugin (m2e)
about 58
URL, for installing 58
Eclipse
about 58
reference link 58
Elastic Load Balancer (ELB)
about 149, 191, 224
control service 149
HA support 194
load balancer 149
using, for high availability 180
Elasticsearch 143
emerging cloud-based application architectures
about 47
cloud-based machine learning models, deploying 53
cloud-based machine learning pipelines, deploying 51, 53
Kappa architecture 51
polyglot persistence 49, 50

EMFRS
  reference link 285
EMR cluster
  authentication 285
  authorization 285
  encryption 284
  securing 284
EMR step (EMR API) 285
EMR-Spark clusters
  configuring 290, 292, 298, 300, 305, 307
  using 290, 292, 298, 300, 305, 307
enhanced networking, Linux
  reference link 83
enqueue operations 40
event handling
  at scale 141
  large-scale API-based architecture, with AWS services 142
  real-time applications, building with Amazon Kinesis Analytics 145
  streaming data, analyzing with Amazon Kinesis Analytics 144

F

failures
  AWS, using fro disaster recovery 186
  DR strategy, testing 188
  high availability, setting for application 183
  high availability, setting for data layers 183
  nature 179
  Route 53, using for high availability 180
  VPC, setting for high availability 179

G

Git command line tools
  about 58
  URL, for downloading 58
Glue Crawlers 281

H

high availability
  auto scaling 181
  implementing, in application 185
  instance 180
  setting up 189
  setting, for application 183
  setting, for data layers 183
  VPC, setting up 179
hybrid cloud 9

I

Identity and Access Management (IAM) 206
Identity Lifecycle Management
  implementing 211
identity token 289
Information Security Management System (ISMS) 204
Infrastructure as a service (IaaS)
  about 6, 9
  features 9
Infrastructure as Code (IAC) 206
infrastructure, at scale
  managing 237
Internet gateway 94
Internet of Things (IoT) 185

J

JDK 1.8
  about 58
  URL, for downloading 58
JSON file
  Resources section 262

K

Kappa architecture 51
Kerberos
  reference link 285
Kinesis Firehose 145
Kinesis Streams
  about 143
  using 143

L

Lambda architecture 50
large-scale API-based architecture
  Amazon API Gateway, using 142
  AWS Lambda, using 142
  Elasticsearch, using 143

implementing, with AWS services 142
Kinesis Streams, using 143
launch configuration 158

M

Maven 3
about 58
URL, for downloading 58
metadata 216
Model-View-Controller (MVC) architecture
about 64
controller 65
model 64
view 65
multi-tenancy
about 10
benefit 30
data extensibility, requisites addressing 34, 38
data security, requisites addressing 32, 34
designing 30, 32

N

natural language understanding (NLU) 78

O

OCSP stapling 208
on-premise applications
migrating, to cloud 13
Origin Access Identity (OAI) 211

P

parallel processing
designing 44
Platform as a service (PaaS)
about 6, 9
features 9
polyglot persistence 49
predictive analytics
best practices 280
principle of least privileges
reference link 215
private cloud 8
production environments 92
production
AWS production deployment architecture 250
centralized logging 268
CloudWatch, setting up 269, 270
go-live activities, planning 249
infrastructure, using as code 261
setting up 250
public cloud 8

Q

QA/test environment 92

R

real-time applications
building 145
refresh token 289
region 93
regional redundancy 182
Relational Database Service (RDS)
about 191
HA support 197, 200, 202
Route 53
region availability 182
regional redundancy 182
router 94

S

SAM Local 373
scalability
AWS infrastructure services, leveraging 133
objectives, defining 130
scalable application architectures
asynchronous process, implementing 133
AWS services, using for out-of-the-box scalability 132
designing 131
loosely-coupled components, implementing 132
scale-out approach, using 132
security analysis
logging 212
security configuration
auditing 213
reviewing 213
security group 94
security

application security 223
AWS IAM, used for securing infrastructure 214
AWS Key Management Service, using 217
objectives, defining 204
responsibilities 205
setting up 214
serverless application developer tooling
reference link 375
serverless applications
authentication 287
authoring 360, 362, 364, 367, 369
authorization 287
AWS SAM 372
deploying 360, 362, 364, 367, 369
deployment, automating 374, 375
developing, with AWS Cloud9 374
reference link 290
serverless big data applications
best practices 275
Service-Oriented Architecture (SOA) 133
sharding 39
Shared Application architecture 12
Shared Everything architecture 12
Shared Nothing architecture 11
Simple Storage Service (S3) 191
Software as a service (SaaS)
about 6, 9
feature 9
Spark Web UIs
reference link 307
Spring Tool Suite (STS)
about 58
URL, for downloading 58
SSL termination
full bridge termination 210
half bridge termination 209
implementing 209
stacks
updating 242, 243, 245
staging environment 92
streaming application
about 15
actionable insights 273
components 272
continuous metric generation 273
data consumer 273
data producer 272
ingest-transform-load 273
patterns 273
service 272
streaming data
Amazon Kinesis Analytics, using 145
Amazon Kinesis Firehose, using 145
analyzing, with Amazon Kinesis 274
analyzing, with Amazon Kinesis Analytics 144
subnets 94
support, at scale
managing 237

T

third-party security solutions
using 212

V

Virtual Private Cloud (VPC)
about 94, 192
setting, for high availability 179

W

Web Application Firewall (WAF)
about 205, 210
reference link 210