Professional Expertise Distilled

Learning AWS

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

Aurobindo Sarkar Amit Shah

[PACKT] PUBLISHING enterprise professional expertise distilled

Learning AWS

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

Aurobindo Sarkar

Amit Shah

[PACKT] PUBLISHING enterprise professional expertise distilled

BIRMINGHAM - MUMBAI

Learning AWS

First published: July 2015

Production reference: 1280715

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

ISBN 978-1-78439-463-9

www.packtpub.com

Credits

Authors
Aurobindo Sarkar
Amit Shah

Reviewers
Jiří Činčura
Brian C. Galura
Mark Takacs
Robert Williamson

Commissioning Editor
Edward Bowkett

Acquisition Editor
Larissa Pinto

Content Development Editor
Anish Sukumaran

Technical Editor
Bharat Patil

Copy Editor
Merilyn Pereira

Project Coordinator
Mary Alex

Proofreader
Safis Editing

Indexer
Tejal Soni

Graphics
Sheetal Aute

Production Coordinator
Manu Joseph

Cover Work
Manu Joseph

About the Authors

Aurobindo Sarkar is a consulting CTO at BYOF Studios. With a career spanning 22 plus years, he has consulted at some of the leading organizations in the U.S., the UK, and Canada. He specializes in Software as a Service product development, cloud computing, cloud economics, big data analytics, Internet of Things (IoT) platforms, and web-scale architectures. His domain expertise runs across financial services, media, mobile gaming, public and automotive sectors. Aurobindo has been actively working with technology start-ups for over 5 years now. As a member of the top leadership team at various start-ups, he has mentored several founders and CxOs, provided technology advisory services, developed cloud strategies, drawn up product roadmaps, and set up large engineering teams. Aurobindo has an MS (Computer Science) from New York University, M.Tech (Management) from Indian Institute of Science, and B.Tech (Engineering) from IIT Delhi.

Amit Shah has a bachelor's degree in electronics. He is a senior manager at Western Outdoor Interactive. He has been programming since the early '80s with the first wave of personal computing—initially as a hobbyist and then as a professional. His areas of interest include embedded systems, Internet of Things (IoT), analog and digital hardware design, systems programming, cloud computing, and enterprise architecture. He has been working extensively in the field of cloud computing and enterprise architecture for the past 4 years.

About the Reviewers

Jiří Činčura is an independent developer focused on clean code, language constructs, and databases, all applied to mostly backend-related stuff. He is a project lead for ADO.NET, a provider for the Firebird project, and the Entity framework support maintainer for the NuoDB database. He's currently creating custom applications for a living for various customers in Europe as well as companies from other countries. Other than that, he conducts training sessions and consultations about new technologies to provide customers with the best information possible to deliver applications in a shorter time and with better maintainability. When he's not programming or teaching, he spends time participating in ultrarunning races.

Brian C. Galura spent his childhood tinkering with subjects such as Java programming and Linux. His professional experience started with VoIP testing at 3Com in suburban Chicago. He then spent 2 years studying computer engineering at Purdue University, before leaving to pursue freelance consulting in Los Angeles. Following several years of freelancing, he developed his expertise in enterprise infrastructure and cloud computing by working for a variety of start-ups and large corporations. Later, he completed a bachelor's in IT while working at Citrix. He is currently working on Citrix's cloud engineering and systems architecture team in Santa Barbara, California.

Mark Takacs got his first job in the early '90s as the only applicant with HTML experience. Since then, his road to DevOps has spanned traditional MVC software development on LAMP and Java, frontend web development in JavaScript, HTML, CSS, network administration, build and release engineering, production operations, and a large helping of system administration throughout. He currently lives and works in Silicon Valley.

Robert Williamson is a senior software engineer working mainly on the development of a graphics engine for flight simulators; he enjoys designing and developing real-time solutions. Before that, he worked in the Computer Graphics and Image Understanding Lab at the University of Missouri Columbia while earning his bachelor's and master's in computer science. His other interests include data visualization, data mining, computer vision, and machine learning. During his spare time, he enjoys traveling and backpacking through national parks.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.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.

`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Instant updates on new Packt books

Get notified! Find out when new books are published by following `@PacktEnterprise` on Twitter or the *Packt Enterprise* Facebook page.

Table of Contents

Preface

With an increasing interest in leveraging cloud infrastructure around the world, the AWS cloud from Amazon offers a cutting-edge platform for architecting, building, and deploying web-scale cloud applications through a user-friendly interface. The variety of features available within AWS can reduce overall infrastructure costs and accelerate the development process for both large enterprises and start-ups alike.

Learning AWS covers basic, intermediate, and advanced features and concepts as they relate to designing, developing, and deploying scalable, highly available, and secure applications on the AWS platform. By sequentially working through the steps in each chapter, you will quickly master the features of AWS to create an enterprise-grade cloud application. We use a three-tiered services-oriented sample application for extensive hands-on exercises.

This book will not only teach you about the AWS components, but you will gain valuable information about key concepts such as multi-tenancy, auto scaling, planning, implementing application development environments, addressing application security concerns, and so on. You will also learn how these concepts relate to cost effective architectural decisions while designing scalable, highly available, and secure AWS cloud applications.

A step-by-step approach to cloud application design and implementation is explained in a conversational and easy-to-follow style. Each topic is explained sequentially in the process of creating an AWS cloud application. Detailed explanations of the basic and advanced features of AWS that address the needs of readers with a wide range of real-world experiences are also included. Expert programmers and architects will appreciate the focus on the *practice* rather than the *theory*.

What this book covers

Chapter 1, *Cloud 101 – Understanding the Basics*, describes basic cloud concepts including the public, private, and hybrid cloud models. We explain and compare the Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS) cloud service delivery models. In addition, we explain multi-tenancy models and the challenges they present in design, implementation, and operations.

Chapter 2, *Designing Cloud Applications – An Architect's Perspective*, describes familiar and not-so familiar architectural best practices in the cloud context. These include designing a multi-tier architecture and designing for multi-tenancy, scalability, and availability. We will also guide you through the process of estimating your cloud computing costs.

Chapter 3, *AWS Components, Cost Model, and Application Development Environments*, introduces you to the AWS components—EC2, S3, RDS, DynamoDB, SQS Queues, SNS, and so on. We will discuss strategies to lower your AWS infrastructure costs and their implications on architectural decisions. We will explain 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 your availability objectives, discuss the nature of failures, and then discuss the design and implementation of specific strategies to achieve high availability. In addition, we will describe the approaches that leverage the AWS features and services for your Disaster Recovery planning.

Chapter 6, *Designing for and Implementing Security*, provides guidance on how to define security objectives, explains your 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, we provide some tips on planning your production Go-Live activities.

What you need for this book

You will need your standard development machine with Spring Tool Suite (STS), Maven, the Git command line tools, the MySQL database, and an Amazon account to complete the hands-on sections in this book.

Who this book is for

This book is targeted at expert programmers/architects, who want to learn AWS. This book assumes that the reader has previously designed and developed multi-tiered applications, not necessarily on a cloud platform, but in an enterprise or a start-up setting. Some familiarity with Spring, MySQL, and RESTful web services is expected.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "This can be done via the command line from the root of the `a1ecommerce` project via a maven goal package."

A block of code is set as follows:

```
jdbc.url=jdbc:mysql:// a1ecommerce.cklrz1a88gdv.us-east-
  1.rds.amazonaws.com:3306/
    a1ecommerceDba1ecommerceDba1ecommerceDb #Endpoint of
      AmazonAmazonAmazon RDS
jdbc.username=a1dbroot # username of Amazon DB instance
jdbc.password=a1dbroot #Password for the Amazon DB instance
```

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

```
sudo apt-get update;
sudo apt-get install tomcat7 mysql-client-5.6;
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "From the RDS dashboard, click on **Instances** in the navigation pane, and then on **a1ecommerce** to view the details of the DB instance."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

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**).

To help you get started on **Amazon Web Services** (**AWS**), we will end the chapter by walking you through the 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
- Steps to create an AWS account
- A brief overview of the AWS management console

What is cloud computing?

Wikipedia defines cloud computing as:

> "*Cloud computing is 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:

> "*Cloud computing is 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, and still others drop the requirement of access via the internet. We define cloud computing as a model that enables the features listed here:

- Users should be able to provision and release resources on-demand
- The resources can be scaled up or 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 as follows:

- The illusion of infinite processing and storage resources, available on-demand, reduces 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.
- The development and test environments can be provisioned on a smaller scale than production environment, and enabled only during normal business hours, to reduce costs.

- 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) for improved defect resolution.
- There will be ease of scaling, both vertically and horizontally, in order to better manage spikes in demand and variations due to business cycles or time-of-day reasons, and so on.
- This encourages experimentation, by trying out new ideas and software by quickly provisioning resources, rather than requisition for resources through time-consuming and cumbersome processes.

In addition, there are several key operational and maintenance-related implications, including no hardware maintenance or data center operations required, zero-downtime migrations and upgrades, ease of replacement of unhealthy machines, ease of implementation of high-availability and disaster recovery strategies, and many more.

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

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 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, private cloud places increase administrative overheads on the customer but give greater control over the infrastructure and reduce 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 several factors, including business criticality of the application, sensitivity of the data, industry certifications and standards required, regulations, and many more. But in some cases, spikes in demand for resources are also handled in the public cloud.

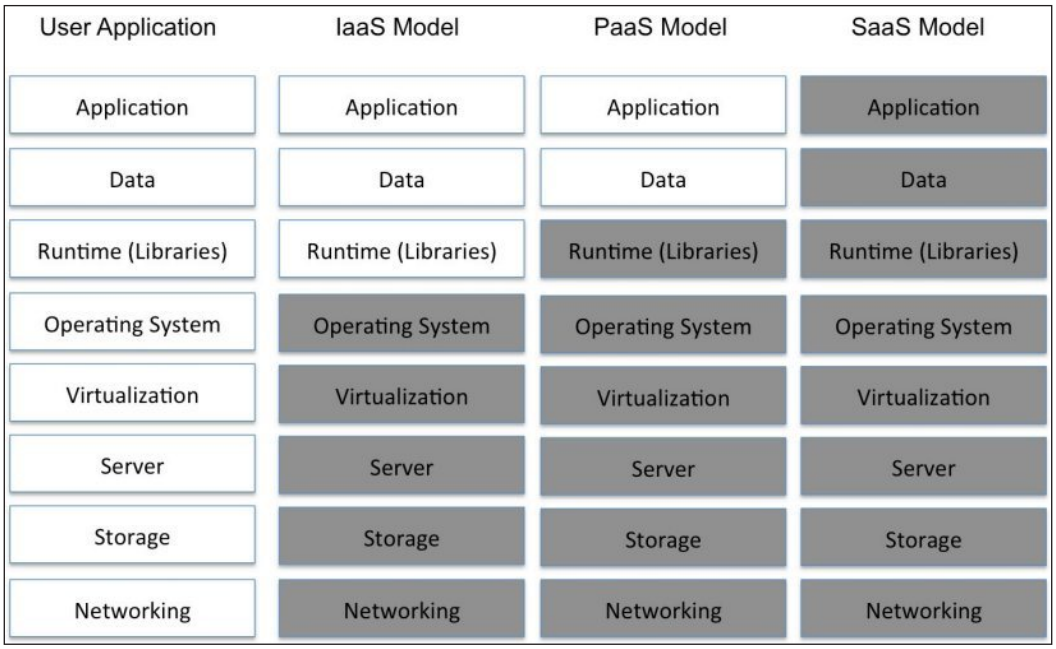
Cloud service models – IaaS, PaaS, and SaaS

There are three cloud-based service models, IaaS, PaaS, and SaaS. The main features of each of these are listed here:

- Infrastructure as a Service (IaaS) provides users the capability 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 and the virtual server provisioning models (typically offered by data center outsourcers). The onus of administering these resources rests largely with the customer.
- In Platform as a Service(PaaS), the service provider makes certain core components, such as databases, queues, workflow engines, e-mails, and so on, which are 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's functionality. However, this model also leads to application-level dependency on the providers' services.
- In the Software as a Service(SaaS) model, typically, third-party providers using a subscription model provide end-user applications to their customers. The customers might 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 many more. Applications that have a very wide user base most often operate in a self-service mode. In contrast, the provider provisions the application for the customer for more specialized applications. The provider also hands over certain application administrative tasks to the customer's application administrator (in most cases, this is limited to creating new users, managing passwords, and so on through well-defined application interfaces).

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

The following diagram illustrates who is responsible for managing the various components of a typical user application across IaaS, PaaS, and SaaS cloud service models. The column labeled **User Application** represents the main components of a user application stack, while the following columns depict the varying levels of management responsibilities in each of the three service models. The shaded boxes are managed by the service provider, while the unshaded boxes are managed by the user.



The level of control 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.

We would like to conclude our introduction to cloud computing by getting you started on AWS, right away. The next two sections 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 that can be used to learn the different components of AWS. With this account, you get access to 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`.

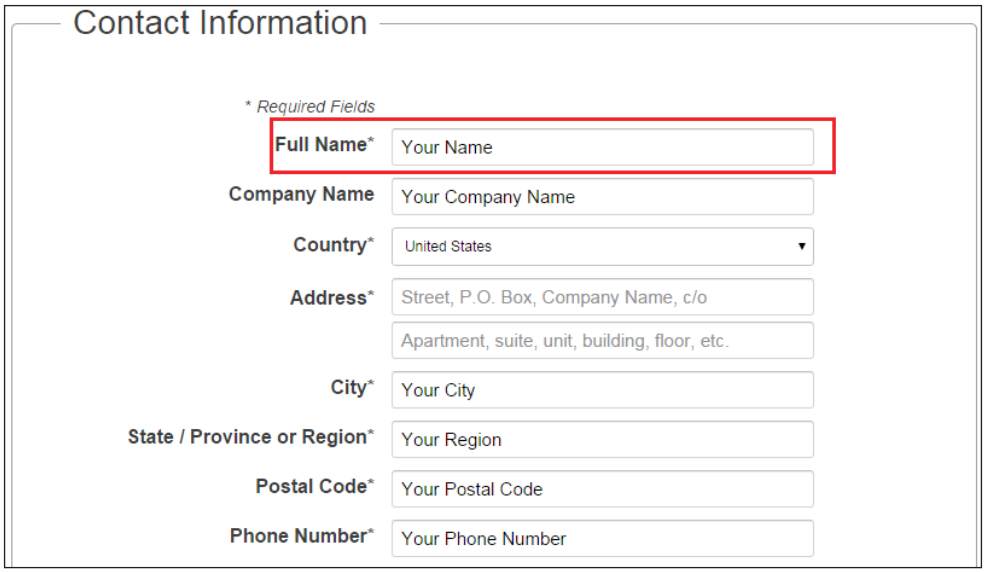
We are assuming that you do not have a pre-existing AWS account with Amazon (if you do, please feel free to skip this section). Perform the following steps:

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

   The process to create a brand new AWS account has started. You can sign in using your existing Amazon retail account, but you will have to go through the process of creating an AWS account; the two accounts are different for accounting purposes, even though they share the same common login. Let's take a look at the following screenshot:

2. After creating a new account or using your existing retail Amazon account, select the **I am a returning user and my password is:** option and click on **Sign in using our secure server**. A set of intuitive screens will guide you through multiple screens in order to to create an AWS account, these include:

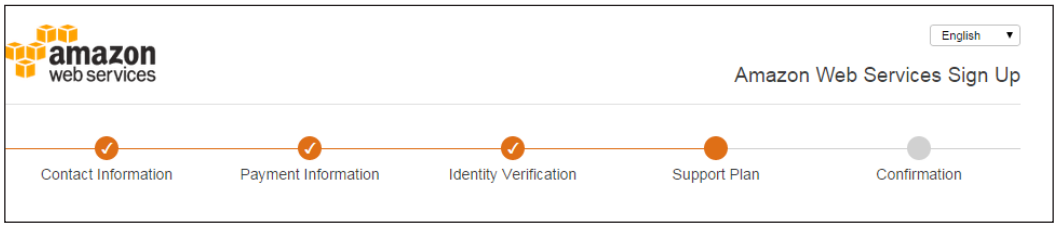
- **Contact Information**: Amazon also uses this information for billing and invoicing. The **Full Name** field is also used by the AWS management console to identify your account, as shown in the following screenshot:



- **Payment Information**: When you create an AWS account and sign up for services you are required to enter payment information. Amazon executes a minimal amount transaction against the card on file to confirm that it is valid and not reported lost or stolen. This is not an actual charge it merely places the 'X' amount on hold on the card which will eventually drop off. The 'X' amount depends on the country of origin.
- **Identity Verification**: Amazon does a call back via an automated system to verify your telephone number.
- **Support Plan**: You can subscribe to one from the following, **Basic**, **Developer**, **Business**, or **Enterprise**. We recommend subscribing to the Basic plan to start with.

The Basic plan costs nothing, but is severely limited and hence not recommended for production. It is an excellent way to learn and get familiar with AWS.

- **Confirmation**: On clicking on **Launch Management Console** you will be requested to login, as shown in the following screenshot:



3. At this stage, you have successfully created an AWS account, and you are ready to start using the services offered by AWS.

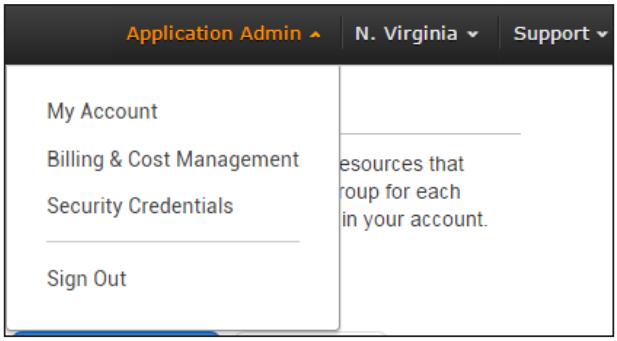
The AWS management console

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

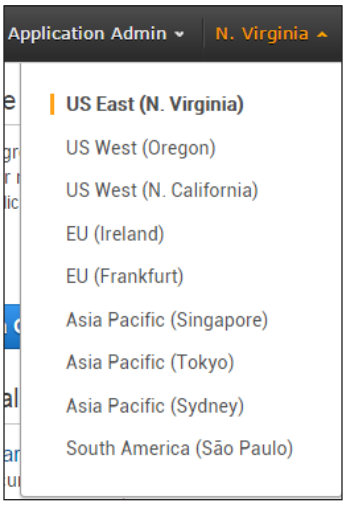
- **Amazon Web Services**: This is a dashboard view that lists all the AWS services currently available in a specific Amazon region. Clicking on any one of these launches the dashboard for the selected service, as shown in the following screenshot:

- **Shortcuts for Amazon Web Services**: On the console management screen, you can create shortcuts of frequently accessed services via the **Edit** option, as shown in the following screenshot:

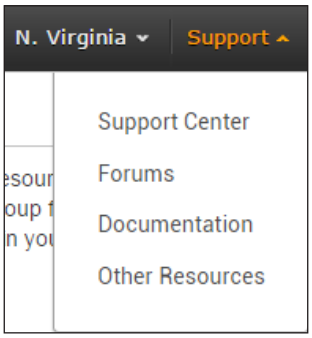
- **Account related information**: This allows you to access your account-related data. This includes security credentials needed to access the AWS resources by your application. The **Billing & Cost Management** option gives you real-time information on your current month's billing; this helps in managing costs, as shown in the following screenshot:



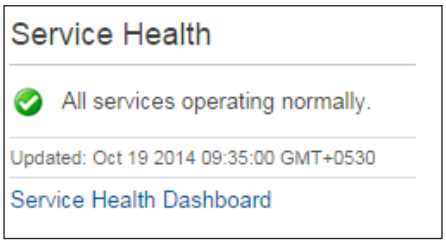
- **Amazon regions**: This option allows you to access the AWS in a specific region. In the following screenshot, all the Amazon Web Services are located in the **US East (N. Virginia)** region:



- **Support**: You can navigate to the **Help**, **Forums**, and **support** pages:



- **Service Health**: Launches the health dashboard of all the Amazon Web Services across all regions, and not of your deployed service:



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. 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 the next chapter we will deep dive into the details of how multitenanted 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 – An Architect's Perspective

As an architect, we are sure you have come across terms such as loosely coupled, multi-tier, services oriented, highly scalable, and many more. These terms are associated with architectural best practices and you 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 approach you take towards cloud application architecture remains the same to a large extent. However, you need to be aware of certain peculiarities of 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 automatically scale up and down. What are the implications of auto scaling on your design?

One of the major differences in cloud-based SaaS applications and on-premise enterprise applications is multi-tenancy. What are some of the design considerations of multi-tenancy? How do you design for UI, services, and data multi-tenancy in a multi-tier architecture?

In this chapter, we describe the familiar and not-so familiar architectural best practices in the cloud context, by covering the following topics:

- Multi-tier architecture
- 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
- Estimating your cloud computing costs
- Sample application is a typical e-commerce web application

Multi-tier architecture

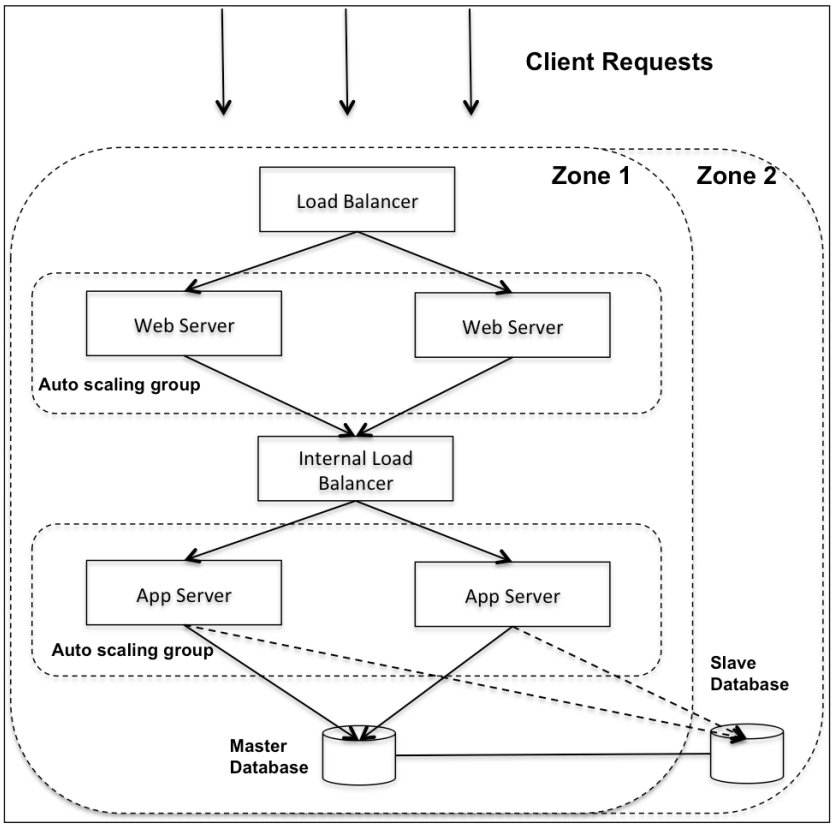
A simple three-tier architecture consists of a UI tier, an application or business tier, and a data tier.

These tiers are ordinarily implemented using web servers, application servers, and databases, respectively.

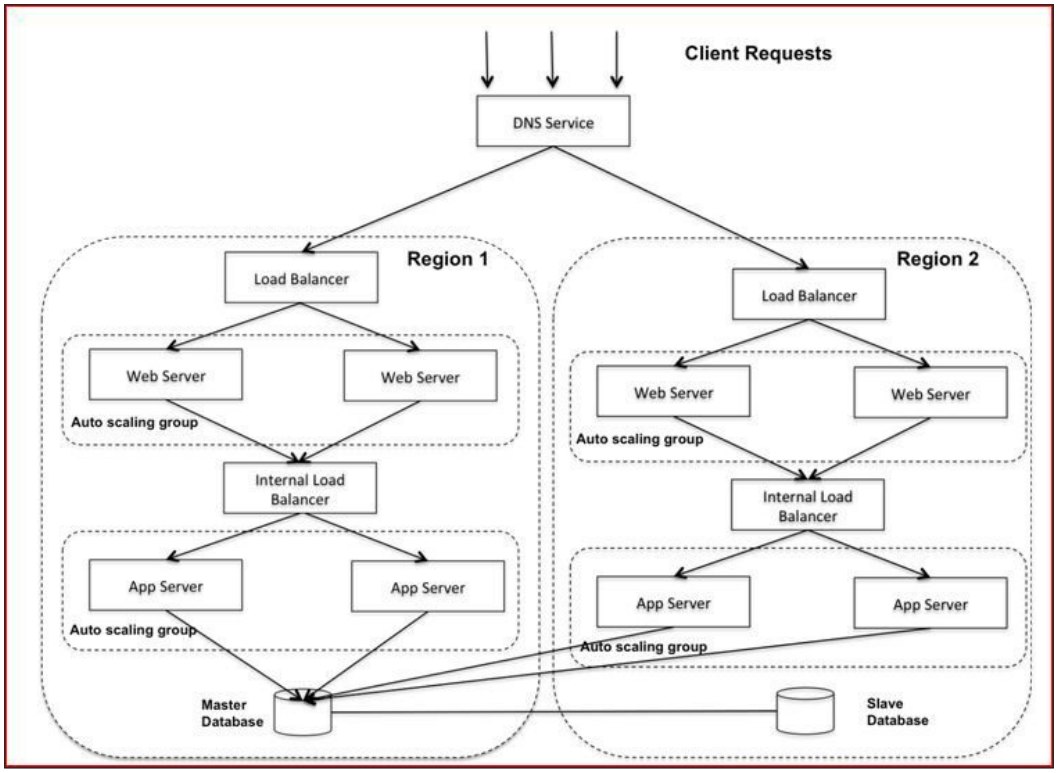
Cloud applications can be deployed at multiple locations. Typically, these locations are regions (that is, separate geographical areas) or zones (that is, distinct locations within a region connected by low latency networks).

This tiered architecture on the cloud 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.

Let's take a look at the following diagram:



It is also possible to separate the tiers across two different regions, to provide for higher level of redundancy including data center wide or zone level failures. While designing high availability architectures across multiple regions, we need to address network traffic flow and data synchronization issues between the regions. Such issues are discussed in more detail in *Chapter 5*, *Designing for and Implementing High Availability*. The following diagram illustrates this architecture:



Designing for multi-tenancy

The major benefit of multi-tenancy is cost saving due to infrastructure sharing and the operational efficiency of managing a single instance of the application across multiple customers or tenants. However, multi-tenancy introduces complexity. Issues can arise when a tenant's action or usage affects the performance and availability of the application for other tenants on the shared infrastructure. In addition, security, customization, upgrades, recovery, and many more requirements of one tenant can create issues for other tenants as well.

Multi-tenancy models may lie anywhere from the share-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 approach to 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 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 and will lead to higher infrastructure costs.

Typically, you would offer this approach to your bigger customers who might 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. Such customers might also have a higher need for customizations.

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 many more. If any of these parameters are heavily skewed in favor of a particular tenant, then it might be advisable to isolate their data.

We can define a separate database schema for each of the tenants (within the same database server instance) for applications having a limited number of database tables. 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, but this approach can reduce costs while separating out the data of each tenant.

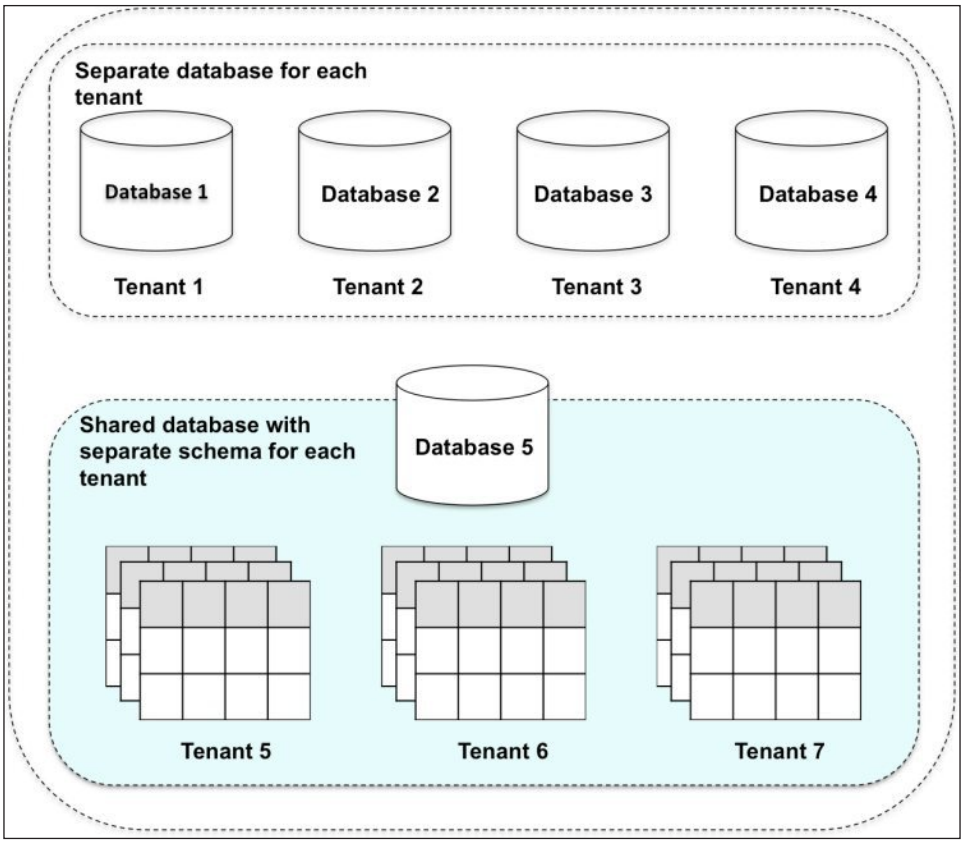
In a shared database, with a shared schema approach, the costs are minimized, but the complexity of the application is much higher. This model works well for cost conscious customers. However, restoring a customer's data is complicated, as you will be restoring specific rows belonging to a specific tenant. This operation can impact all other tenants using the shared database.

In cloud architectures, the main factors to consider while designing multi tenancies are the security, extensibility, and scalability. In addition, multi-tenancy brings additional complexity from a DevOps perspective, and we need to ensure that we are able to effectively manage upgrades and troubleshoot, bugs and maintain high service levels and operations' efficiency.

Data security

There are two levels of security to be considered—at the tenant level (typically, an organization) and the end-user level, who is a member or an employee 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 can 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.

When multi-tenancy is realized by having separate databases or separate schemas per tenant, you can restrict access at 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 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). 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. There are several options available from the cloud service provider and third-party vendors for implementing encryption to protect your data. These range from manual ones implemented on the client-side to fully automated solutions. 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 nonauthorized user accesses it. This is more critical for shared database/schema model. 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. It is therefore 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.

Data extensibility

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 customizations to the database schema. You must ensure that you don't change your schema for a tenant so much that your product no longer fits into the SaaS model. But you do want to bake in sufficient flexibility and extensibility to handle custom data requirements of your customers (without impacting subsequent product upgrades or patch releases).

One approach to achieve extensibility in the database schema is to preallocate 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 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 defined 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 metadata table that contains the field label, data type, and such information. This approach cuts out potential waste and does not limit the number of fields available for customization as in the first approach, but is obviously more complicated to implement.

tbl_customers

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

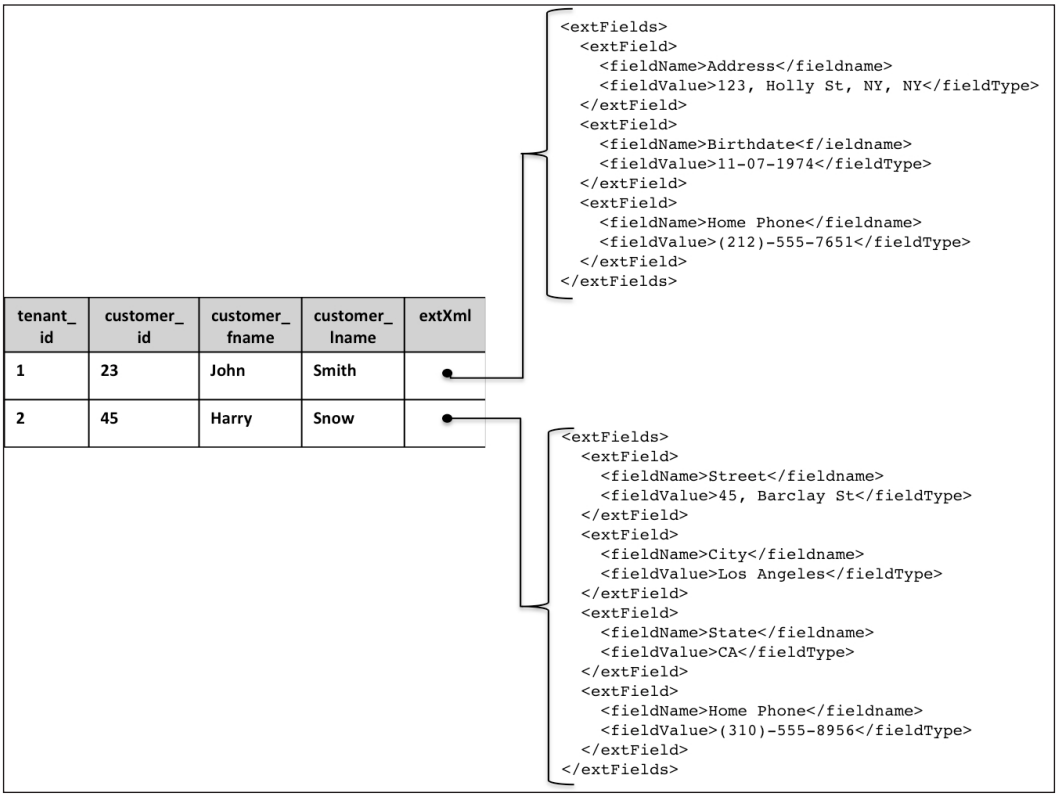
tbl_customers_metadata_link_values

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 the preceding 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, as shown in the following figure:



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 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 tenants, as this can have a ripple effect on the application code and the user interface.

In this section, we have primarily covered multi-tenant approaches for relational databases. Depending on your particular application requirements, for instance, type and volume of data, and types of database operations, a NoSQL database can be a good data storage solution. NoSQL databases use nontabular structures, such as key-value pairs, graphs, documents, and so on to store data. The design techniques in such cases would depend on your choice of NoSQL database.

Application multi-tenancy

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 schemas. To ensure 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 frontend or the UI is also 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 high-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 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.

Other application multi-tenancy-related issues include tenant-specific notifications, new tenant provisioning and decommissioning, and so on.

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 reach a certain threshold on CPU, memory, disk (capacity and throughput), or network bandwidth, you will repeat the exercise to handle 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 number of 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.

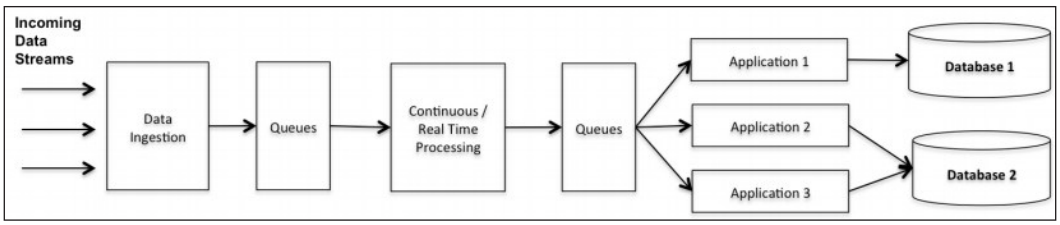
Typically, cloud applications are designed to be horizontally scalable. In most cases, the application services or business tier is specifically designed to be stateless, so that compute nodes can be added or deleted with no impact to the functioning of the application. If the application state is important, then it can be stored externally using the caching or the storage service. Depending on the application, things such as 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 off by the cloud provider.

An architecture comprising of loosely coupled components is a well-accepted approach and the best practice. This is especially true while building highly scalable systems. Loose coupling allows you to distribute your components and scale them independently. In addition, loose coupling allows parts of your system to go down without bringing the whole system down. This can improve the overall availability of your application.

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 is distributed and your messages automatically replicated across several nodes. In addition, one of these nodes may be unavailable, or fails, when your request arrives, or during the processing of your request.

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

- 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 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, even if the queue contains messages. 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 on an online shopping site might 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 be easily enabled through an API or a UI.
- In a cloud queue service, it is important to differentiate between a dequeue and 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 time out value is usually configurable. What happens if 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 time out 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 count.

The number of messages in your queue is a good metric to use for auto scaling your processing tier. It is also a great trigger to raise alerts to your operations team.

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

Automating 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, centrally storing application configuration parameters, and booting new instances quickly by bootstrapping your instances. In addition, you will need to continuously monitor your system metrics to dynamically take actions such as auto scaling.

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.

There are several tools and systems available from Amazon and other third-party providers that can help you automate your infrastructure. These include OpWorks, Puppet, Chef, and Docker.

Designing for failure

Assuming 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 loads getting back to normal levels after a spike. Nodes might be rebooted by the cloud platform. There can also be unexpected application failures. In all cases, the design goal should be to handle such error conditions gracefully and minimize any impact to the user experience.

There should be a strong preference to minimize human or manual intervention. Hence, it is preferred to implement strategies using services made available by the cloud platform to reduce the chances of failures or automate recovery from such failures. For example, you can use AWS CloudFormation to install, configure, and start applications on Amazon EC2 instances. AWS CloudFormation includes a set of helper scripts based on `cloud-init`. You can call these scripts from your AWS CloudFormation templates to automate installation, configuration, and updating of your applications on EC2 instances.

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

- Do not store application state on your servers because if your server gets killed then you will lose any application state. Sessions should never be stored to local filesystems. 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 file system.
- 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 there are several good storage options available. For instance, based on your application-specific requirements, the cloud platform's reliable queuing service can be a good choice.
- 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. In addition, it is often useful to include application-specific sequence of calls or requests up to the point of failure, to help trace the source of the problem. 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 might 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 simple.

Remember to log your retry attempts and raise operator alerts if the threshold on the number of retry attempts is crossed. 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 might 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 long polling.
- Avoid single points of failure. Plan to distribute your services across multiple regions and zones (that is, different data centers in the same region), and also implement a robust failover strategy. This will minimize the chances of an application outage due to individual instances, availability zone, or 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 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 downtown.

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 design for concurrency throughout your architecture, from data ingestion to its processing. So use multithreading to parallelize your cloud service requests, distribute load using load balancing, ensure multiple processing components or service endpoints are available via horizontal scaling, and so on.

As a best practice, you should exploit both multithreading and multi-node processing features in your designs. For example, using multiple concurrent threads for fetching objects from a cloud data storage service is a lot faster than fetching them sequentially. In the precloud or noncloud 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 launched, used, and then released using APIs. In addition, frameworks such as Hadoop have reduced the earlier complexity and expenses involved in building 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.

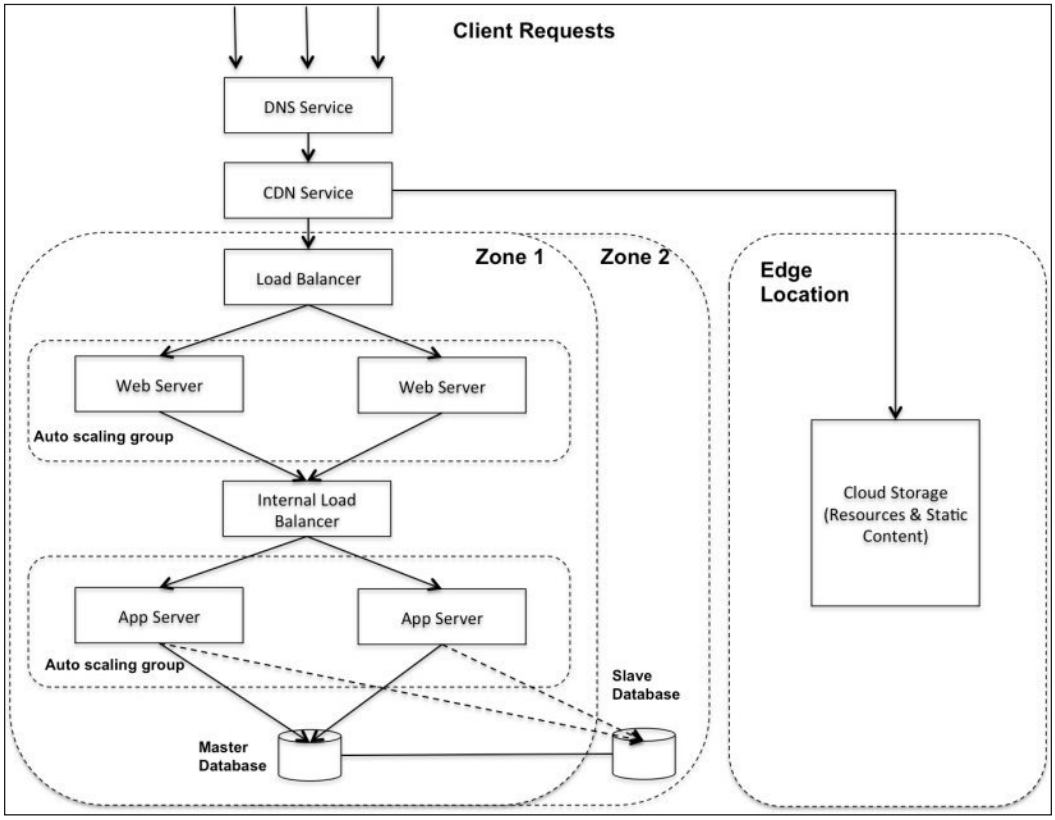
You will need to attack the latency through approaches that can improve the user experience by reducing the perceived and real latency. For example, some of the techniques you can use include using memory optimized instances, right sizing your infrastructure, using caching, and placing your application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

Perceived latency can be reduced by prefetching 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 downloaded to your browser, they don'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 as a **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. On Amazon's CDN service, 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 diagram 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. If 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.

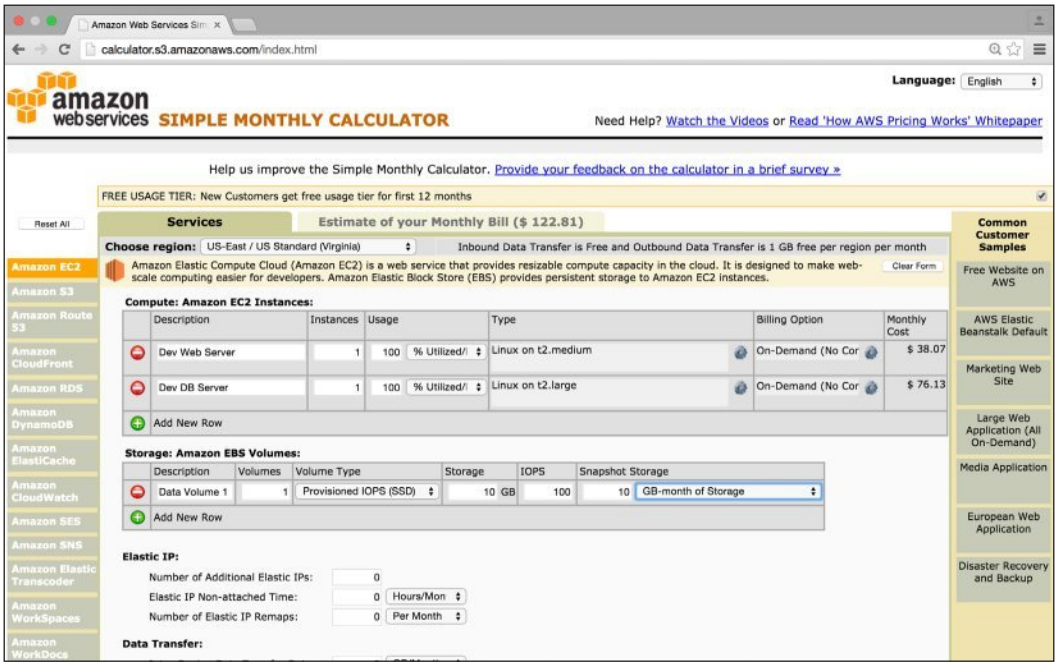
Another approach used to improve scalability and performance is to deploy one or more read replicas close to your end users. This is typically used for read-heavy applications. The read traffic can be routed to these replicas for reduced latencies. These replicas can also support resource-heavy queries for online report generation, serve read-only requests, or run offline queries to support light analytics applications while your main database is down for maintenance or operations activities.

Changes to the source database are applied to the read replicas continuously, but there is a small lag involved. These lags should be monitored to ensure they are within acceptable ranges. Suitable operator alarms should be raised if the delays exceed specified thresholds. Hence, read replicas are considered to be eventually consistent.

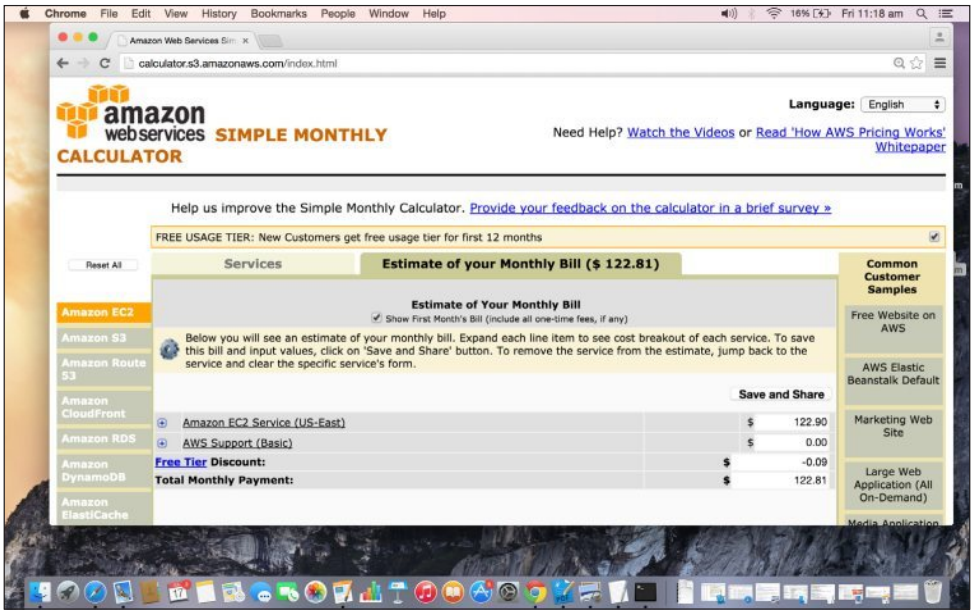
Estimating your cloud computing costs

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 cost 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 get you to ballpark cost estimates quickly.

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 for estimating the costs of 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 lifecycle. 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 might 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 to identify 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 the finance or 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 that we will develop later on. We will also show you how to deploy on the AWS infrastructure. We will leverage off the AWS infrastructure to reduce project timeline and also show you what specific AWS code is needed to support nonfunctional requirements.

This application is not production grade and is only developed to familiarize you with the AWS concepts and infrastructure.

The code base for this application is in Java and the framework used is Spring4.*x* along with MySQL as the database. We will not delve into the design or the specifications as it is not in the scope of the book nor will we develop all the functional use cases defined in the specifications. We will however, dive deep into the nonfunctional specifications as they tend to match what the cloud provides. So, let's start the story.

Suppose `Electronics retailer A1 Sales` has decided to create an e-commerce site to boost their brand and revenues. A1 Sales have identified functional and nonfunctional requirements, which are typical of any e-commerce web application. They have made the decision not to invest in a data center and leverage off the current available cloud infrastructure; hence the e-commerce web application needs to be geared and ready for the cloud from day one.

The nonfunctional requirements will be used by the architect to design the overall solution on AWS cloud. The nonfunctional requirements identified are as follows:

- **Operational Cost**: The architected solution for the e-commerce application should have a minimum monthly operational cost as nothing is free on the cloud. The solution at the minimum should meet the minimum requirements for scalability, availability, fault tolerance, security, replication, and disaster recovery.

- **Scalability cloud infrastructure**: The cloud infrastructure should scale the application up or down by adding/removing application nodes from the network, depending on the load on the application.
- **Scalability application**: The architected solution should be designed in a decoupled and stateless manner, which lends itself to support scaling.
- **High availability**: The architected solution will be designed in a manner which avoids single point failures in order to achieve high availability.
- **Fault tolerant**: The application should be coded to handle cloud services' related failures to the extent possible.
- **Application security**: The application should use an encrypted channel for communications. All the confidential data should be stored in an encrypted format. All the files at rest should be stored in an encrypted format.
- **Cloud infrastructure security**: The cloud infrastructure should be configured to close all the unnecessary network ports with the help of the firewall. All the compute instances on the cloud should be secured with SSH keys.
- **Replication**: All the data should be replicated in real time to a secondary location to reduce the window for data loss.
- **Backups**: All the data from the databases shall be backed up on a daily basis.
- **Disaster recovery**: The architected solution should be designed in a manner that it is easy to recover from an outage with minimal human intervention, with the help of automated scripts.
- **Design for failure**: The architected solution should be designed for failure; in other words, the application should be designed, implemented, and deployed for automated recovery from failure.
- Should be coded using open source software and open standards to prevent vendor lock-in and to drive costs down.

Setting up our 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 that the user has the following packages installed in his development environment:

- **Eclipse or Spring Tool Suite (STS)**: For downloading STS/Eclipse the links are `http://spring.io/tools/sts/all` and `https://eclipse.org/downloads/`.
- **JDK 1.7**: The download link is `http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html`.
- **Maven 3**: For downloading Maven 3, the link is `http://maven.apache.org/download.cgi`.
- **Git command line tools**: For download the link is `http://git-scm.com/downloads`.
- **Eclipse with Maven plugin (m2e)**: m2e is installed by default when using the STS. You can install the previous M2Eclipse release by using the following update site from within Eclipse, by navigating to **Help | Install New Software** on `http://download.eclipse.org/technology/m2e/releases`.

The following instructions mentioned are for Linux but not limited to, they can be also used for Windows and Mac OS X with minor or no modifications.

Let's get started:

1. To begin with, create a folder named `a1electronics` in your preferred workspace `mkdir a1electronics`.
2. Next, download the source code from the GitHub repository.
3. Switch to the created folder `a1electronics`, and clone the source code from the `git` repository, using the following command:

   ```
   git clone https://github.com/a1electronics/ecommerce a1ecommerce
   ```

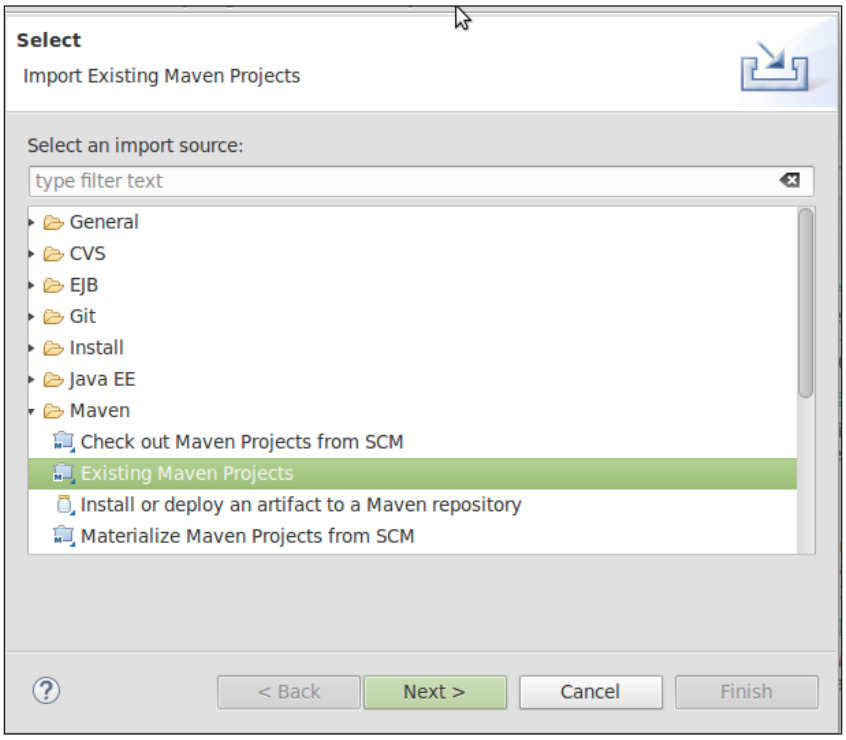
4. Now that you have the source code, the next step is to import the project into Eclipse or STS. Alternatively, you can run it directly from the command line.

5. To run the application directly from the command line, irrespective of importing the project into STS, using the following command:

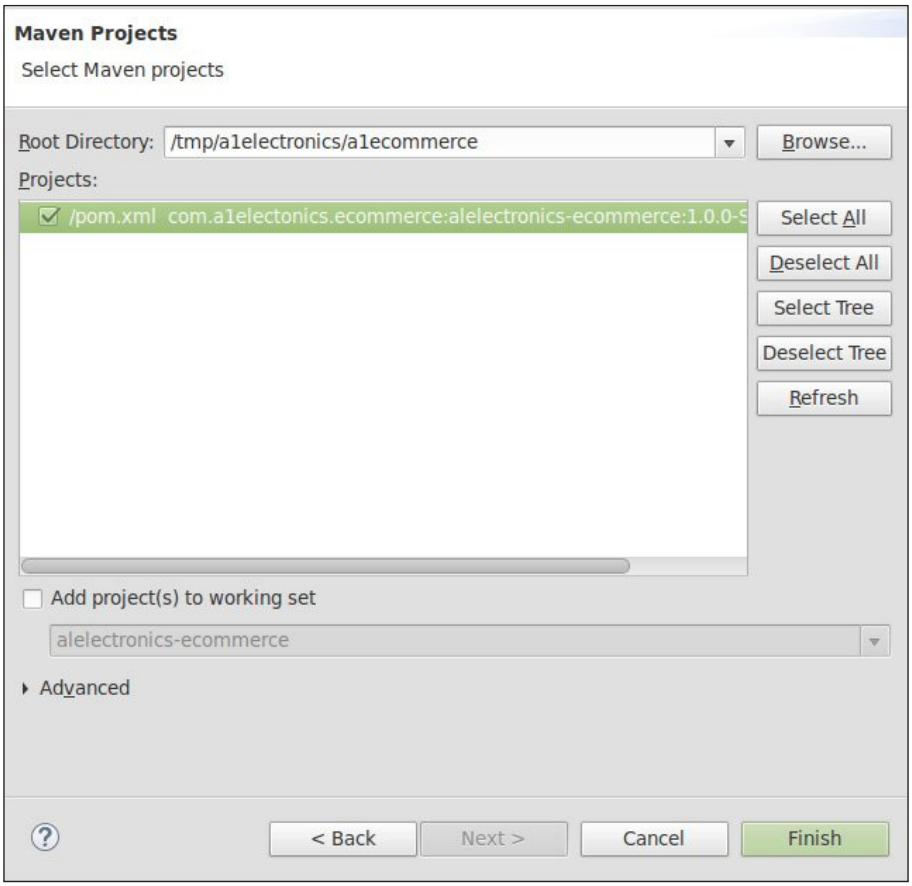
```
mvn tomcat7:run
```

   This will launch the web server tomcat along with the application on port 8888. You can type in the following URL in your favorite browser `http://localhost:8888/a1ecommerce` and viola you have the `A1 Electronics` home page.

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



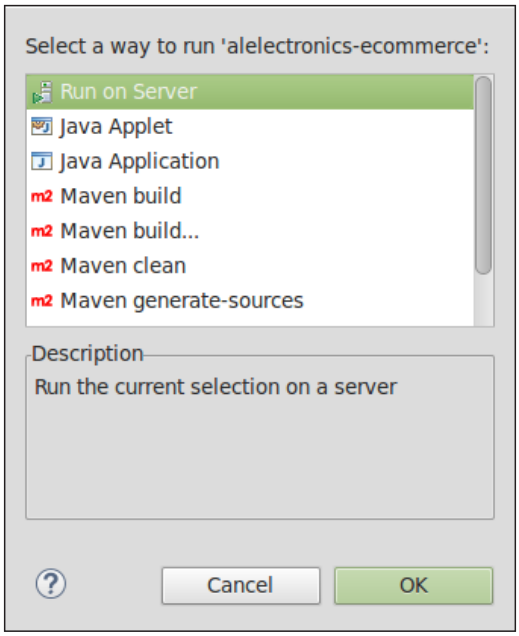
7. In the next step, as shown in the following screenshot, you will need to point to the folder where you checked out the application from the `git` repository. As described in the earlier section, clicking on **Finish** will successfully import your project into STS:



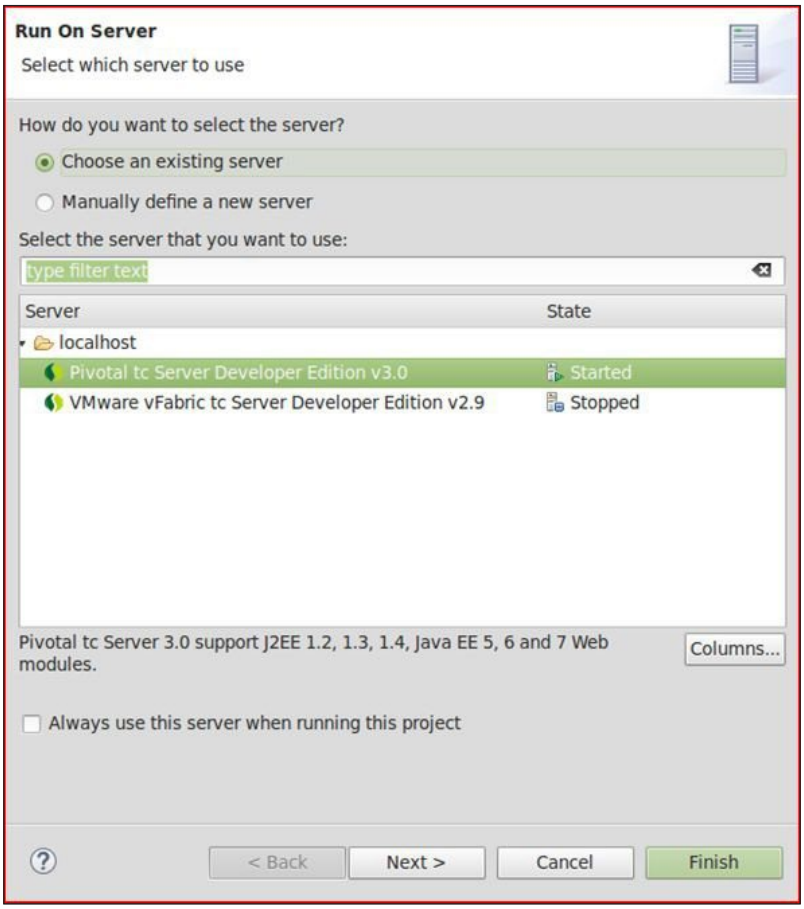
Running the application

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

1. You can launch the imported project from the **Package Explorer** by selecting the menu option **Run** | **Run**; this will open a pop-up window, as shown here:



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 screenshot. Clicking on **Finish** will launch the `a1electonics` e-commerce application from within the STS:



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 the servlet container. This can be done via the command line from the root of the `a1ecommerce` 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 `a1ecommerce` project.

Summary

In this chapter, we explained the differences in design and implementation of cloud-based application. We reviewed some of the architectural best practices in the cloud context. More specifically, we described multi-tiered, 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. We also created a simple application that we intend to expand and elaborate in the coming chapters to illustrate the 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 the AWS-specific cloud services, the costing model, and application development environments.

3

AWS Components, Cost Model, and Application Development Environments

This chapter will describe the main AWS components and services. We will also cover strategies to lower your cloud infrastructure costs, and how they influence your architectural decisions. Furthermore, this chapter will discuss the typical characteristics and functions 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.

AWS components

AWS offers a variety of infrastructural services. The list of AWS services is a continuously growing list of services with several of them in preview mode at any given time. In this section, we will describe some of the main AWS services.

Amazon Elastic Compute Cloud (EC2)

Amazon EC2 is a web service that provides compute capacity in the AWS cloud. You can bundle the operating system, application software, and associated configuration settings into an **Amazon Machine Image** (**AMI**). You can then use these AMIs to provision multiple virtualized instances as well as decommission them using web service calls. EC2 instances can be resized and the number of instances scaled up or down to match your requirements or demand. These instances can be launched in one or more geographical locations or regions, and Availability Zones (AZs). Each region comprises of several AZs at distinct locations, connected by low latency networks in the same region.

Amazon Elastic Block Storage (**Amazon 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. You can create auto scaling groups using the auto scaling feature to automatically scale your capacity based on CloudWatch metrics. You can also distribute incoming traffic by using the **Elastic Load Balancer** (**ELB**) service. You can also use the AWS CloudTrail service to monitor AWS API and AWS SDK calls for your account.

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.

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.

Amazon EBS

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

Amazon CloudFront

The Amazon CloudFront service is a 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 the Amazon CloudFront service.

Amazon Glacier

Amazon Glacier is low-cost storage service that is typically used for archiving and backups. The retrieval time for data on Glacier is up to several hours.

Other AWS Storage services include Amazon Storage Gateway (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 the AWS cloud environment).

Amazon RDS

Amazon Relational Database Service (**Amazon RDS**) provides an easy way to setup, operate, and scale a relational database in the cloud. Database options available from AWS include MySQL, Oracle, SQL Server, PostgreSQL, and Amazon Aurora (in preview at this time). You can launch a DB instance and get access to a full-featured MySQL database, while reducing effort on common database administration tasks like backups, patch management, and so on.

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.

Amazon ElastiCache

If your application is read-intensive, then you can use the 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, automates patch management, and enables monitoring through integration with Amazon CloudWatch. ElastiCache can be scaled-up/scaled-down in response to the application load.

Amazon Simple Queue Service

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

Amazon Simple Notification Service

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

Amazon Virtual Private Cloud

Amazon Virtual Private Cloud (**Amazon VPC**) allows you to extend your corporate network into a private cloud contained within AWS. Amazon VPC uses the IPSec tunnel mode that enables you to create a secure connection between a gateway in your data center and a gateway in AWS.

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.

AWS Identity and Access Management

AWS **Identity and Access Management** (**IAM**) enables you to 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.

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 arising in your cloud environment.

Other AWS services

There are several other AWS services that assist you in the administration of your cloud environment. These include CloudTrail (records AWS API calls), AWS Config (provides you with a resource inventory and current configuration of your AWS resources), AWS CloudHSM (helps you meet contractual obligations and/or compliance requirements), and AWS Key Management (to manage your data encryption keys).

In addition, AWS provides several services for deployment and management of your applications. These include AWS Elastic Beanstalk (for deploying and scaling web applications), AWS OpsWorks (an application management service), AWS CloudFormation (for provisioning a set of related AWS resources), and AWS CodeDeploy (for automating code deployments).

Other applications' related services from AWS include Amazon EMR (a hosted Hadoop framework), Amazon Kinesis (for real time streaming data ingestion and processing), Amazon SWF (a workflow service), Amazon AppStream (for streaming from the cloud), Amazon Elastic Transcoder (for conversion of media files), Amazon SES (a bulk e-mail service), and Amazon CloudSearch (for applications that need a scalable search service functionality). AWS also offers payment and billing services that leverage Amazon's payment infrastructure.

Other than 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.

Optimizing cloud infrastructure costs

It is important to understand your cloud-costing model, so that you are able to manage costs better. Typically, a substantial part of your bill comprises of 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 many more). However, storage costs can also be significant for certain applications such as a photo-sharing application. In this section, we will focus on several strategies that will help you cut your cloud infrastructure costs.

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.

Most often pursuing these cost-cutting measures can lead to a leaner and a more robust architecture. In addition, these measures are a lot easier to implement on AWS cloud than in the traditional data center-based infrastructure.

With an ever-increasing number of services and customers, Amazon has been able to leverage economies of scale and pass on additional savings to their customers. In fact, they have reduced prices over forty times since 2006, and prices continue to be revised downwards for various services on an on-going basis.

The infrastructure setup process in the precloud 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 through architect-build-monitor steps. Cloud infrastructure is dynamically allocated and deallocated so we do not have to be 100 percent right in all our infrastructure design decisions, the first time around. We can iteratively improve our architecture while meeting our infrastructure cost objectives.

The free version of AWS Trusted Advisor is available to all customers through the AWS Management Console. This version includes a limited number of performance and security recommendations. Additional features are available for startups and for customers who sign-up for Business-level and Enterprise-level support. AWS Trusted Advisor helps you provision your resources by following best practices, inspects your environment, and guides you in optimizing your costs. These savings can help defray the AWS support costs.

There are several strategies that can result in substantial savings, and most of these are relatively easy to implement. We discuss the main ones to provide you guidance on lowering your cloud infrastructure spending in the next few sections.

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 I/O 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 that are 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 GPU 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 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 you shortlist two instance type—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 type for comparison purposes. In order to conduct the performance analysis, create a set of test cases to test a few scenarios in your application. Monitor the CPU utilization for these instances at different loads, say 1000, 2000, and 3000 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.

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 prewarmed. 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, by 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.

Enhanced networking option is available on C3, C4, D2, I2, and R3 instances, but not on general purpose instances.

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.

Finally, 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.

Turn-off unused instances

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 is over. It is important to check your bills and usage data to minimize such costs. Tag your instances with the environment name, owner's name, and so on to identify the instances, and the primary owner or cost center, quickly.

Instances can be switched on and off easily, so ensure you switch off your dev, test, and training instances after office hours and through the weekends. You can also automate your infrastructure for this purpose using AWS CloudFormation. On cloud, instances are disposable; hence, you do not need to keep them on when they are not being used. You can easily save 30-40 percent on your bills this way.

Use auto scaling

Automatically scale your compute instances to the extent required; otherwise, scale down automatically. You can define launch configurations for your EC2 instances and then set up appropriate auto scaling groups for them. You can select parameters such as the minimum and maximum number of instances. This helps automate the process of saving money, by turning off unused instances during scale down.

During a scale-up it can take a few minutes for your new instances to come online. So ensure you account for this while establishing your thresholds. Do not set the threshold too high (for example, at 90 percent CPU utilization) because there is a high chance your existing instances will hit 100 percent utilization before your new instances have spun up. It is a good practice to set the CPU utilization threshold to be between 60-70 percent, as it gives you sufficient headroom. 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 (if there is an immediate dip 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 to arrive at the right threshold values. However, avoid setting multiple scaling triggers per auto scaling group because, if this increases the chance of conflict in your triggers, then 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 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 events. Special events such as a big sales event by a popular e-commerce site, or a news site during the Olympics, or the elections might lead to disruptions due to huge demand on resources during these events. In such cases, it might be a better approach to overprovision 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 also 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.

Now that 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 costs and usage at this stage, then monitor them closely 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.

Use reserved instances

Reserved instances can help save 50-60 percent or higher on your instance costs (versus using on-demand instances). You have flexibility to pay all, part, or nothing upfront, and they are available for a duration of 1 year or 3 years at much lower hourly rates. You can also modify or sell your reserved instances if your requirements change. Typical breakeven on these instances vary between 5 months to 10 months depending on the duration of the contract. Reserved instances are flexible, for example, they can be moved between AZs and their sizes can be modified (within the same instance family).

As production instances are typically required to run 24/7/365 in a reliable manner, reserved instances are a good fit for enterprise applications (in production). For dev/test environments (and in start-ups), 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.

Use spot instances

Spot instances can be the most cost effective option for experimentation and learning purposes, and also for establishing economical cloud environments. 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.

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. If there is available capacity, then Amazon will fulfill that request. However, your instance is terminated if the spot price becomes 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 availability zones. 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 availability zone, in case the price in your current availability zone goes higher than your set price.

Spot instances give you an opportunity to name your own price and can potentially save you 80-90 percent of your instance-related costs. However, understand the risks associated with using them. You can leverage auto scaling to reduce your overall risks, for example, you can define one group with spot instances and a second with on-demand instances.

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

Use Amazon S3 storage classes

Using the **Reduced Redundancy Storage** (**RRS**) option in Amazon S3 storage can reduce your costs by storing non-critical 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 percent data durability versus 99.999999999 percent using the standard option. This can lead to savings of 15 to 20 percent on storage.

Aside from enabling RRS, Amazon Glacier storage class can be used for storing backups and archiving old data. Amazon Glacier is low cost storage with 99.999999999 percent data durability. Data restores from Glacier storage can take 3 to 5 hours. However, this can result in 50-60 percent savings on storage. You can also specify life-cycle rules to automate data movement from S3 to the Glacier storage.

Reducing database costs

Caching and Read Replicas can reduce the capacity required for your database instance in read intensive transactions/applications. 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 might be lower than additional capacity allocation for your database instance for your application type).

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.

Using AWS services

AWS makes available a bunch of ready-to-use services that you can integrate into your application. This 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, you can use Amazon CloudFront in front of your web architecture. CloudFront caches your static and dynamic content thereby helping you scale down the architecture behind CloudFront.

Cost monitoring and analysis

The AWS platform provides a set of tools to help monitor and analyze your costs. These include the AWS TCO calculator, a simple monthly calculator (described in *Chapter 2*, *Designing Cloud Applications – An Architect's Perspective*), the AWS billing console that shows you an itemized bill, and AWS Cost Explorer that gives you costs' trends information across different time periods. You can also set AWS Billing Alerts, that will 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 in your 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, comparison across time periods, and underutilized instances.

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

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 required. This section discusses these environments and their features.

Development environments

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 will 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 shutdown at the end of each day and through the weekends. However, during crunch periods these environments are kept running for extended hours.

QA/Test environment

This environment is typically provisioned for supporting functional and nonfunctional testing activities. They use small instances and 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 shutdown on a daily basis and during 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 will be expensive to keep it running continuously; hence, it should be brought only when necessary to support the aforementioned activities. Application deployments occur only when required, for instance, to test the application before a production migration.

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.

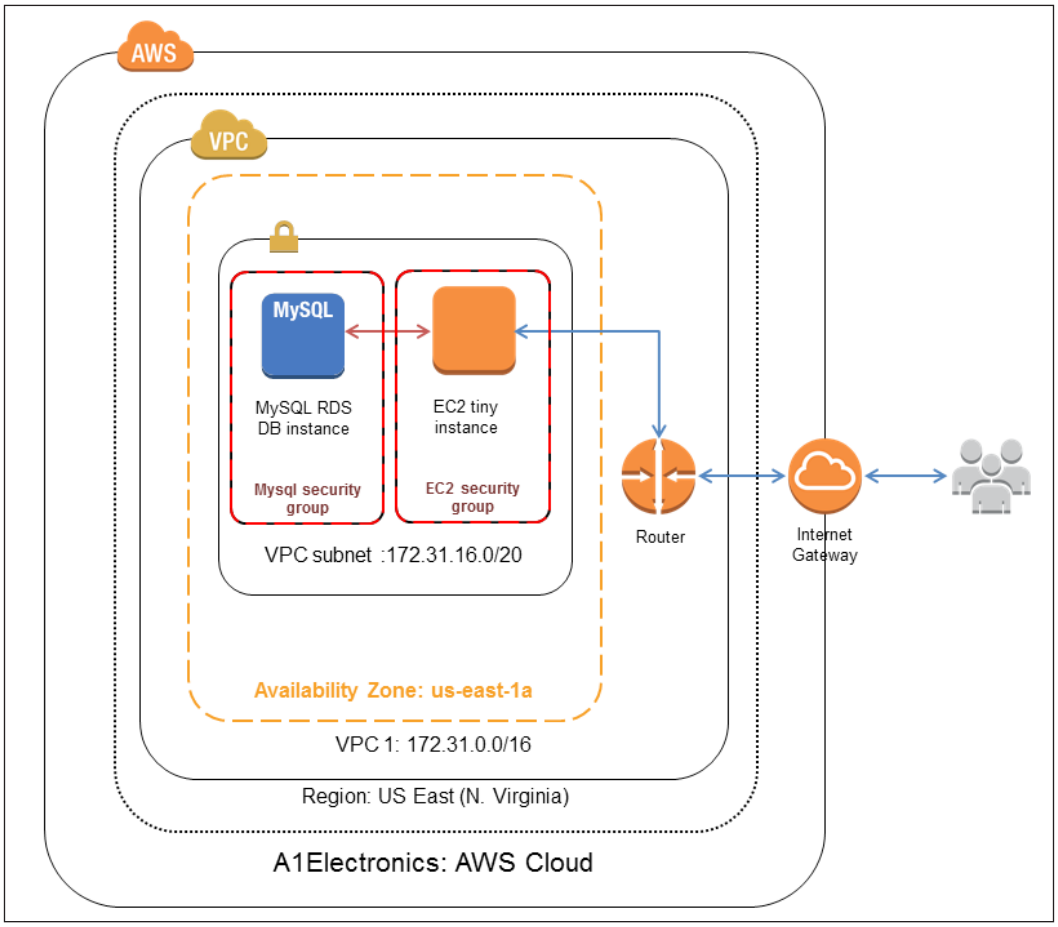
Setting up the AWS infrastructure

This section introduces you to provisioning the AWS infrastructure in order to deploy and run the `A1Electronics` e-commerce application securely on AWS. You will also see the code changes required at the application level. 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. We will automate the entire process, which will allow you to launch and stop existing RDS and EC2 instances, and also create new ones. So let's begin.

The AWS cloud deployment architecture

Before we start, we need to have the deployment architecture in place. The term deployment architecture here describes the manner in which a set of resources like the web server, the application server, databases, DNS servers, load balancers, or any other specific AWS resources are configured on the network to fulfil the system requirements (and ultimately satisfy your business goals). The following diagram shows a simple AWS deployment architecture for our A1Electronics e-commerce application. Production grade AWS deployment architecture will be discussed later, in *Chapter 7*, *Deploying to Production and Going Live*.

The following figure is created with Amazon provided AWS icons that can be downloaded from `http://aws.amazon.com/architecture/icons/`:



Let's get familiar with the AWS terms in the preceding diagram.

- **Region**: AWS services are hosted in multiple locations around the word and these are known as regions. The regions are connected through the public internet. The main criteria for choosing a specific AWS region are:
    - **Location 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, the **US East (N. Virginia)** region is selected.

    - **Price**: The products and services offered by Amazon are priced differently across the regions. For example, we can choose a region with the cheapest pricing 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/`.
    - **Not all 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`.

- **Availability Zone**: Availability zones (AZ) can be treated as traditional data centers within a region. AZs in the same region are designed to provide infrastructure redundancy in the event of a catastrophic outage, such as earthquakes, snowstorms, Godzilla attacks, and so on. The number of AZs in a region is region specific. In our example, we select the **us-east-1a** AZ.
- **EC2 Instance**: This 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 high network I/O memory associated with it. You cannot have a low compute EC2 instance with high memory and network I/O. EC2 instances have fixed CPU to memory ratios. It is best to select a micro instance for 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)**: Amazon 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, and 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 source and destination IP address ranges to accept or send your data to.
- **Virtual Private Cloud (VPC)**: 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 centre. 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 that allows us define a total of 65536 subnets or a total of 65534 addressable resources.

AWS resources launched within a VPC aren't addressable via the global internet, EC2, or by resources in any other VPC. Resources can be accessed only by resources running within the same VPC.

- **Subnets**: 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 the VPC. For example, connecting to a database server in other VPCs.
- **Internet gateway**: Each VPC also comes with a default Internet gateway to connect to the public Internet; this Internet gateway is on the VPC side of a connection to the public Internet.

Let's now begin the construction.

AWS cloud construction

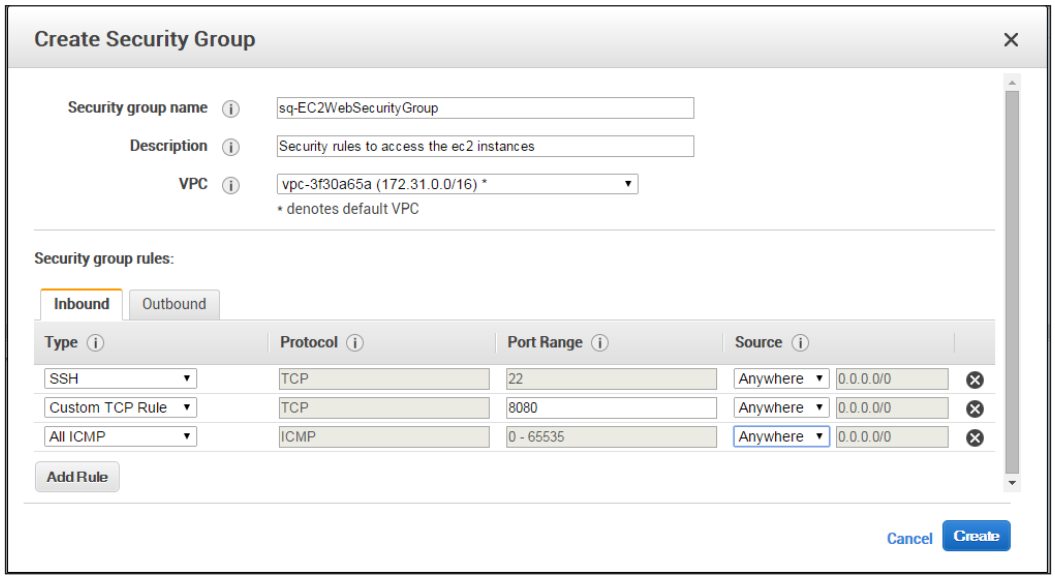
To create a working AWS cloud infrastructure, you will need to create security groups, key pairs, users and roles, MySQL RDS instances, EC2 instances, Elastic IPs, and wire them all together. We will create this in a bottom up manner where we create the base AWS constructs such as security groups, key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

Creating security groups

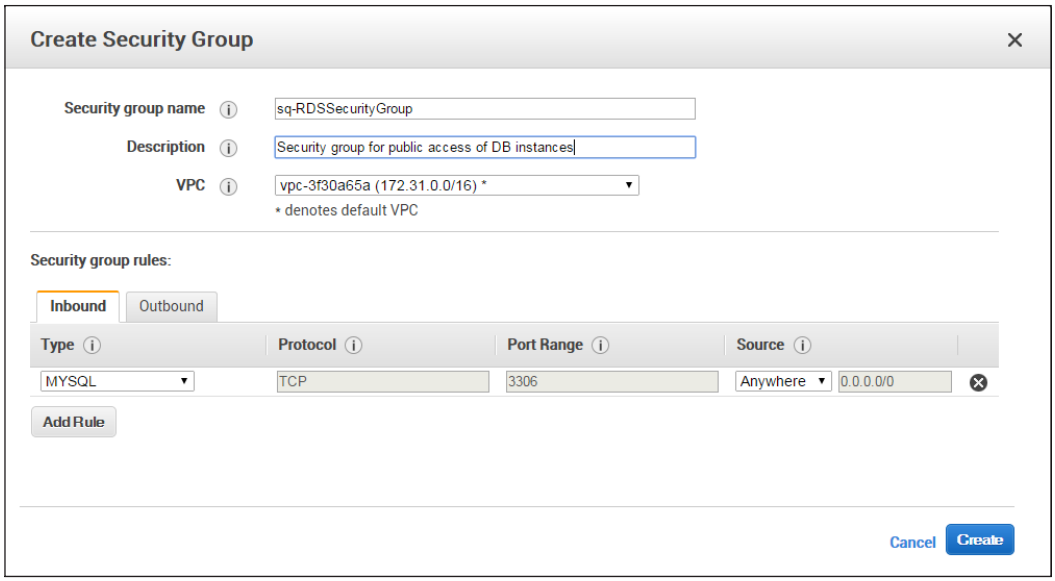
As per the deployment architecture diagram, we need to create two security groups; one for the EC2 instance and the other for the RDS MySQL instance. To create the group, perform the following steps:

1. From the EC2 dashboard, click on **Security Groups** from the navigation pane and then 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 a public Internet



3. Create a security group for MySQL RDS instances to allow access from the Internet.
    - During the development phase, we need to have direct access to databases from our development environment. This makes it is easy to change, 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 production environments, it is recommended to allow database access only from within the VPC. Select **Anywhere** from **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 a dynamic IP address, then you will need to update this rule to the most recent, as shown in the following screenshot:



Creating EC2 instance key pairs

AWS uses public/private keys to securely connect to your instances. The public key will be retained by AWS, while the private key is downloaded to your computer as soon as it is created. To create a key pair, perform the following steps:

1. From the EC2 dashboard, click on **Key Pairs** from the navigation pane and then on the **Create Key Pair** button.
2. Enter `ec2AccessKey` when prompted with a dialog box asking to enter the key pair name. This key pair name will be used while configuring the EC2 instances and in CloudFormation scripts.

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

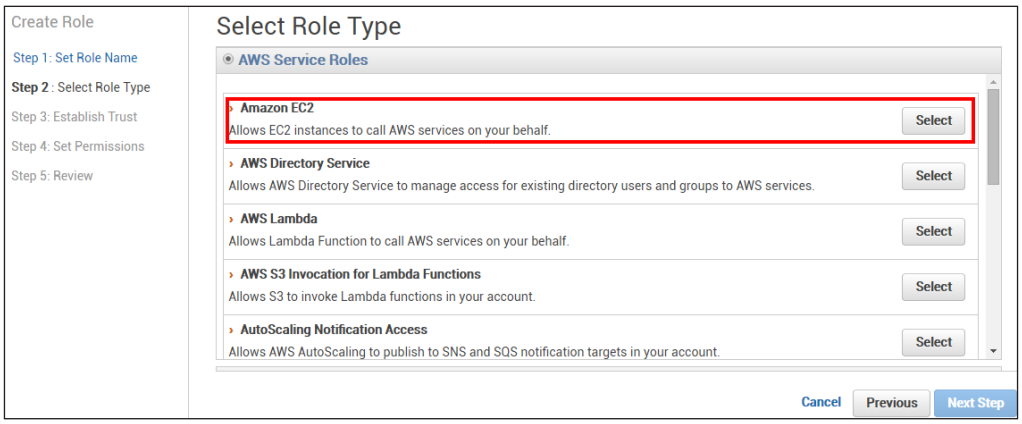
As soon as you create the key pair, your private key will be immediately downloaded to your computer. Secure this private key. This private key file can be only downloaded once during the creation of the keys. You cannot change access keys in your EC2 instances once they have been assigned.

Creating Roles

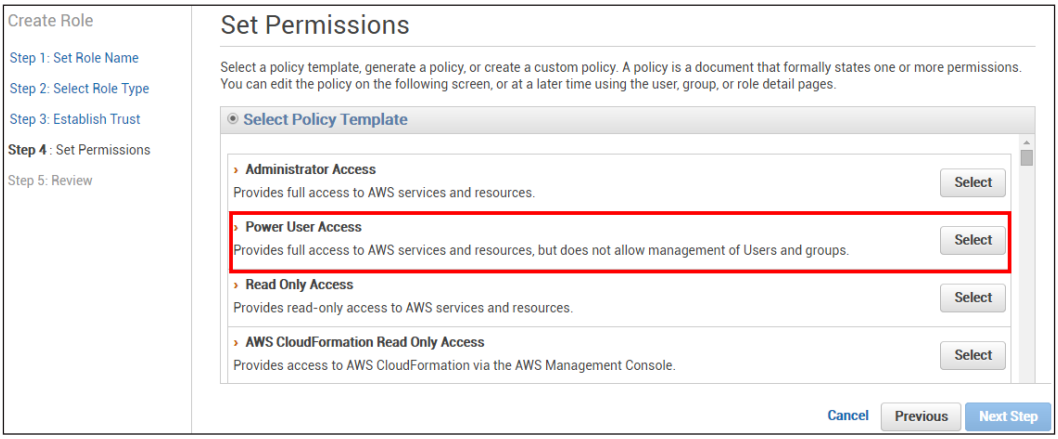
Role is a set of permissions that grant access to AWS services. Roles are independent of users or groups. You will need a strategy to distribute and rotate the credentials to your EC2 instances; especially, the ones which AWS creates on your behalf, for example, Spot instances or Auto Scaling groups. A good security practice is credential scoping—granting access only to the services your application requires. AWS solves the issues of credential scoping and credential distribution via IAM roles.

1. From the IAM dashboard, click on **Roles** in the navigation pane and then on the **Create New Role** button.
2. Create a role named `ec2Instances` for our EC2 instances that have access to all the AWS provided services, as shown in the following screenshot:

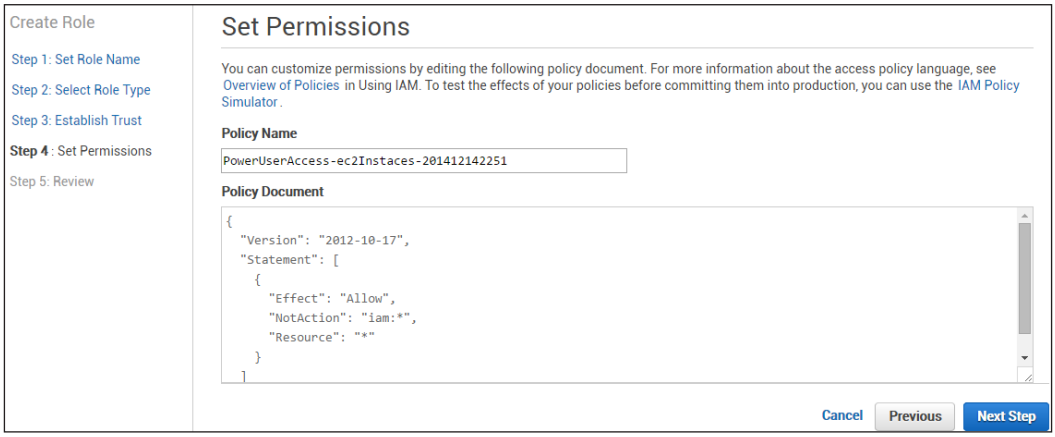
3. The next step is to grant permissions to the selected AWS services. Click on the **Amazon EC2** role type, as shown in the following screenshot:



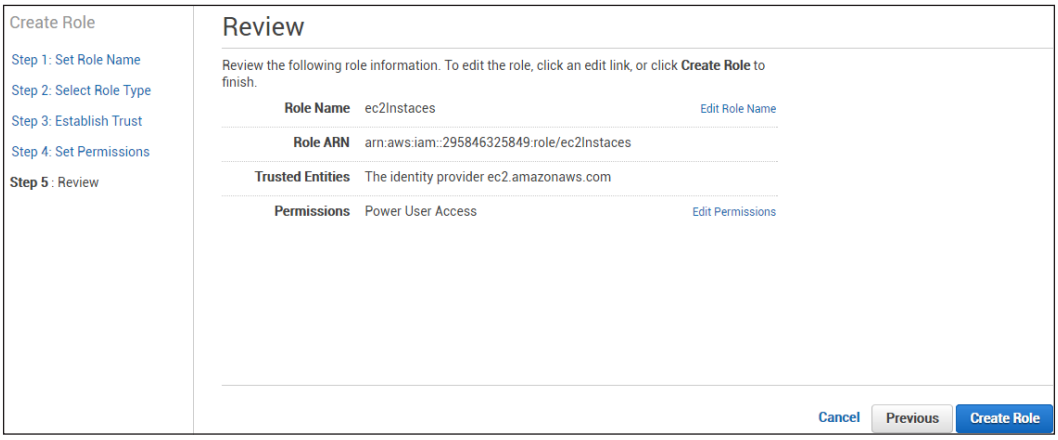
4. Next, we will assign permissions for the selected role. Select the **Power User Access** option. For now, we do not have any credential scoping. Read and write permissions for all AWS services are granted to the selected role. Permissions to the role can be reassigned even when the EC2 instance is running. Let's have a look at the following screenshot:



5. The following screen allows you the option of copying the permissions and testing them in the IAM simulator. It also allows you to modify the permissions as per your requirements. No changes are required to be made here for now. Click on the **Next Step** button, as shown in the following screenshot:



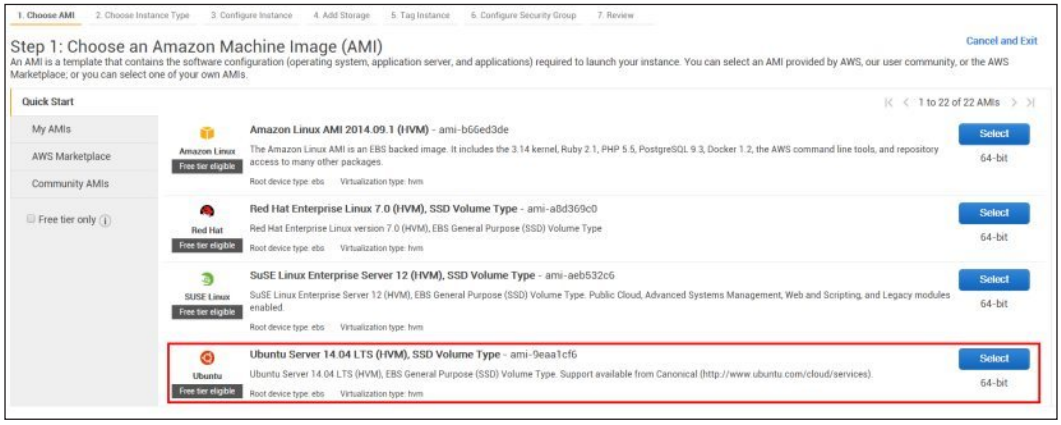
6. The last screen allows you to preview your selected options before creating a role. Click on the **Create Role** button, as shown in the following screenshot:



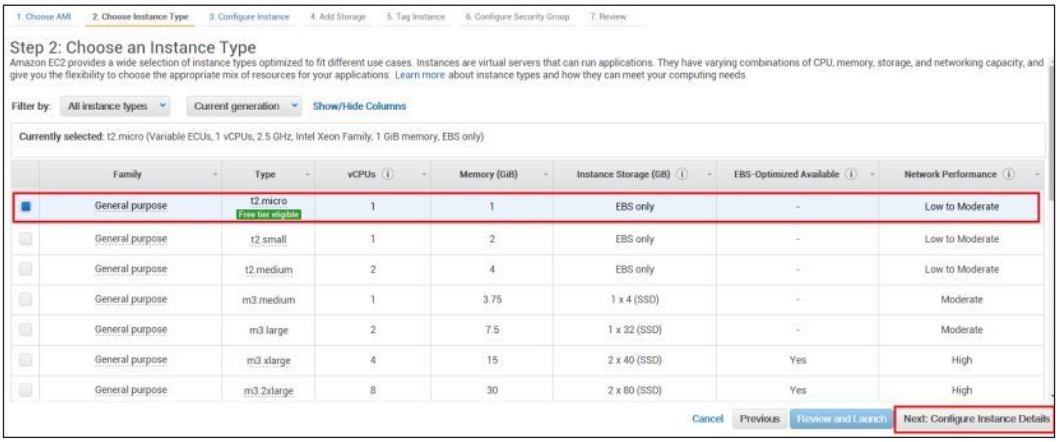
Creating an EC2 Instance

Since we have already done the groundwork (steps 1 to 3), it is just a matter of wiring the EC2 instance. Perform the following steps:

1. From the EC2 dashboard, click on **Instances** in the navigation pane and on the **Launch** instance. This will start a process of provisioning an EC2 instance.
2. The next step is to choose an operating system for the EC2 instance; this is done by choosing the correct **Amazon Machine Image** (**AMI**) as per our requirements. Select the **Ubuntu Server 14.04 LTS (HVM) SSD Volume Type** AMI, as shown in the following screenshot:



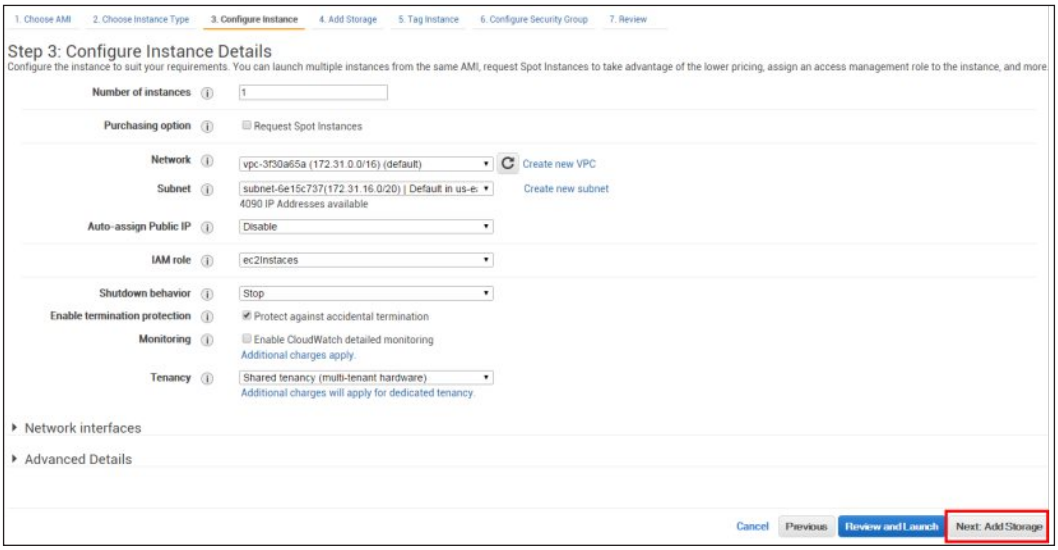
3. After selecting an AMI image, the next option is to choose an instance type. The instance is the virtual server that will run our application. Select the **t2.micro** instance that is included in the free-tier for a period of 1 year from the date you have created your AWS account. Click on the **Next:Configure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



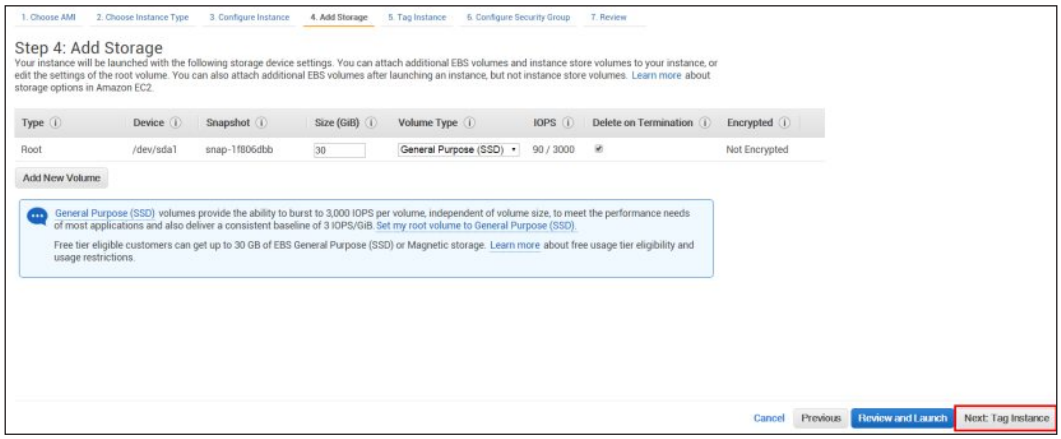
4. Next, we configure the instance. Here, we have several options and we need to make the most appropriate choices:
    - **Number of instances**: This allows launching of multiple AMI instances. By default, it is set to `1` (no need to change that). You can always launch multiple instances via the EC2 dashboard.
    - **Purchasing Option**: Since we are using the free tier, we can ignore this. The idea of purchasing this option relates to excess capacity for an instance type in an AWS region made available for use at a lower price than the advertised price.
    - **Network**: By default, all EC2 instances are launched in a VPC. We use the default VPC.
    - **Subnet**: By default, each subnet is associated with an availability zone within a region. Select the **172.31.16.0/20** subnet associated with us-east-1a AZ.

- **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 addresses. The public IP is not persistent. If we want the public IP address to be persistent across restarts, then we have to use an Elastic IP that we will set up in step 5. Set this to **Disable** for now.
- **IAM role**: Select the role **ec2Instances** created earlier in step 3.
- **Shutdown behavior**: An instance can be either stopped or terminated on shutdown. Select **Stop**.
- **Enable termination protection**: This is a means to disable the terminate option for the EC2 instance in the EC2 dashboard. Select this option.
- **Enable Monitoring**: This is for enabling collection of metrics and analysis via AWS CloudWatch. Logging of basic metrics is free (with restrictions). Refer to `https://aws.amazon.com/cloudwatch/pricing/` to know what's free and what you have to pay for. For our purposes, you do not need to select this option.
- **Tenancy**: Shared tenancy uses an over-subscription model to rent the hardware among the customers; this makes the performance of the EC2 instance unpredictable. 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.
- **Network Interface**: This option is used to add extra network interfaces to the EC2 instance. No changes are needed.
- **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. No changes are needed.

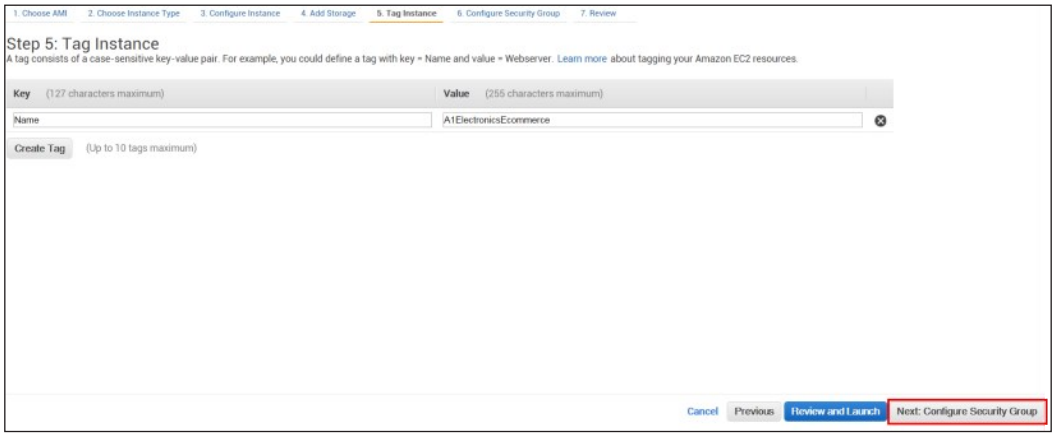
5. Click on **Next: Add Storage** to provision persistent storage, as shown in the following screenshot:



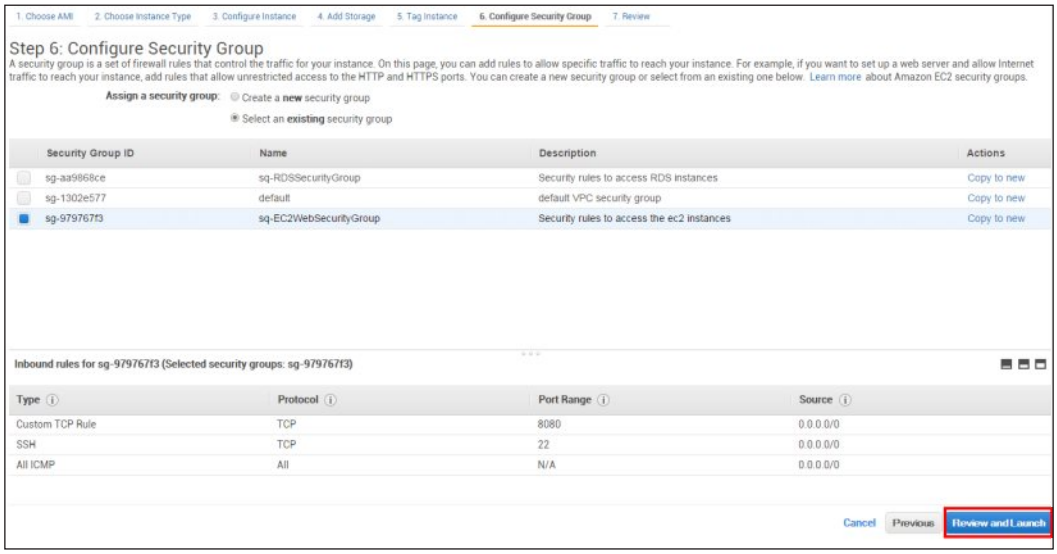
6. Next, we configure the persistence storage also known as elastic block storage (EBS). It is the hard disk for your EC2 instance. Up to 30 GB of disk is available in the free tier, which is sufficient for most applications. Select **General Purpose (SSD)** from the **Volume Type** column. The data 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**, as shown in the following screenshot:



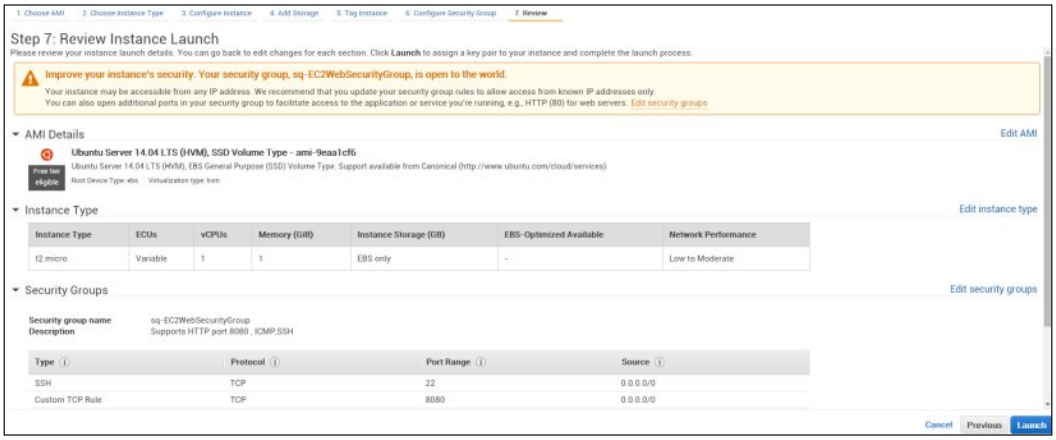
7. Next, we tag the EC2 instance. Tags do not have any semantic value and are treated purely as strings in a key-value form. You can work using the tags with the AWS management console, EC2 API, and EC2 command line interface tools. Click on **Next: Configure Security Group**. Let's have a look at the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



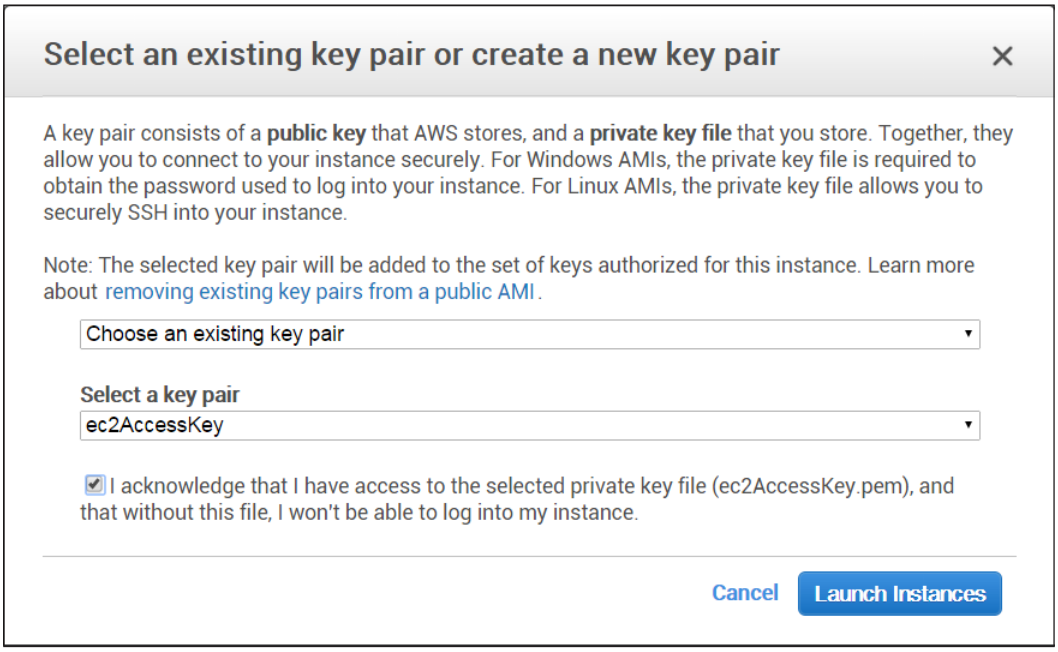
8. Next, we assign the security group **sq-EC2WebSecurityGroup** we defined earlier in step 1. Click on the **Select an existing security group** radio button to view all the available predefined security groups. Select **sq- EC2WebSecurityGroup** from the list. Click on **Review and Launch**, as shown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



9. Next, we can review the options we have selected, and modify them, if required. Click on **Launch** to launch the instance. Let's have a look at the following screenshot:

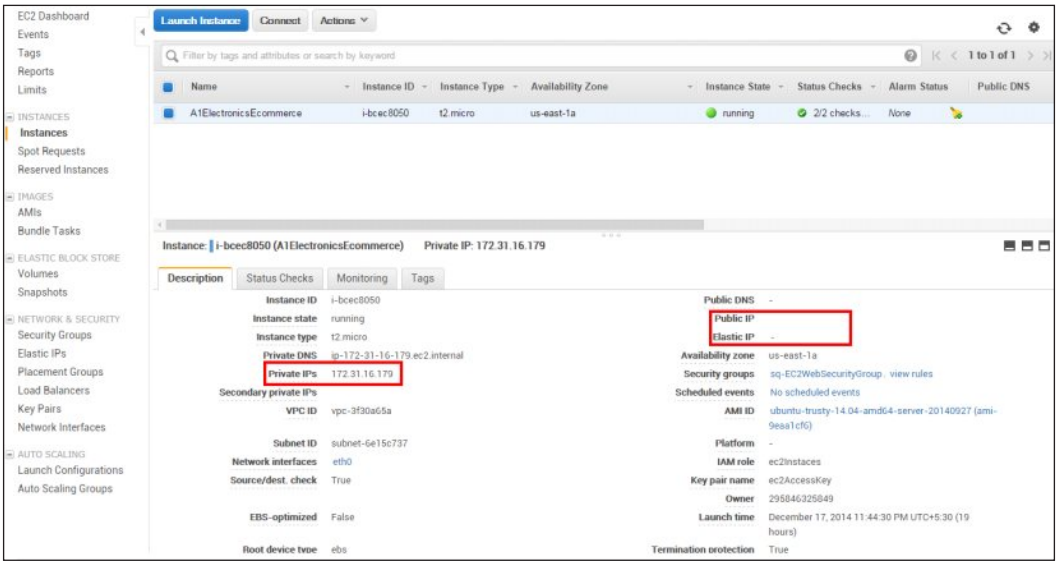


10. Upon launch, the EC2 instance will prompt you to select the public/private key pair, which was created in Step 2. Select the **ec2AccessKey** from the drop down list box. Click on **Launch Instances** to launch the EC2 instance. Let's have a look at the following screenshot:



The key pair once assigned to an instance cannot be changed. Make sure you store your private key securely.

11. Your EC2 instance will take some time to start. You cannot access it 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 **172.31.16.179**. This EC2 instance can be used only for communication between the instances in your VPC. Next, create an elastic IP and assign it to the EC2 instance so that it can be accessed via the public internet. Let's have a look at the following screenshot:



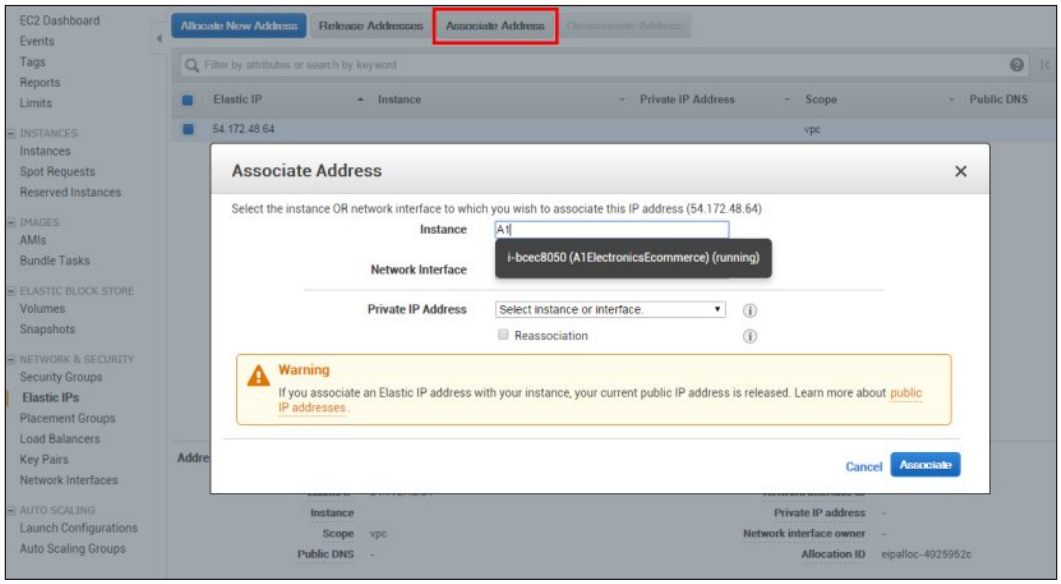
Elastic IPs (EIP)

EIPs are dynamically remappable static public IP addresses that make it easier to manage EC2 instances. Each EIP can be reassigned to a different EC2 instance when needed. You control the EIP address until you choose to explicitly release it. An EIP 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; hence consuming a single EIP.

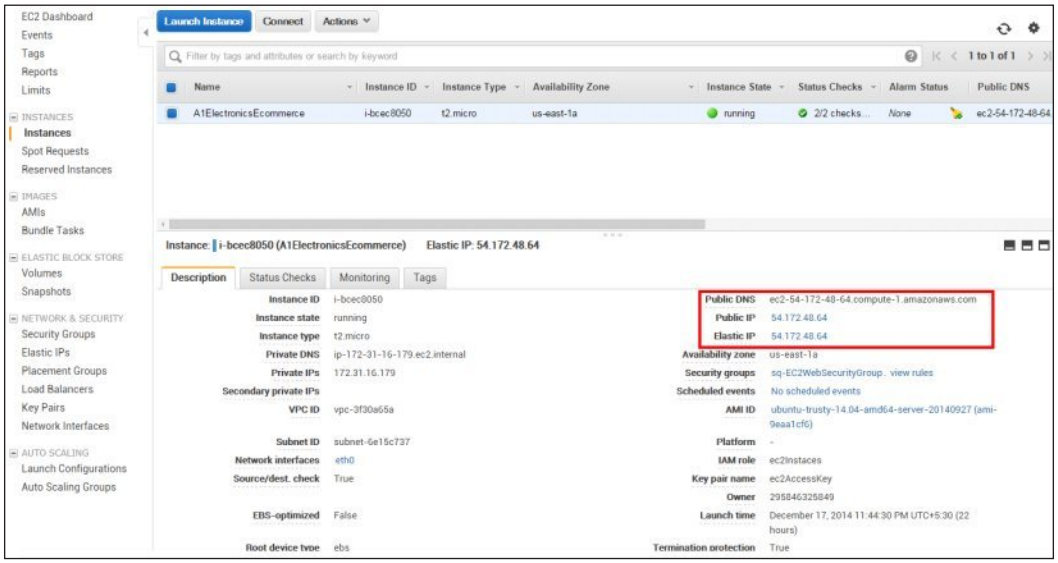
You will be charged for all EIPs not associated with running EC2 instances. It is charged at 0.01$/hour for each EIP that is not associated.

To create an EIP, perform the following steps:

1. From the EC2 dashboard, click on **Elastic IPs** in the navigation pane and then on **Allocate New Address**. This will assign a new EIP to your account.
2. The next step is to associate the EIP to an instance. Click on **Associate Address**. Type the tag name of the EC2 instance you want to associate this EIP with on **Instance**. Select the instance, and click on **Associate**. Let's have a look at the following screenshot:



3. From the EC2 dashboard, click on **Instances** in the navigation pane and 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 **54.172.48.64**, or by the domain name **ec2-54-172-48-64.compute-1.amazonaws.com** from the terminal. Let's have a look at the following screenshot:

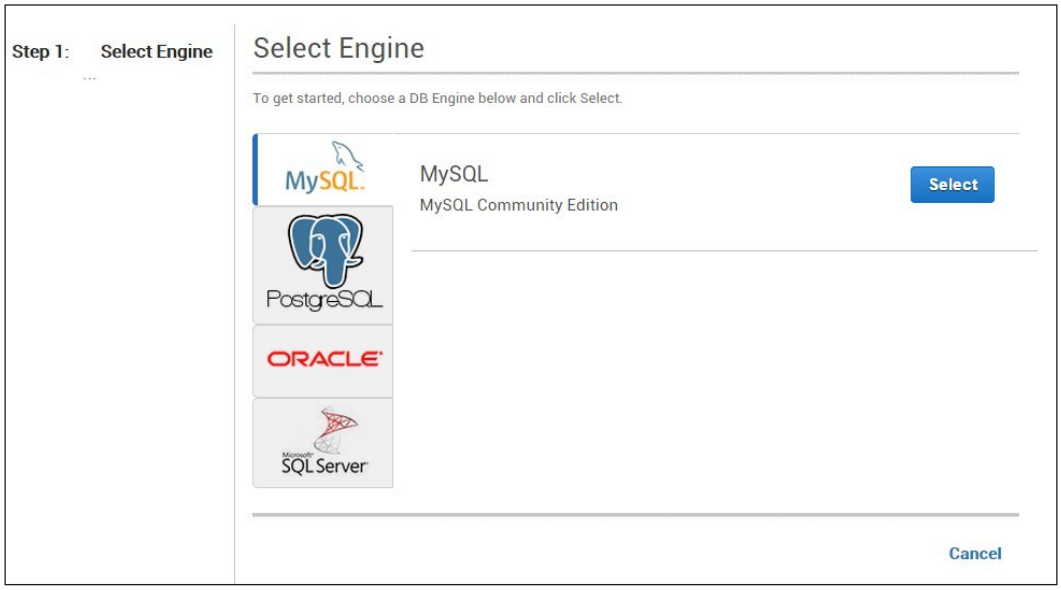


Amazon Relational Database Service

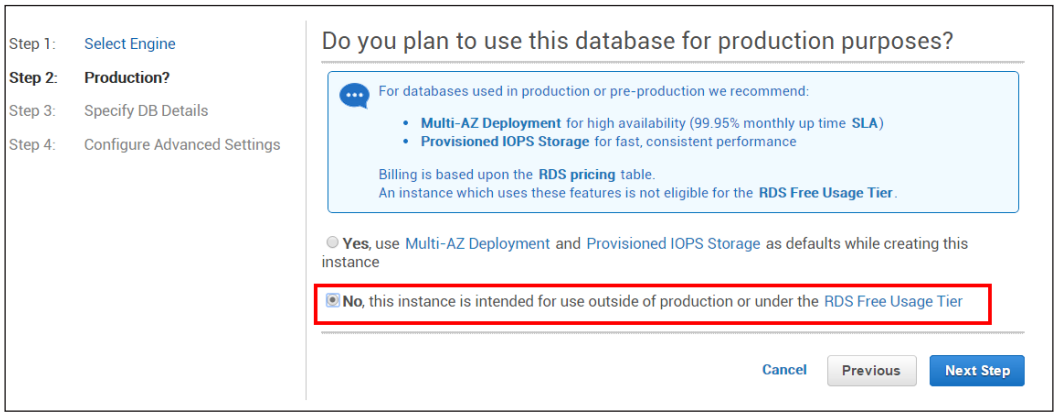
As we have seen, the `A1ElectronicsEcommerce` EC2 instance is up on the cloud; we now need to create a RDS instance within our VPC for our `A1ElectronicsEcommerce` web application. Since we have already defined it under the *Creating security groups* section, it is now just a matter of wiring it to the RDS instance. Perform the following steps:

1. From the RDS dashboard, click on **Launch a DB 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 Community Edition** and click on **Select**. Perform the following steps:

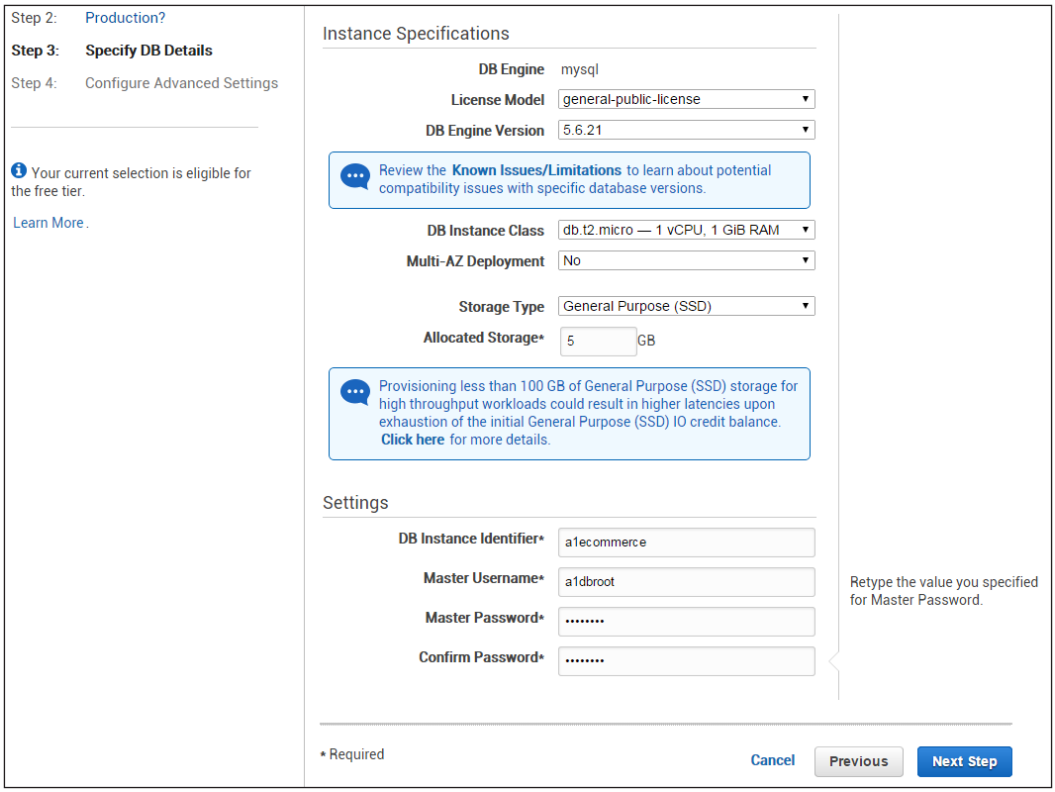


3. The next step is to decide whether the RDS DB instance will be used for the production environment or outside of it. Under the production environment, RDS provides an option for high availability. It also provides an option of provisioning IOPS for your RDS DB instance as per your application's need. All this sounds good, but the costs can add up quickly, so unless you have a convincing business case, avoid this choice. We select the **No** radio button and click on **Next Step**. Let's take a look at the following screenshot:



4. The next step is to configure the RDS instance. The following are the properties:
    - **License Model**: Since we chose **MySQL Community Edition**, **general-public-license** is the only option available.
    - **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.
    - **DB Instance Class**: This is the same as choosing the EC2 instance type. This will select the virtual server that 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 that is available for the free tier.
    - **Multi-AZ Deployment**: This option is for high availability as discussed earlier. Select **No** from the drop-down list.
    - **Storage Type: Select General Purpose (SSD)**: The other options are **Provisioned IOPS**, which kicks in only if your allocated storage is 100 GB or more, and **Magnetic**, which is slower.
    - **Allocated Storage**: You can use the default that is 5 GB. The free tier allows storage up to 20 GB.
    - **DB Instance Identifier**: This is the identifier for the MySQL server database instance, and this identifier is used for defining the DNS entry for the DB instance. Type `a1ecommerce` in the text field.
    - **Master Username**: This is the master login name to access the DB instance; it needs to start with a letter. Enter `a1dbroot` for the master username.
    - **Master Password**: This is the password for the master username.
    - **Confirm Password**: Type in the master password again.

5. Click on **Next Step** for advanced settings, as shown in the following screenshot:

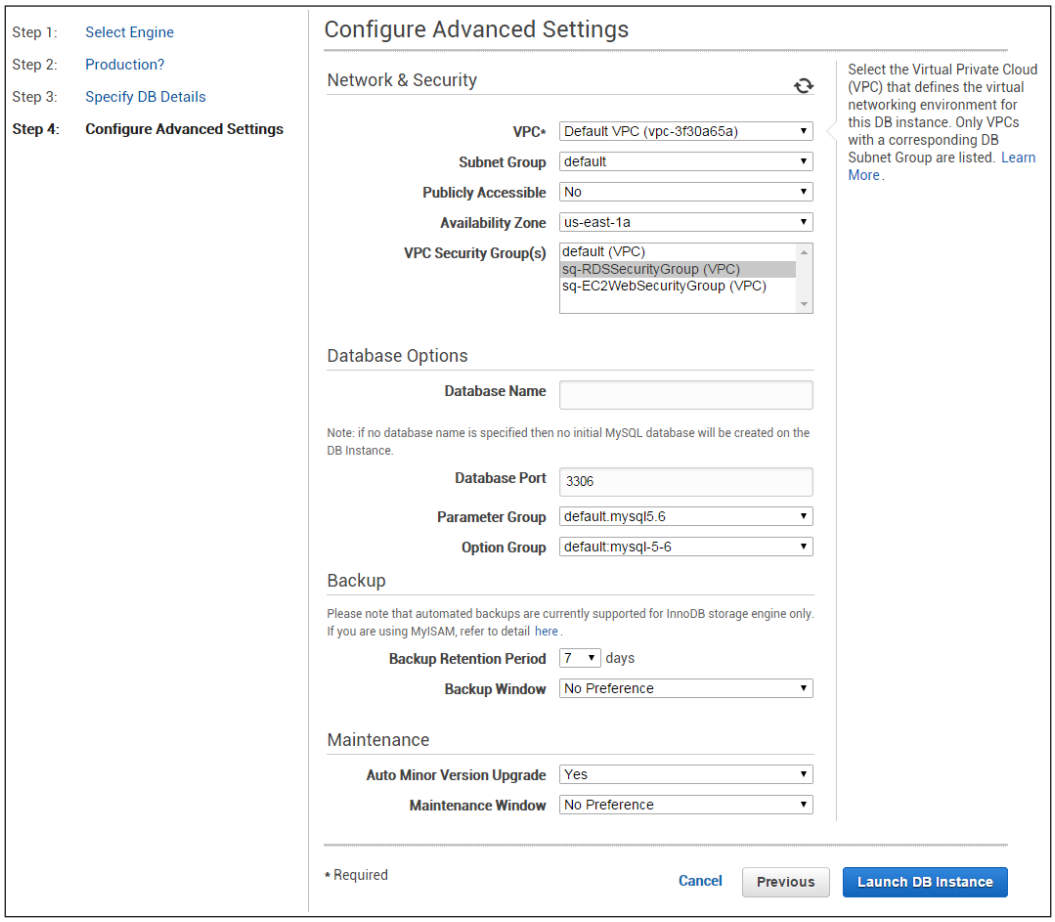


6. Next, we configure Advanced Settings, which have the following properties:
    - **VPC**: This is the VPC network where the DB instance will reside. It is the same default VPC network in which our EC2 instance resides. Since there is only one VPC defined, select the Default VPC from the dropdown.
    - **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. From the architecture diagram, this DB instance is in the same subnet as the EC2 instance. This can be achieved by selecting the correct availability zone. Select **default** from the dropdown.

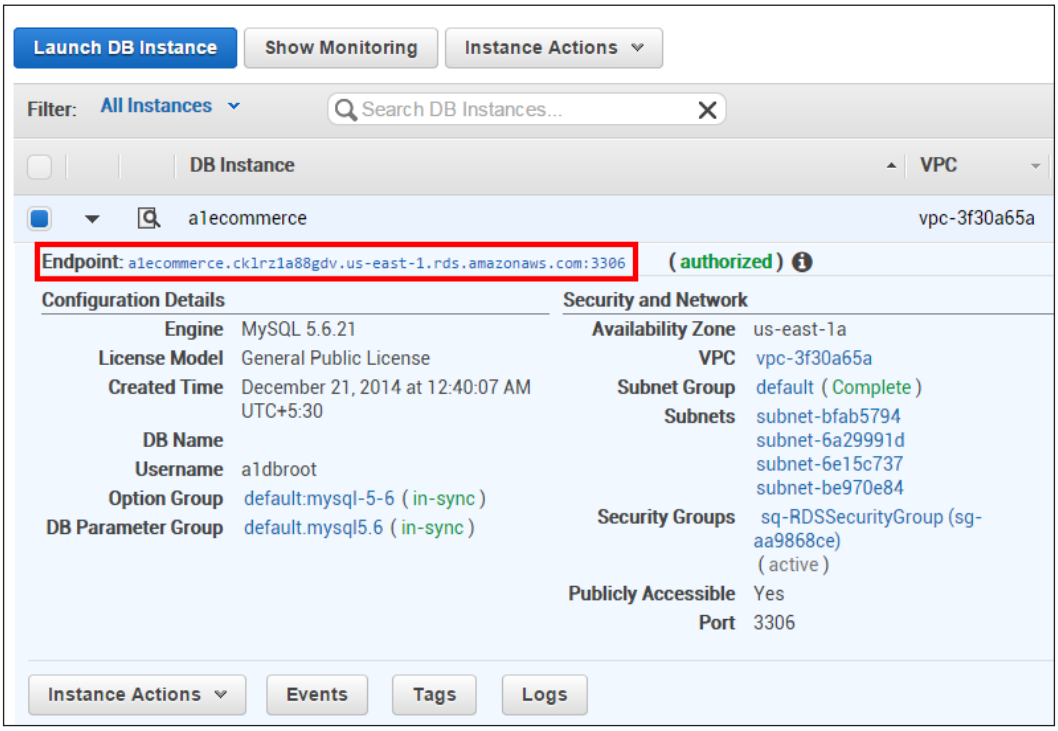
- **Publicly Accessible**: This is a good security practice to hide your databases from the Internet. However, access to the DB instance is only possible after remotely logging into the EC2 instances running within the same VPC or by setting up SSH tunnels. During the development phase, this becomes very inconvenient and frustrating to manage database schema changes, viewing data, and debugging. So by keeping things simple, select **Yes** from the dropdown. 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.
- **Availability Zone**: Select **us-east-1a** from the drop-down box, which is the same as where our EC2 instances are deployed. It assigns the correct subnet to the DB instance.
- **VPC Security Groups**: Select **sq-RDSSecurityGroup** from the list box (created earlier in step 1).
- **Database Name**: This is the name of the database to which an application connects to. Name it `a1ecommerceDb.Name`.
- **Database Port**: This is the default MySQL port. Do not change the default port number, which is set to `3306`.
- **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**.
- **Option Group**: An option group allows us 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**.
- **Backup Retention Period**: This is the number of days Amazon RDS keeps the automatic backup for the instance. The range is from 1 to 35 days. This helps enable one-click restoration of the data in case of disaster recovery. Selection of **0** days disables backup retention. Select **7** from the dropdown.
- **Backup Window**: This is 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 the development cycle, this can be set to **No Preference**.
- **Auto Minor Version Upgrade**: Amazon RDS will automatically update the DB instance only for minor updates. Select **Yes** from the dropdown.

- **Maintenance Window**: This performs any modifications to the DB instance such as changing the DB instance class, storage size, password, multi availability zone deployment, and so on. 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.

7. Click on **Launch DB Instance**, this will create a DB instance and launch it. Let's have a look at the following screenshot:



8. 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 notice there is no IP address associated with the DB instance; the only way you can access this DB instance is via the endpoint. Let's have a look at the following screenshot:



Software stack installation

The next step is to remotely log in to the EC2 instance and install the Apache Tomcat and MySQL client libraries. We will use the private key file created and downloaded under *Creating EC2 instance key pairs*. Perform the following steps:

1. Copy the private key to the `.ssh` folder in your home directory; if for some reason it does not exist, then create it and make sure it has read/write/executable rights assigned for the owner (`drwx------`). To log in from your Linux command line, type the following to assign correct rights to the private key we downloaded in step 2. 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. Use the following command:

```
chmod 700 ~/.ssh/ec2AccessKey.pem
```

- Remote login: After executing the command in the previous step, we can login remotely. The default user name for Ubuntu AMIs is ubuntu, and the IP address to connect to the EIP of the EC2 instance is 54.172.48.64 (from step 5). Type `Yes` when you get a warning that the authenticity of the host `54.172.48.64` can't be established:

```
ssh -i ~/.ssh/ec2AccessKey.pem ubuntu@54.172.48.64
```

- 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-5.6;
```

- Verify Tomcat7 installation: Open any browser and type `http://54.172.48.64:8080`. You will see a default Apache Tomcat page on the browser.
- Verify the MySQL access from the EC2 instance: From the EC2 instance command line, type key in the endpoint (URL) exactly as displayed on the RDS instance dashboard:

```
mysql -ua1dbroot -p -h a1ecommerce.cklrz1a88gdv.
  us-east-1.rds.amazonaws.com
```

When prompted for the password, type the password you entered while creating the DB instance in step 6. At the end of it, you should see a MySQL command prompt.

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

2. Now, we have the DB instance configured, the Tomcat webserver is up and running, and all we need to do next is to deploy the `a1ecommerce` application on the EC2 instance. Before the application can be deployed on the EC2 instance modifications are required in the source files:

   - Point to correct configuration file: Set up the drivers and the database schema to use with RDS by changing line 39 in PersistenceContextConfig.java in `src/main/java/com/a1electronics/ecommerce/config` folder to `@PropertySource(value = { "classpath:application-cloud.properties" })`

- Change the database endpoint: In the `application-cloud.properties` file in the `src/main/resources` folder, change the following properties:

```
jdbc.url=jdbc:mysql:// a1ecommerce.cklrz1a88gdv.us-east-
  1.rds.amazonaws.com:3306/
    a1ecommerceDba1ecommerceDba1ecommerceDb #Endpoint of
      AmazonAmazonAmazon RDS
jdbc.username=a1dbroot # username of Amazon DB instance
jdbc.password=a1dbroot #Password for the Amazon DB instance
```

- After modifying the `data-access.property` 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, type the following command:

```
mvn package
```

This will create an `a1ecommerce.war` file. In the target folder, copy this to the EC2 instance for deployment:

```
scp –i ~/.ssh/ec2AccessKey.pem taget/a1ecommerce.war
ubuntu@54.172.48.64:~/
```

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
```

This deploys the WAR file to the Apache Tomcat web server.

- You have successfully completed deploying a web application on the Amazon cloud. To verify this, in the browser type `http://54.172.48.64:8080/a1ecommerce`, and you should see the `A1Electronics` e-commerce 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, and deployment services. Next, we described some techniques for lowering 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 for achieving application scalability. Next, we will describe the AWS auto scaling 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 auto scaling rules.

4
Designing for and Implementing Scalability

In this chapter, we will introduce some key design principles and approaches to achieving scalability in your 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, Service 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 our sample application.

In this chapter, you will learn the following topics:

- 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 your application architecture to be scalable in order to leverage the highly scalable infrastructure services provided by AWS cloud. Your application should respond proportionally to the increase in resources consumed, and be operationally efficient and cost effective. For example, if you lift-and-shift your on-premise application to the cloud and vertically scale your instances to meet increasing load, then it is likely that it will either become very expensive, your application's increasing resource requirements will necessitate another move to larger instances soon, or both. Hence, it is vital that you design your application to work together with the infrastructure to meet your scalability requirements. In order to design, implement, and operate effectively, you should define an initial set of scalability metrics, for example, the number of requests to be served per second, average and peak number of simultaneous users to be supported, and so on. These metrics will help you establish an initial baseline that you can then benchmark against. This will also let you set appropriate targets for your developers and operations staff, and help you iteratively optimize your designs, processes, and costs. In most cases, if you split your application into small components and then optimize them using the AWS infrastructural features, then you will obtain the best results.

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 the cloud context. Let's have a look at a few of these design principles.

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

Leverage AWS PaaS services wherever possible to receive the benefits of scalability and availability without the associated administrative headaches or design complexity. For example, you can leverage the RDS or the DynamoDB services available for scalable relational and NoSQL database services, respectively. Similarly, you could leverage the AWS SQS for a highly scalable queuing service without having to roll out your own implementation or managing an open source product deployed in an EC2 instance.

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 leverage elasticity of the AWS cloud infrastructure. For example, you can choose the right sizes and number of EC2 instances you need automatically and on-demand, to meet varying requirements.

Implement loosely coupled components

Loosely coupled applications are typically implemented using message-oriented architectures. You can use the AWS SQS service for this purpose. SQS queues are commonly introduced between components to buffer messages. This ensures that the application will perform in situations of high concurrency, unpredictable loads, and/or load spikes.

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

In addition, you should try to design your components to be stateless as far as possible. This will help you distribute your components more effectively. In situations where you need to store the session state, ensure that you do so at a central location so that it is accessible from any instance serving user requests. This is especially important in the auto scaling context where the number of instances varies in response to the demand.

Implement asynchronous processing

Implementing asynchronous processing wherever possible in your application can improve scalability. However, ensure you include sufficient information in your logging records to be able to trace and troubleshoot problems. This is typically done using AWS SQS queues. Ensure you implement a dead letter queue for queue requests that fail after several retries (usually between 3-5 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 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. CloudFront can be used to deliver your entire site including static, dynamic, streaming, and interactive content. It can also work with a non-AWS origin server that stores your original content.

Static content or files include CSS, HTML, images, and so on that are stored in Amazon S3 (and not on your web server instance). This can reduce the 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 reducing overall costs (by reducing the size or the number of EC2 instances required for your web servers).

In the case of dynamic content, for example, 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 mobile application responses.

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 rollout new versions of your application behind the ELB with no service interruption to your customers. You can also deploy your web application on EC2 instances behind ELBs and use Amazon CloudFront to deliver your entire site.

Implementing auto scaling using AWS CloudWatch

The auto scaling feature offered by the AWS cloud infrastructure services allows you to automatically increase or decrease the numbers of instances supporting your load dynamically. The most common way to set this up is by using the AWS CloudWatch service. You can use AWS CloudWatch to measure CPU utilization, network traffic, and so on. You can also define custom metrics for your application. Using these metrics you can setup appropriate thresholds for auto scaling.

ELBs can work within an AWS region only. However, they can work across multiple availability zones within a region. Typically, AWS Route 53 is used for distributing traffic across multiple AWS regions.

Scaling data services

There are several options for data services available from AWS that are optimized for specific use cases. Choose the most appropriate one as per your application's needs. For example, you can choose the RDS service to use the MySQL databases and create read replicas for use in your reporting applications. Depending on your specific requirements, the read replicas can be hosted on a different class of machines (larger or smaller instances) than your RDS instance. This not only serves your application needs efficiently, but also helps 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 80-90 percent) 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 and with ElastiCache to further reduce your costs of RDS usage.

Refer to the extensive documentation available from Amazon for architectural blueprints, technical blogs, white papers, and videos for in-depth guidance on effective scalability strategies to follow for each of the AWS services. In addition, events such as AWS re:Invent, webinars, and meetups with Amazon architects are great sources of information on scalable architectures.

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 a flash sale or in response to breaking news to handle a huge surge in traffic. Remember to benchmark how long it takes for your application components to come up and be available to service requests. This will help you accurately deploy your resources in a timely manner.

Setting up auto scaling

This section introduces you to dynamic scaling for your deployed application. As explained, the application will either scale-out, that is, more EC2 instances will be added or scaled in, that is, running EC2 instances will be removed based on some measureable 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

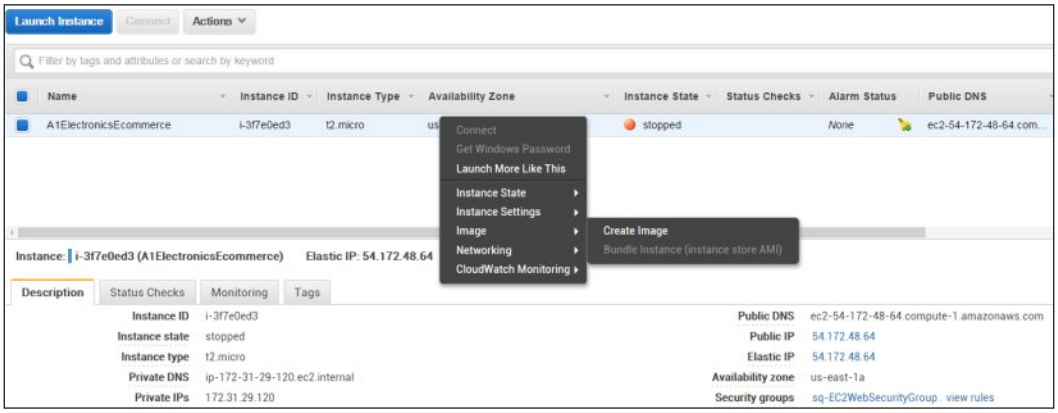
To create a working AWS auto scaling, we will create an **Elastic Load Balancer** (**ELB**), a base AMI (which will be our EC2 instance running our e-commerce application), Launch Configuration (that is, the base AMI to launch in an EC2 instance), and alarms in CloudWatch in order to add/remove instances that apply to an **Auto Scaling Group** (**ASG**). Perform the steps listed in the following sub-sections to setup auto scaling for your application.

Creating an AMI

An **Amazon Machine Image** (**AMI**) is a master image for the creation of virtual servers on the Amazon cloud. An AMI contains instruction to launch an EC2 instance; this includes an operating system, machine architecture 32 bit or 64 bit, software stack for your applications, launch permissions, disk size et.al. You start with the basic AMI that is provided by Amazon, the user community or the market place, and then customize as per your requirements; you can also create an AMI of your running EC2. An AMI is a prerequisite for creating and using an auto scaling group. We will use the `A1ElectronicsEcommerce` instance that we had created in *Chapter 3*, *AWS Components, Cost Model, and Application Development Environments* to create the AMI.

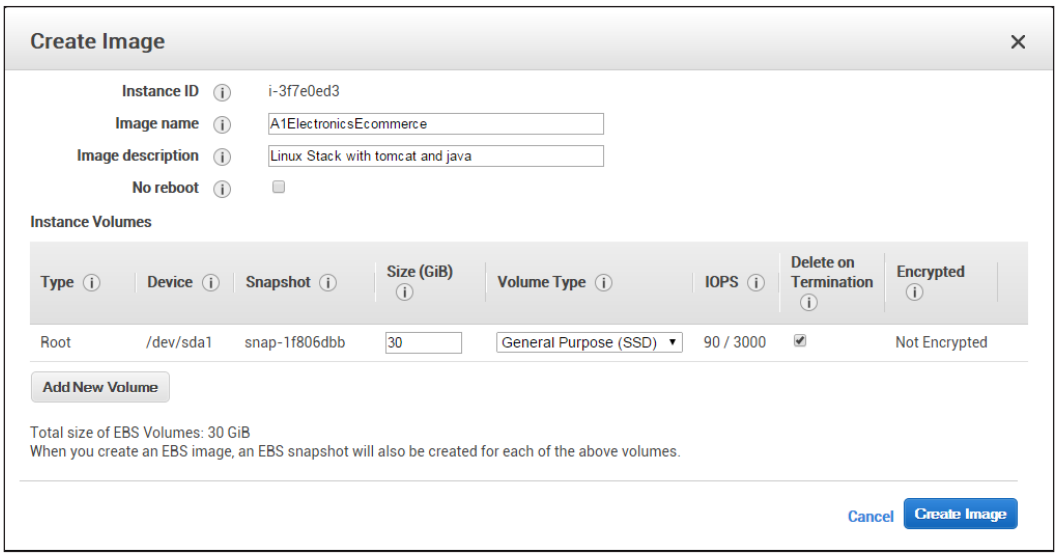
Let's have a look at it:

1. **Creating an AMI**: From the EC2 navigation pane, click on **Instances** to view all your EC2 instances. Select the **A1ElectronicsEcommerce** instance and then right-click on it; this will display a pop-up menu with all the actions you can perform on the selected instance. Select **Image** and then **Create Image** from the menu to create an AMI, as shown in the following screenshot:



2. The next step is to name the AMI and allocate the disk space for 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, 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 instances are created or terminated depending upon the metrics you have configured for. During the launch of an EC2 instance, EBS volumes are created and referenced by the AMI; in our case, it is the **Root** volume. When the EC2 is terminated, the volume is not deleted, so over a period of time you accumulate EBS volumes that you have to pay unnecessarily, to store. As our application is stateless and does not store any data on the instance, we can delete the EBS volume and hence save on the cost. Let's have a look at the following screenshot:



Creating 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 only 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, Nginx as a reverse proxy, or HAProxy. Amazon ELB is designed to handle unlimited concurrent requests per second with a gradually increasing load pattern. It is not designed to handle a sudden increase in requests, such as promotional sales, online exams, or online trading, where you might have a sudden surge of requests. If your use case falls in the latter category you need to request the Amazon Web Service support team to prewarm the ELBs to handle the sudden load.

The ELB consists of three parts; they are as follows:

- **Load Balancer**: This monitors and handles the requests coming in through the Internet/intranet and distributes them to EC2 instances registered with it.
- **Control Service**: It automatically scales the handling capacity in response to incoming traffic by adding and removing load balancers as needed, and also does a health check on the load balancers.
- **SSL Termination**: ELB provides SSL termination that saves precious CPU cycles encoding and decoding SSL within your EC2 instances attached to the ELB. All it requires is a X.509 certificate to be configured within the ELB. It is optional; you still have an option of terminating the SSL connection in the EC2 instance as well.

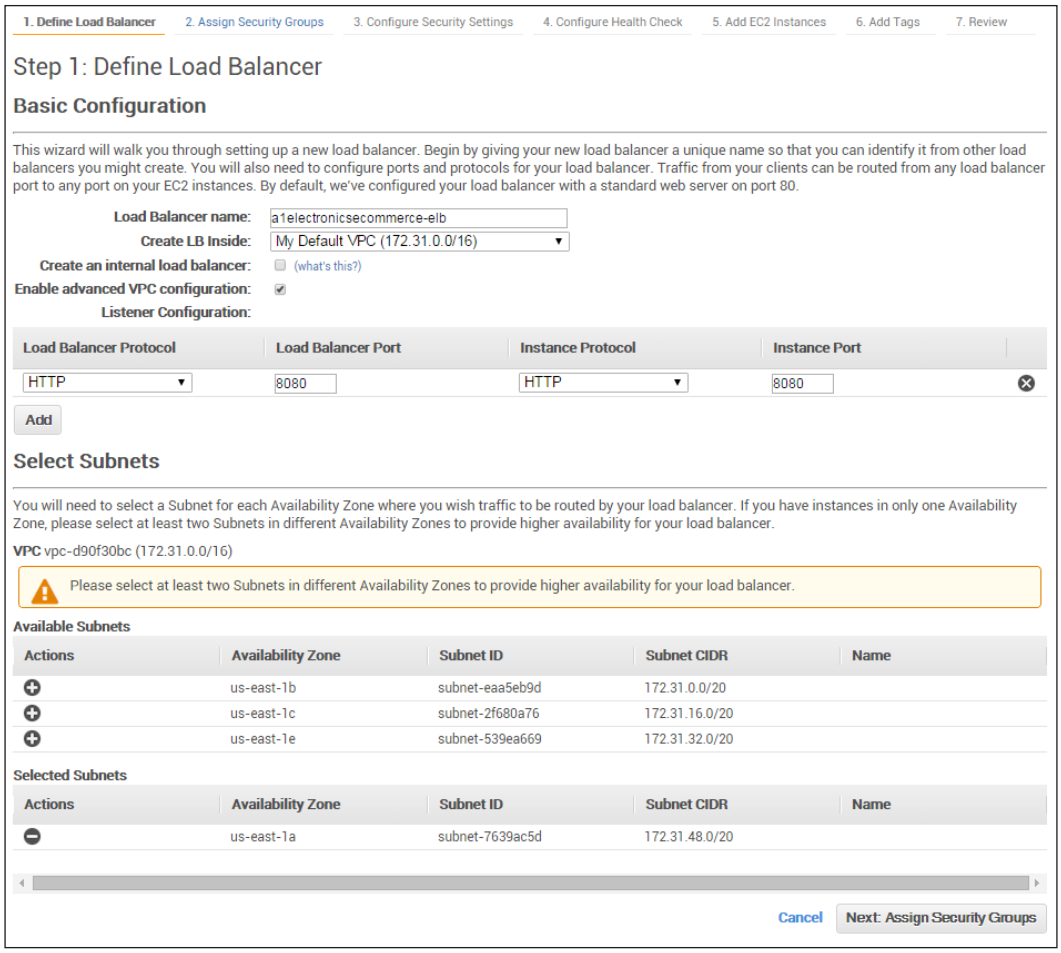
Let's begin the creation of ELB:

1. From the EC2 navigation pane, click on **Load Balancers** under **Network & Security** and then on **Create LoadBalancer**. The fist step is to name the load balancer and configure the protocols it will service. They are as follows:
    - **Load Balancer name**: This is a name that uniquely identifies a load balancer. This name will be a part of the public DNS name of your load balancer.
    - **Create an internal load balancer**: An internal load balancer balances the Intranet traffic in a private subnet, that is, between your internal servers, for example, between web servers and application servers. Do not select this option.

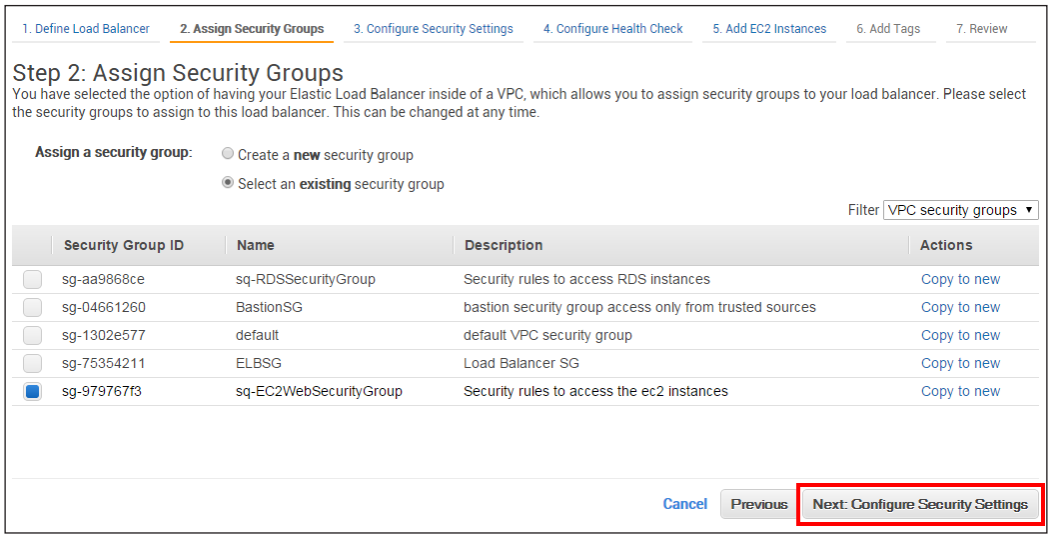
Enable this option only when you want to load balance traffic between internal servers.

    - **Enable advanced VPC configuration**: The advanced VPC configuration option allows you to specify your own subnets. Select this option if you want to route traffic to EC2 instances running in specific availability zones. Select this option as we want to route the traffic to EC2 instance running in availability zone us-east-1a. The other use case is when we want to design for high availability, it is discussed in detail in *Chapter 5*, *Designing for and Implementing High Availability*.

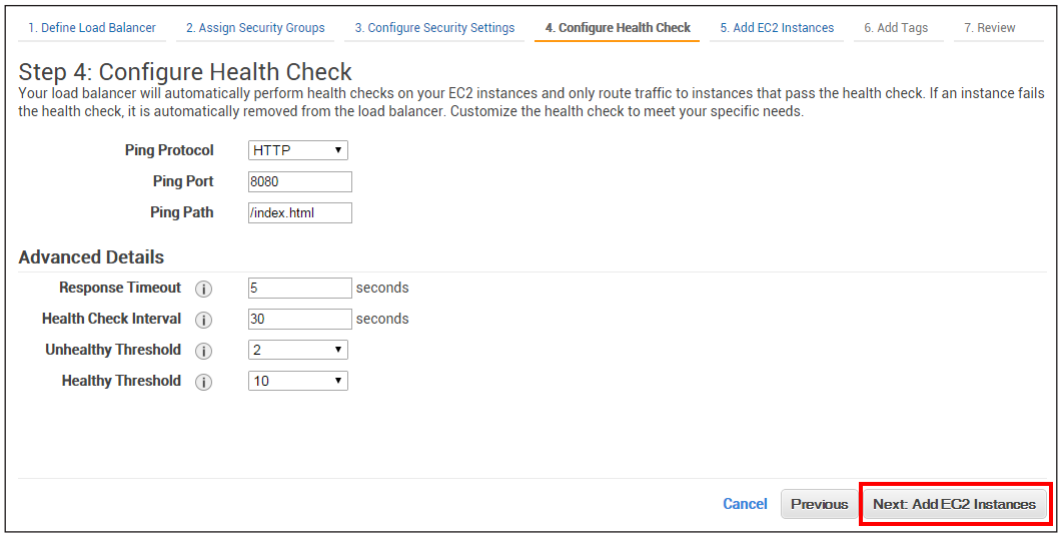
- **Listener Configuration**: A listener is the process that listens for incoming requests of a protocol on a specific port from the client's side and relays it to an EC2 instance configured for a protocol and a port. It supports protocols both at the transport layer (TCP/SSL) and the application layer (HTTP/HTTPS). Our Apache Tomcat server listens on port 8080; we enter port 8080 both on the **Load Balancer Port** and the **Instance Port**. The acceptable ports for both HTTPS/SSL and HTTP/TCP connections are 80 and 443 and between 1024-65535. Select **HTTP** as the protocol on both **Load Balancer Protocol** and **Instance Protocol**.
- 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 such that each unique subnet is associated with an availability zone. The purpose of this is to make your application resistant to failures. If an availability zone goes down all the instances in that availability zone will not respond and hence will be detected via the ELB's health check. The ELB will then start routing the incoming requests to your healthy instances running on the other availability zones within the same region. Since our application is deployed in **us-east-1a** availability zone we add to our **Selected Subnets** by selecting it from the **Available Subnets** list. This does not imply that by simply selecting the subnets from the availability zone will magically auto scale your application. You should have an EC2 deployed in that subnet. Ignore the warning **Please select at least two Subnets in different Availability Zones to provide higher availability for your load balancer** as it for setting up a high availability deployment, more of it in *Chapter 5*, *Designing for and Implementing High Availability*.



2. Now, we assign a security group to our ELB. We have already created the security group **sq-EC2WebSecurityGroup** in *Chapter 3*, *AWS Components, Cost Model, and Application Development Environments* under the *AWS Cloud Construction* section. Select the **sq-EC2WebSecurityGroup**:



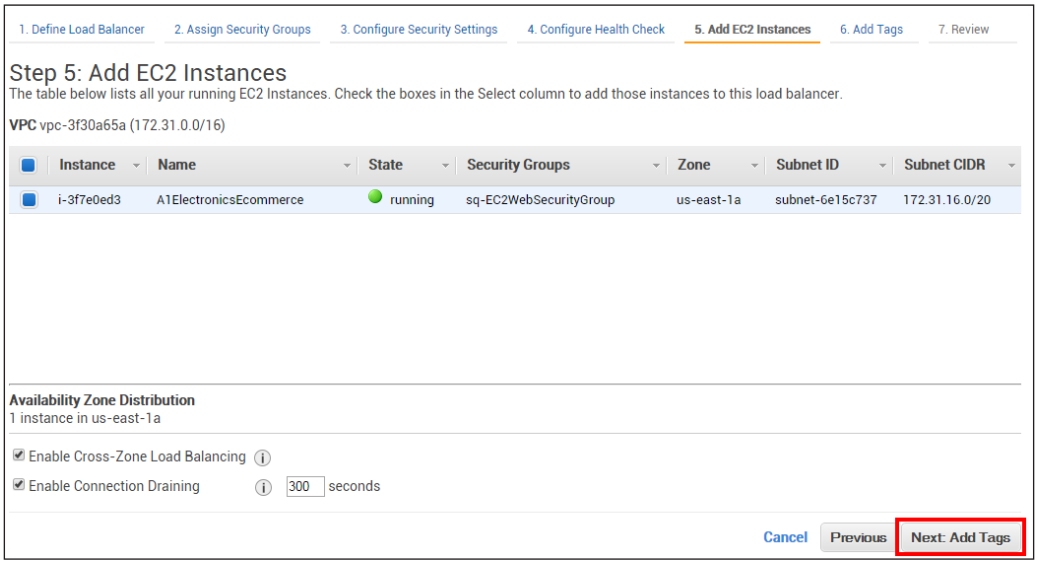
3. The next is to configure the **Security Settings**, since SSL is not being used with the ELB, this step is skipped. This section will be revisited later in *Chapter 6*, *Designing for and Implementing Security*.
4. Next, we configure the health check. ELB periodically sends requests to test the availability of the EC2 instances registered with it. These tests are called health checks. EC2 instances that respond to pings at the time of the health check are marked as `InService` and the instances that do not respond 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. ELB will route requests only to `InService` instances. In the following section, you will define what an `InService` and `OutOfService` EC2 instance is. In your web application, you define a URL that 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. Let's have a look at the following screenshot:



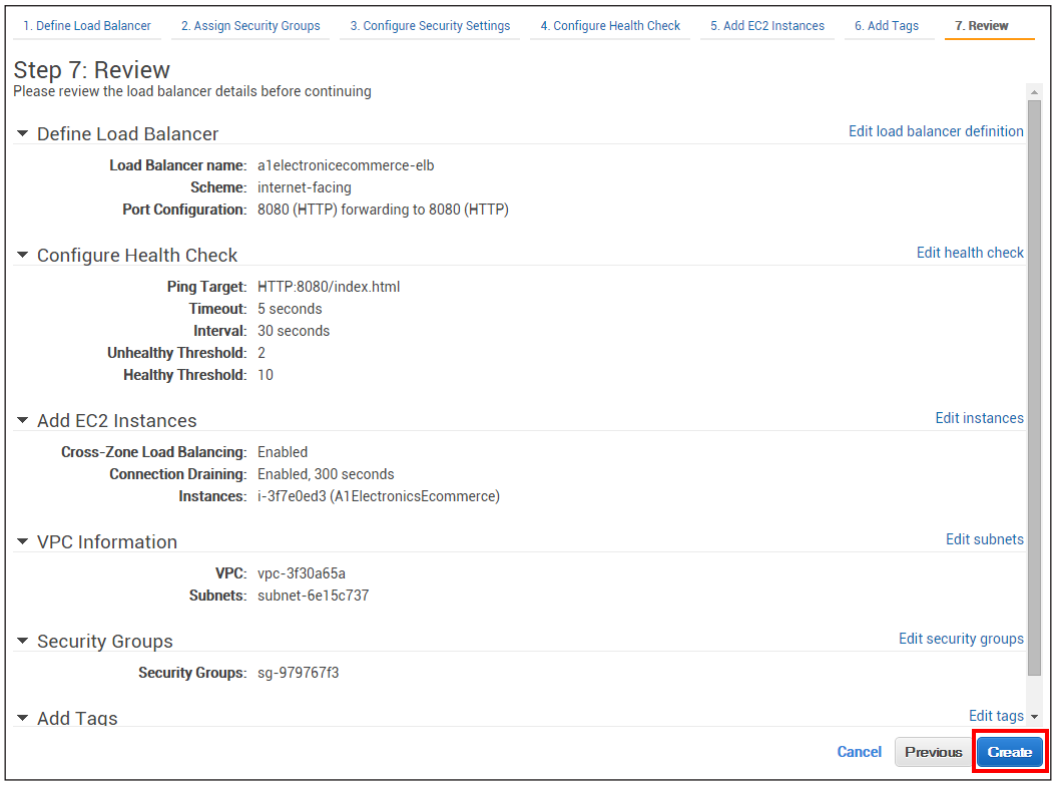
- **Ping Protocol**: This is the protocol to connect to on the EC2 instance. It can be TCP, or HTTP/HTTPS. Select **HTTP** from the dropdown.
- **Ping Port**: This is the port to connect to with the instance. Enter `8080`, which is the default port of our Apache Tomcat server.
- **Ping Path**: This is 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**: This is the 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**: This is the amount of time in between the health checks. If you have 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**: This is the 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 of *HealthCheck Interval * Unhealthy Threshold* seconds. Use the default value `2`. If you do not want your servers taken offline after two consecutive bad health checks, then you can increase this value. This is typically done for applications requiring longer startup time.

- **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`.

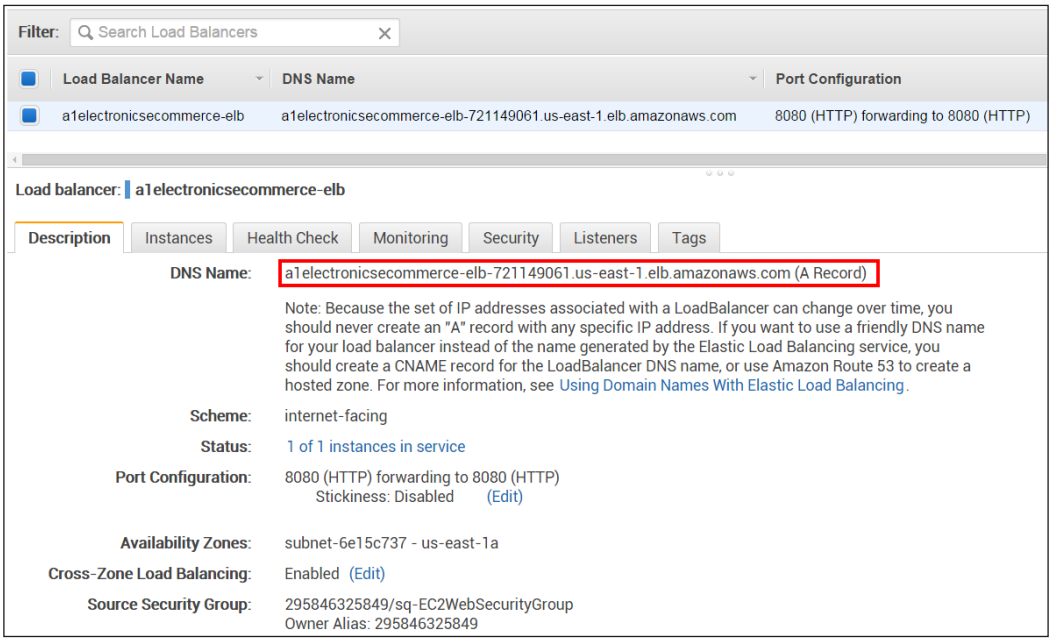
5. Next, we add any running instances we have to ELB. As we are creating this ELB for an auto scaling group, we can skip this. The auto scaling group when activated will add the EC2 instance to the ELB on the fly:
   - **Enable Cross-Zone Load Balancing**: This option allows the ELB to route traffic across the availability zone.
   - **Enable Connection Draining**: This feature only works when used in conjunction with auto scaling. In auto scaling, the instances are dynamically added or removed depending on the policies defined. The auto scaling should not deregister an instance from the ELB when it is in the middle of processing a request that potentially could mean an unhappy customer. If the ELB is processing a request, then this feature delays deregistering the instance by a predefined time period, thereby, potentially allowing the in-process request to complete. In addition, it stops routing traffic to the instance. After the elapsed time period, the ELB will deregister the instance and hopefully by that time, the instance would have processed all the pending requests. The default time period is 300 seconds, but you can change it as per your application needs. Let's have a look at the following screenshot:



6. Next, we add a tag to the ELB, **Key** is `Name` and **Value** is `A1ElectonicsEcommerce-ELB-us-east1a`. These key-value pairs can be named as per your naming convention if you have one.
7. The final step is to review all the configuration data and change it if required. Click on **Create** to create the ELB, as shown in the following screenshot:



8. 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 `xyz-721149061.us-east-1.elb.amazonaws.com`. In a production environment, you need to associate your custom domain name with your ELB domain name by registering a CNAME with your domain DNS provider registrar.



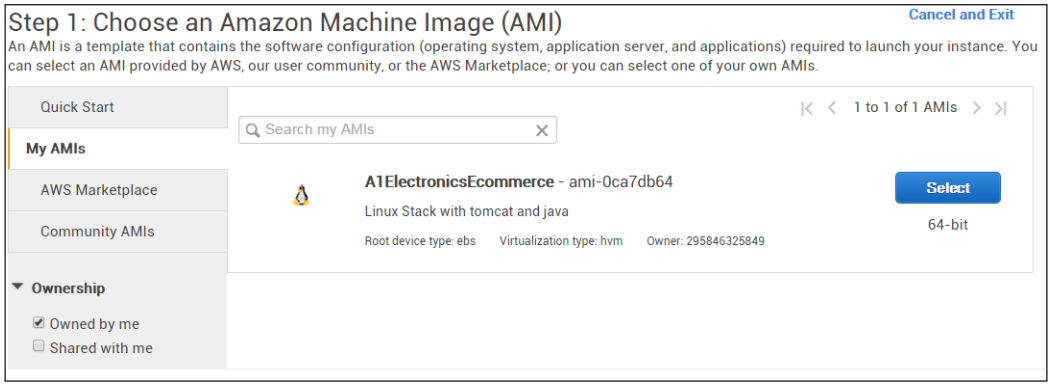
Creating a launch configuration

A launch configuration is a template, which is used by the Auto Scaling Group to select and configure the EC2 instances. This includes configuring IAM role, configuring the IP address, disk size, security group, and public/private key-pair selection to access the instances.

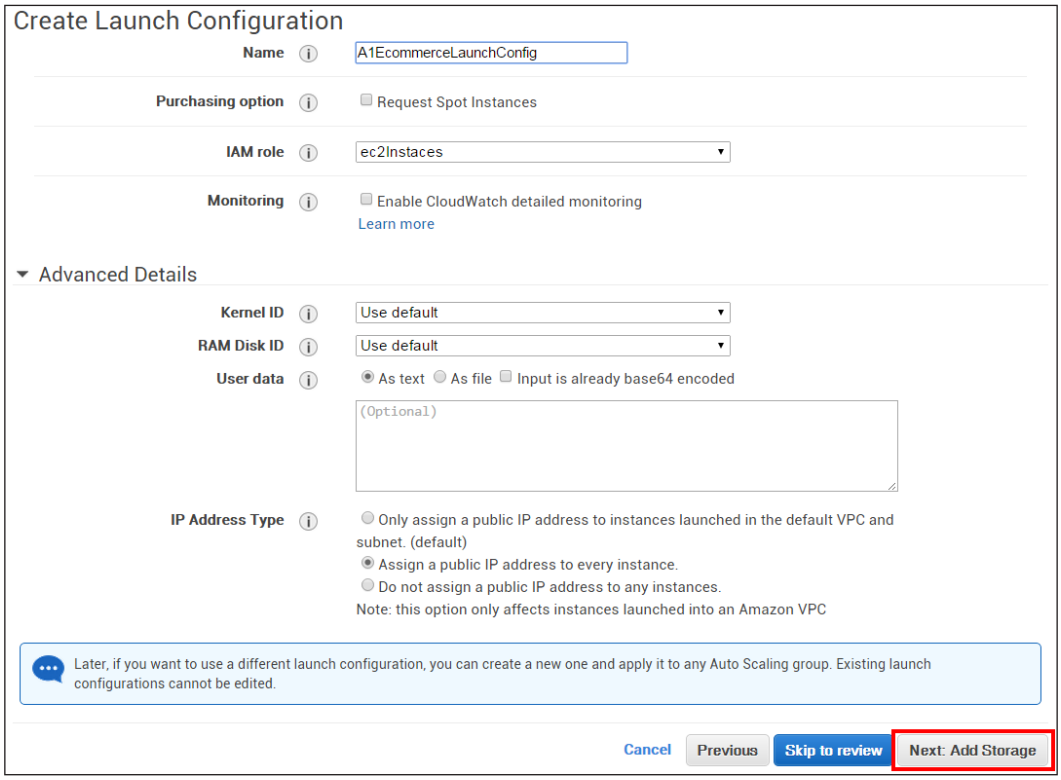
You cannot modify the launch configuration after you have created it. There is a limit of 100 launch configurations per region.

Let's have a look at the steps:

1. From the EC2 dashboard navigation pane, click on **Launch Configurations**, then on **Create Auto Scaling group**, and on **Create Launch Configuration**, to start the launch configuration 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, as shown in the following screenshot:

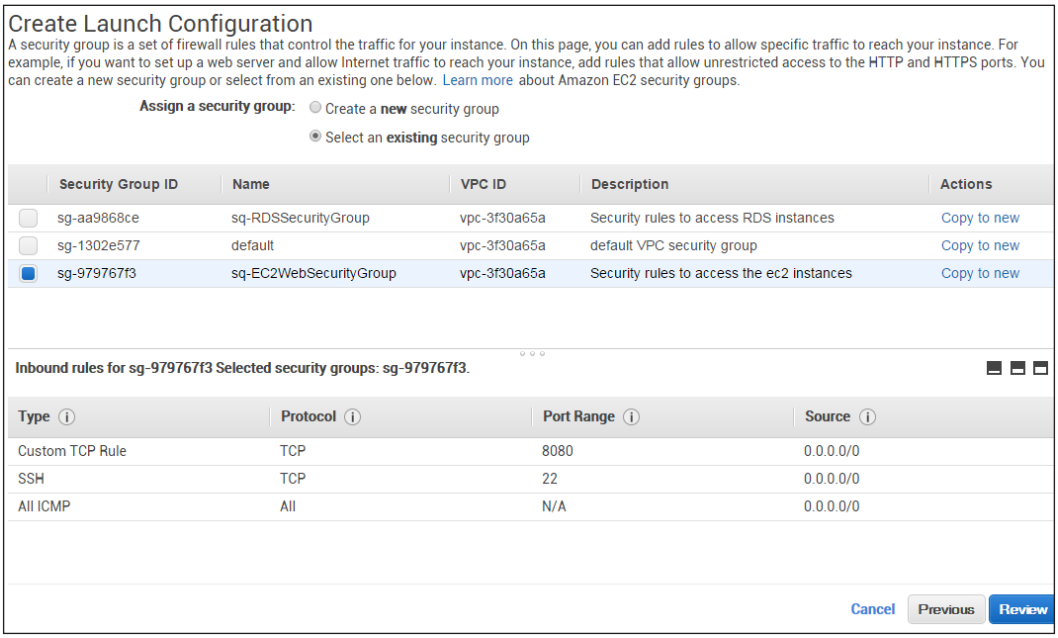


3. The next step is to configure the AMI. Apart from filling in the usual suspect's values such as name, most of the other parameters are already discussed in *Chapter 3, AWS Components, Cost Model, and Application Development Environments* under the *Creating an EC2 Instance* section. We only modify **IP Address Type** and select the option of **Assign a public IP address to every instance** from **Advanced Details**, so that we can SSH into it. Let's have a look at the following screenshot:



As a good security practice, in production servers, select the option of **Do not assign a public IP address to any instances**. If you want to access the instances, then you can create an EC2 instance that 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.

4. Next, we add storage to the AMI. Here, use the defaults unless your requirement is for high disk bandwidth.
5. Next, we configure the security group; use the one that was created in *Chapter 3, AWS Components, Cost Model, and Application Development Environments* under the *Creating Security Groups* section. Let's have a look at the following screenshot:



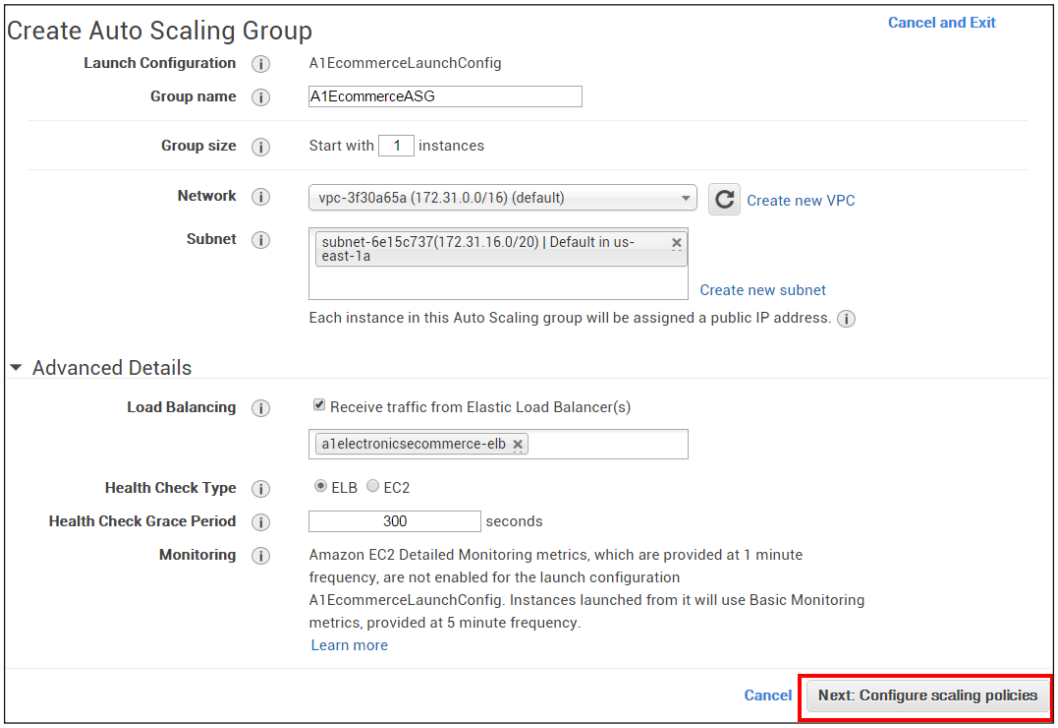
6. Next, we review the launch configuration, make changes if required, and then click on **Create launch configuration**.
7. Before the launch configuration is created, you need to provide the public/private key-pair to SSH into the instance. Select the public/private key created in *Chapter 3, AWS Components, Cost Model, and Application Development Environments* under the *Creating EC2 instance key pairs* section, that is, `ec2AccessKey`.

Creating an auto scaling group

After the launch configuration is created, you are directly taken to the creation of the auto scaling group. Perform the following steps:

1. The first step is to configure the auto scaling group details, which are as follows:
    - **Launch Configuration**: This is the launch configuration for this auto scaling group. This is selected in our case and is set to **A1EcommerceLaunchConfig**.
    - **Group name**: This is the name of 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 with accepted latencies. Before deploying to production, it is good practice to benchmark the application to determine its capacity. Since ours is a demo with the aim to keep the costs to bare minimum, we start with `1` instance.
    - **Network**: Select the VPC where the auto scaling group will launch. Use the default VPC already created in *Chapter 3*, *AWS Components, Cost Model, and Application Development Environments*.
    - **Subnet**: Select the subnet for the auto scaling group within the selected VPC. In our case, there are four subnets representing the four availability zones available within the us-east-1 region. Select the default subnet in the **us-east-1a** availability zone.
    - **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 under *Creating Elastic Load Balancer* section, that is, `a1electronicsecommerce-elb`. If you are using other means of load balancing such as using nginx as a reverse proxy and then unchecking this option.
    - **Health Check Type**: The auto scaling group performs a health check 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 elastic load balancer.

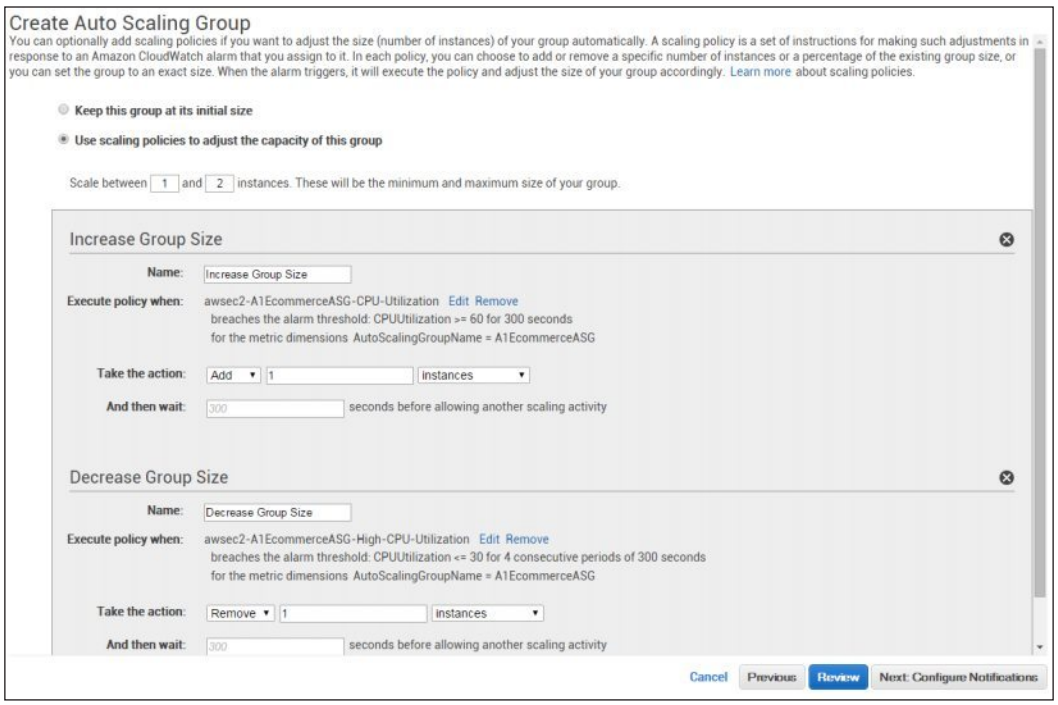
- **Health Check Grace Period**: When an instance comes up, it might take some time before it starts responding to the health checks and pass the auto scaling groups' health check. This time should always be greater than the expected boot time of the instance plus the startup time of the application. In our case, the default value of `300` seconds is fine. Let's have a look at the following screenshot:



2. Next, we configure the scaling policies for the auto scaling group. A policy is a rule that defines how to scale-in our scale-out response to varying conditions. The three options are as follows:
    - **Keep this group at its initial size**: This is a static option. The auto scaling group does not scale-in or scale-out the number of instances defined under group size as defined in step 1. 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 on the conditions. These conditions can be configured as alarms and monitored by CloudWatch. 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 sets the min/max limit for number EC2 instances in the auto scaling group. This is typically set after you know the load you are expecting, and the cost you are ready to bear. For our purpose, set this minimum to `1` and maximum to `2`.
- **Increase Group Size**: Next, we define the policy that will increase the instance in an auto scaling group when an alarm associated with it goes off. They are as follows:
- **Name**: This is the name of the increase group. Use the default value.
- **Execute policy when**: This is where we create an alarm that will be triggered by cloud watch 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 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 total machines running in the auto scaling group or by a fixed number of instances. Since our maximum instances in our auto scaling group are 2, increasing the percentage does not make sense. Percentages work if you have hundreds of instances in your auto scaling group and would want to add multiple instances at a time.
- **Decrease Group Size**: Next, we define the policy that will decrease the instance in an auto scaling group when an alarm associated with it goes off. They are as follows:
- **Name**: This is the name of the increase group. Use the default value.
- **Execute policy when**: This is where we create an alarm that will be triggered by cloud watch when it is breached. Since there are no alarms set, click on **Add new alarm**. See step 6 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-in situation we remove an instance from the auto scaling group. Removing instances can be done either in terms of percentage of total machines running in the auto scaling group or by a fixed number of instances. Since our minimum instances in our auto scaling group is `1` decreasing the percentage does not make sense.



3. While creating an alarm to scale-out, it is very important to know how the alarm is set for scaling out, that is, to add more instances to the auto scaling group. The idea is to trigger the alarm 5-10 minutes before 75 percent of the target threshold is reached. You need to take into account the boot up time of the instance as well as of 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 that can be used are network utilization and memory utilization. A list of EC2 metrics collected by cloud watch is available at `http://docs.aws.amazon.com/AmazonCloudWatch/latest/DeveloperGuide/ec2-metricscollected.html#ec2-metrics`. The alarm we want to set in plain English is `trigger the alarm when the CPU utilization of the EC2 instance is greater than 60% for at least 5 minutes`.
    - **Send a notification to**: Cloud watch uses Amazon **Simple Notification Service** (**SNS**) to send the notifications to the recipients. Click on **Create** to create a new SNS topic to send the message to, name it `A1EcommerceASG-ScaleOut`.
    - **With these recipients**: You can add e-mail addresses of up to 10 recipients in a comma separated format when the alarm triggers. For notifications to work, each recipient must reply on a subscription confirmation mail sent by SNS.
    - **Whenever**: This is a computed aggregation of the metric over a period of time. Select **Average**. The other options are as follows:

      Minimum: This is the lowest value observed during the specified period.

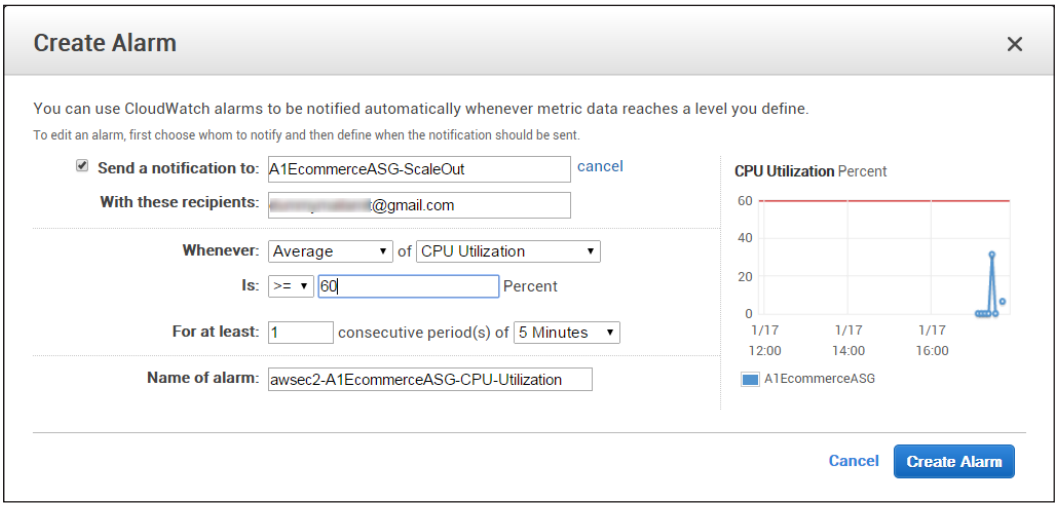
      Maximum: This is the highest value observed during the specified period.

      Sum: This is an addition of all the values during the specified period.

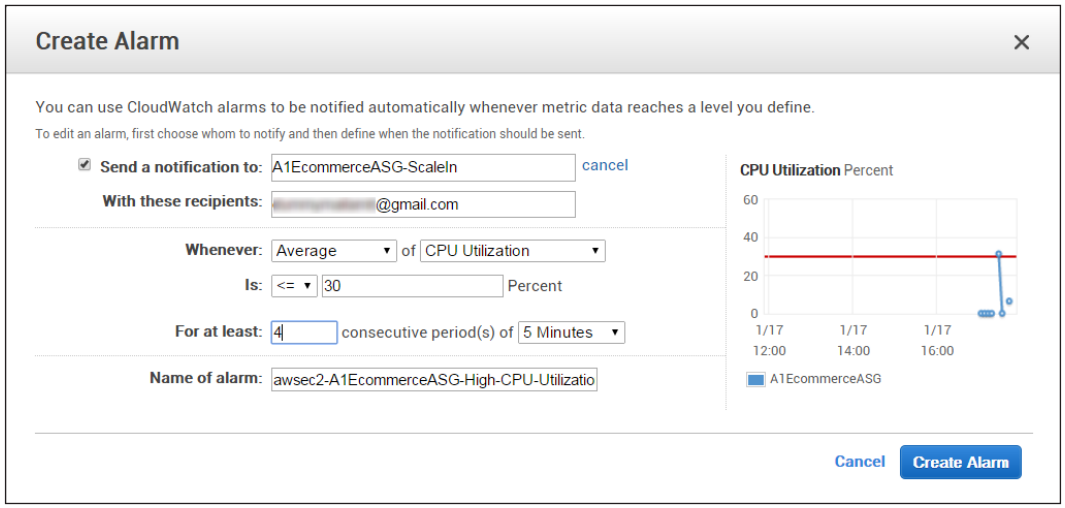
      SampleCount: This is the number of data points used in the calculation.

      Average: This is the value of the Sum/Sample count during the specified period. This gives an average value and can be used to increase or decrease the instance count.

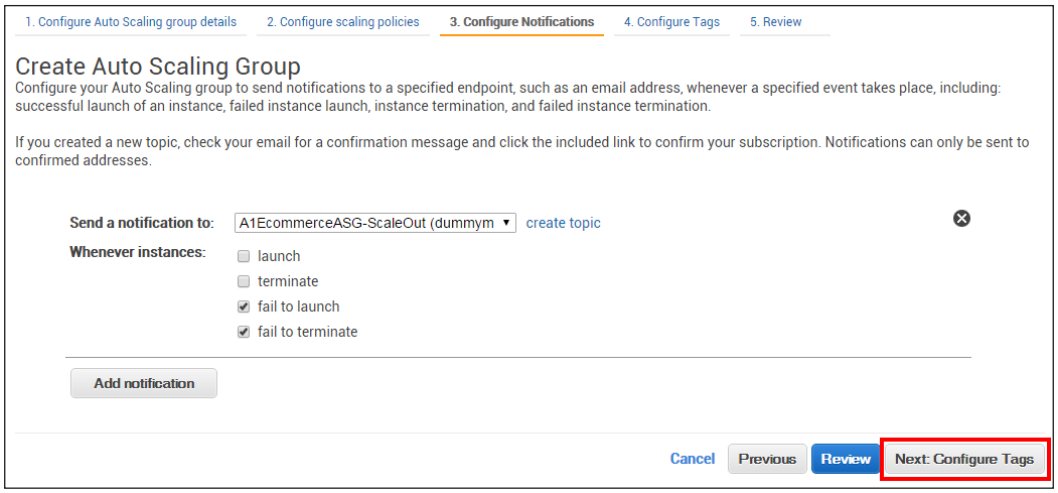
- **Of**: This is the metric we are interested in to monitor and set the alarm against. Select the CPU Utilization, since it's the most logical choice if we want to add or remove instances. The other useful metrics that can be used are network utilization and memory utilization. A list of EC2 metrics collected by cloud watch is available at `http://docs.aws.amazon.com/AmazonCloudWatch/latest/DeveloperGuide/ec2-metricscollected.html#ec2-metrics`.
- **Is**: This defines the metric threshold. In our case, we want to add an instance when the **CPU Utilization Percent** reaches more than 80 percent. Enter `60` as the threshold value with 20 percent headroom to account for irregular spikes and an instance failing to startup. The threshold percentage is the average of all the instances in the auto scaling group and not any specific instance.
- **For at least**: This is the specified period after which the alarm is triggered. By default, the CloudWatch sampling period is 5 minutes, and is offered free of charge. This is 1 minute as a paid service. The parameter for consecutive period(s) cannot be less than the metric selected for the sampling period. Enter `1` in the **For at least** column and select **5 Minutes** from the period(s) dropdown.
- **Name of alarm**: A name is automatically generated, as shown in the following screenshot:



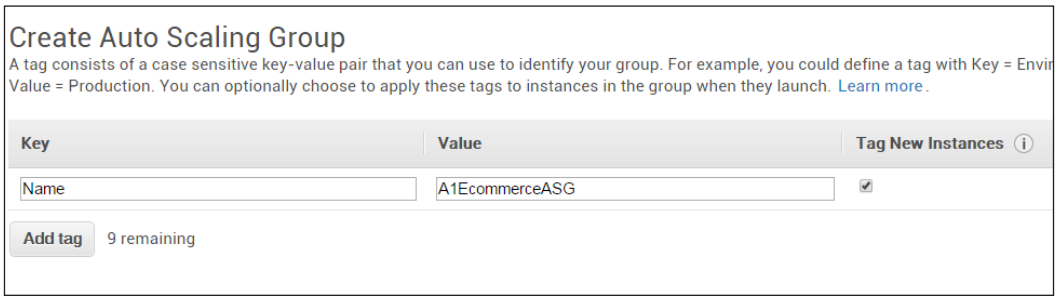
4. While creating an alarm for scaling in, care should be taken to not remove capacity too quickly. This helps avoid rapid cycles that alternate between creation and deletion of instances. This can happen when the scale in policy is aggressive. Take time to scale slowly, that is, scale-in only when the CPU Utilization is less than 30 percent for a period of 20 minutes:
    - **Is**: This defines the metric threshold. In our case, we want to remove an instance when the **CPU Utilization Percent** goes down to 30 percent.
    - **For at least**: This is the specified period after which the alarm is triggered. Enter `4` in **For at least** and select **5 Minutes** from the **consecutive period(s) of** dropdown. This will ensure that the alarm is triggered after 20 minutes, as shown in the following screenshot:



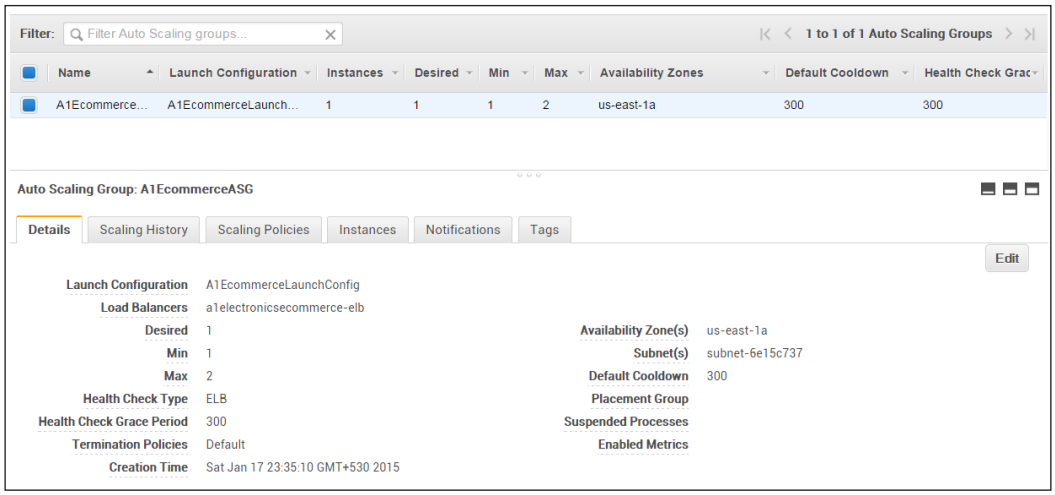
5. Next, we configure the auto scaling group to send the notifications to the Amazon SNS topic whenever a scaling event takes place. Currently, we are only interested in **fail to launch** and **failed to terminate** scaling events. But in production, it is strongly recommended to send all the scaling events, as shown in the following screenshot:



6. 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. Tags are useful when there are several auto scaling groups configured for different layers. In such situations, it becomes easier to identify the EC2 instances in the dashboard. Make sure you to tick **Tag New Instances**. Enter input into **Key** as `Name` and **Value** as `A1EcommerceASG`. Add other tags as per your naming convention, as shown in the following screenshot:



7. 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 cloud watch and topics in SNS. For viewing SNS topics, you can go directly to `https://console.aws.amazon.com/sns/home?region=us-east-1#dashboard` and for cloud watch alarms, `https://console.aws.amazon.com/cloudwatch/home?region=us-east-1#`. When the auto scaling group starts, it starts with the minimum number of instances. To verify that the auto scaling group is working as configured, copy the DNS name of the elastic load balancer as described in step 9 in *Creating Elastic Load Balancer*. In this case, it is `http://a1electronicsecommerce-elb-721149061.us-east-1.elb.amazonaws.com:8080`; please do not forget to add the port number. If the auto scaling group is correctly configured, then you should see the default Tomcat web page, as shown in the following screenshot:



Testing auto scaling group

The next step is to test the auto scaling group. It should add an instance to when the CPU Utilization is greater than 60 percent for 5 minutes and remove an EC2 instance if the CPU Utilization falls in less than 30 percent for 20 minutes. The easiest way to test this is to load the CPU for more than 5 minutes and check whether an EC2 instance is added.

The public IP address of the instance is available from **Instances** in the EC2 dashboard navigation pane:

1. Log in to the instance via `ssh`, using the following command:

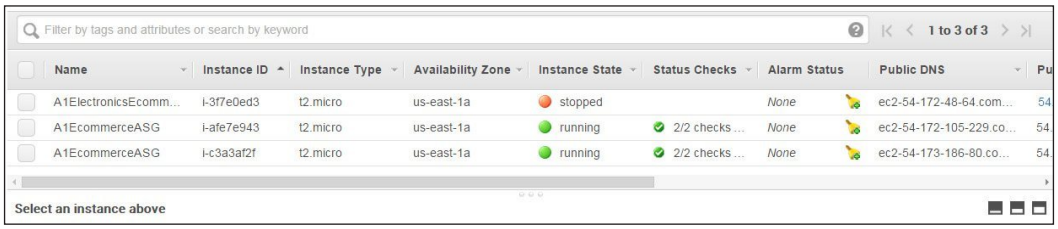
   ```
   ssh -i ~/.ssh/ec2AccessKey.pem ubuntu@54.172.48.64
   ```

2. To tax the CPU use `bc`, which is precision calculator language to compute a value which takes take a lot of time, for example, raising 2 to the 10 billionth power which in turn pushes the CPU to 100 percent:

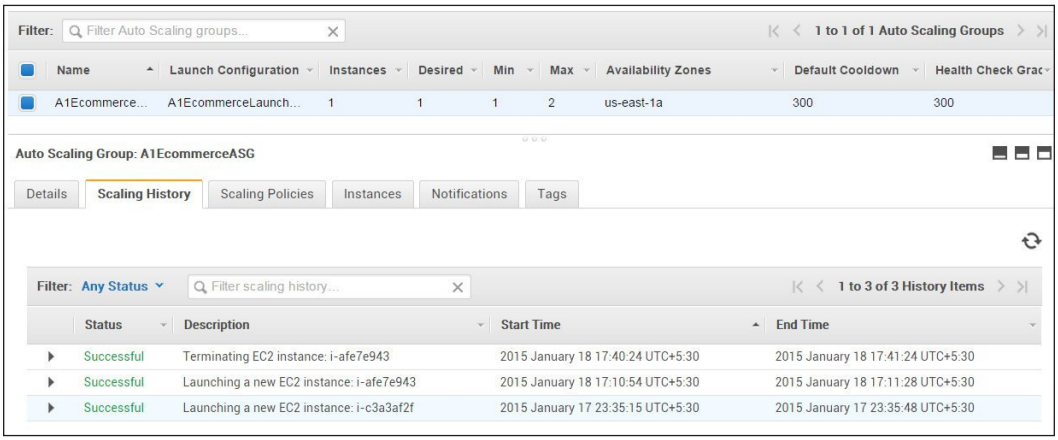
   ```
   echo 2^1234567890 | bc
   ```

Now, we have set the stage for the auto scaling group to add new instance whenever the average load crosses 60 percent for more than 5 minutes. There are multiple ways to verify that an instance has been added when the scale-out alarm is breached. They are mentioned here:

- As we have set a notification with the scale-out alarm, an e-mail will be sent to the configured e-mail address with the all the alarm details
- A new EC2 instance is added to the instances view in the EC2 dashboard, as shown in the following screenshot:



- The auto scaling group view in the EC2 dashboard has a tab **Scaling History**, which displays all the scaling events, as shown in the following screenshot:



In the same way, we can verify the scaling in by the auto scaling group, that is, removal of an EC2 instance when the average CPU utilization of the EC2 instances falls below 30 percent 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 percent and scaling in threshold will be breached after a period of 20 minutes.

Scripting auto scaling

Creating the auto scaling via the user interface is all good when you are doing it for the first time but later on it gets cumbersome and repetitive with potential for errors. The ideal way would be to automate the creation of an auto scaling group via command line script. For the purpose of automation, Amazon provides a command line interface using your favorite language to manage your AWS services. We will now create the complete auto scaling group right from creating an AMI, elastic load balancer and the auto scaling group itself. It is assumed that you are familiar with the Linux command line. The values for the command line parameters are exactly the same as the ones used via the Amazon web interface in the previous section. You will need to change the names if you already have auto scaling working. For complete automation, all the commands described here can be easily incorporated into a shell script. To make this happen, you need to install the Amazon command line library in your development environment.

The installation process described here assumes you have root access:

1. Since Amazon's command line interface is based upon Python, we need Python in our development machine. Check if you have Python installed in your development machine:

   ```
   python –version
   ```

   - If you do not have python installed, then before you proceed further you need to install python version 2.7 or later. A great link to install it under Ubuntu is `http://askubuntu.com/questions/101591/how-do-i-install-python-2-7-2-on-ubuntu`

2. Now that you have Python installed in your development machine, the next step is to install `pip`, which is an installation manager for python packages:

   ```
   sudo apt-get install python-pip
   ```

3. Check the pip installation:

   ```
   pip –help
   ```

4. To install the Amazon command line interface (CLI), use the following command:

   ```
   sudo pip install awscli
   ```

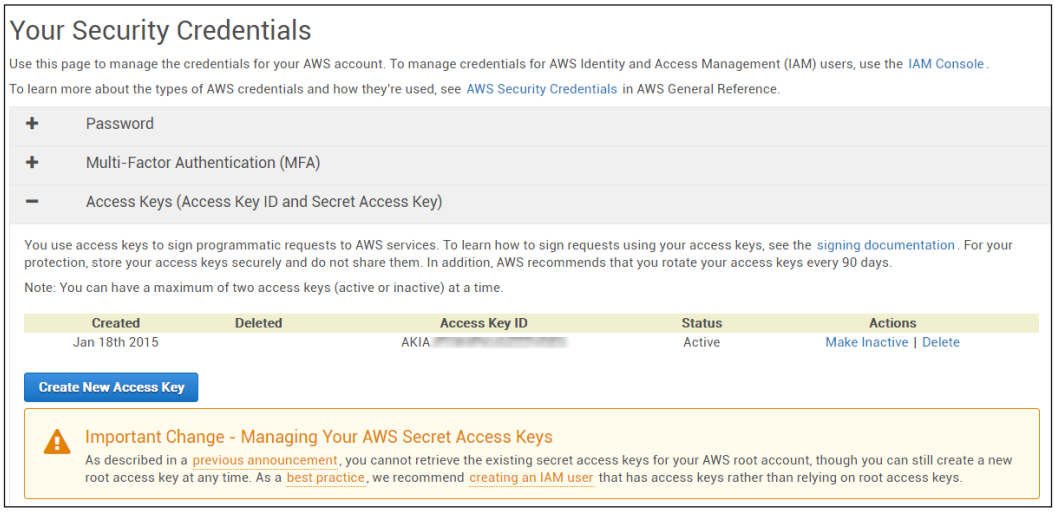
5. To upgrade the CLI, use the following command:

   ```
   sudo pip install --upgrade awscli
   ```

6. The CLI package has an auto completer, which servers the purpose of presenting the probable options, but its installation is specific to the Linux shell you have configured. Make sure you add this command in your `.bashrc` file so it executes every time you login. For bash shell, use the following command:

   ```
   complete -C '/usr/local/bin/aws_completer' aws
   ```

7. The last step is to set up the credentials of your AWS account and the settings to be used by the CLI. If you do not have the access keys you can create it via the IAM management console:




```
aws configure
AWS Access Key ID [None]: Your AWS access key
AWS Secret Access Key [None]: Your AWS secret access key
Default region name [None]: us-east-1
Default output format [None]: json
```

Since the CLI has been configured, let's begin the creation of the auto scaling using the command line.

Creating an AMI

To create an AMI, we need the instance ID of the `A1ElectronicsEcommerce` instance on which the AMI will be based upon:

1. The EC2 instances can be queried via the CLI:

   ```
   aws ec2 describe-instances
   ```

   In response, the EC2 service returns the configuration and status of all EC2 instances running. Since we have tagged our EC2 instance as `A1ElectonicsEcommerce` it becomes easy to locate in the response:

   ```
   "Tags": [
     {
       "Value": "A1ElectronicsEcommerce",
   ```

```
    "Key": "Name"
  }
  ],
```

Instance ID is available in the JSON response (at the same level as `Tags`):

```
"InstanceId": "i-3f7e0ed3",
```

Make a note of the instance ID as it will be used in creating a launch configuration.

2. To create the AMI, use the following command:

```
aws ec2 create-image --instance-id i-3f7e0ed3 --name
  "A1ElectronicsEcommerce" --description "Linux Stack with
    tomcat and java"
```

In response, the EC2 service returns the AMI ID. Make a note of this AMI, as it will be used later in creating launch configuration:

```
{
    "ImageId": "ami-5c790734"
}
```

Creating an Elastic Load Balancer

Creating a load balancer requires you to define the load balancer, configure health check, and add tags. To create ELB, the command line expects the IDs for the VPC subnet, security group perform the following steps:

1. Execute the following command to retrieve the details of your security group:

```
aws ec2 describe-security-groups
```

In response, the EC2 service will configure all the security groups, since we are only interested in **sq-EC2WebSecurityGroup**. From the response, we have `VpcId`:

```
"GroupName": "sq-EC2WebSecurityGroup",
  "VpcId": "vpc-3f30a65a",
  "OwnerId": "295846325849",
  "GroupId": "sg-979767f3"
```

2. Similarly, we can query the subnets using the following command:

```
aws ec2 describe-subnets
```

ELB will be hosted on the us-east-1a availability zone; from the response, we have the `SubnetId`:

```
  {
    "VpcId": "vpc-3f30a65a",
    "CidrBlock": "172.31.16.0/20",
    "MapPublicIpOnLaunch": true,
    "DefaultForAz": true,
    "State": "available",
    "AvailabilityZone": "us-east-1a",
    "SubnetId": "subnet-6e15c737",
    "AvailableIpAddressCount": 4086
  },
```

3. Now, we are ready to create an ELB from the command line:

```
aws elb create-load-balancer --load-balancer-name a1electronicse
commerce-elb --listeners Protocol=HTTP,LoadBalancerPort=8080,
InstanceProtocol=HTTP,InstancePort=8080 --subnets subnet-6e15c737
--security-groups sg-979767f3  --tags Key=Name,Value=A1Electronics
Ecommerce-ELB-us-east-1a
```

In response, the EC2 service returns the ELB DNS name:

```
{
    "DNSName": "a1electronicsecommerce-elb-1387886298.us-east-1.
elb.amazonaws.com"
}
```

4. Next, we enable connection draining, using the following command:

```
aws elb modify-load-balancer-attributes --load-balancer-name
a1electronicsecommerce-elb --load-balancer-attributes "{\"Connecti
onDraining\":{\"Enabled\":true,\"Timeout\":300}}"
```

In response, the EC2 service returns the modified attributes:

```
{
  "LoadBalancerAttributes": {
    "ConnectionDraining": {
    "Enabled": true,
    "Timeout": 300
    }
  },
  "LoadBalancerName": "a1electronicsecommerce-elb"
}
```

5. Next, we associate a health check with the ELB:

```
aws elb configure-health-check --load-balancer-name
a1electronicsecommerce-elb --health-check Target=HTTP:8080/index.
html,Interval=30,UnhealthyThreshold=2,HealthyThreshold=10,Timeo
ut=5
```

In response, the EC2 service returns an added health check:

```
{
  "HealthCheck": {
    "HealthyThreshold": 10,
    "Interval": 30,
    "Target": "HTTP:8080/index.html",
    "Timeout": 5,
    "UnhealthyThreshold": 2
  }
}
```

Creating launch configuration

To create the launch configuration IDs for AMI, a security group is needed and also the key name and IAM role. We have the IDs for AMI (`ami-5c790734`) and security group (`sg-979767f3`). In addition, we have the values for key name (`ec2AccessKey`) and IAM role (`ec2Instaces`) as defined in *Chapter 3, AWS Components, Cost Model, and Application Development Environments*. Let's have a look at the following command:

```
aws autoscaling create-launch-configuration --launch-configuration-
name A1EcommerceLaunchConfig --key-name ec2AccessKeys --image-id ami-
5c790734 --security-groups sg-979767f3 --instance-type t2.micro --
instance-monitoring Enabled=false --no-ebs-optimized --associate-
public-ip-address --placement-tenancy default --iam-instance-profile
"ec2Instaces" --block-device-mappings "[{\"DeviceName\":
\"/dev/sda1\",\"Ebs\":{\"VolumeType\":\"gp2\",\"VolumeSize\":30,\"Del
eteOnTermination\":true}}]"
```

Amazon's auto scaling service doesn't return any response. To check whether the launch configuration is created, you need to issue another command line:

```
aws autoscaling describe-launch-configurations --launch-
  configuration-names A1EcommerceLaunchConfig
```

Creating an auto scaling group

The final step is to create an auto scaling group and set the alarms. To create the auto scaling group from the command line, you need the launch configuration name (`A1EcommerceLaunchConfig`), VPC subnet (`subnet-6e15c737`), and elastic load balancer (`a1electronicsecommerce-elb`):

1. Creating the auto scaling group:

```
aws autoscaling create-auto-scaling-group --auto-scaling-
group-name A1EcommerceASG --launch-configuration-name
A1EcommerceLaunchConfig --load-balancer-names
"a1lecronicsecommerce-elb" --health-check-type ELB --health-
check-grace-period 300 --vpc-zone-identifier subnet-6e15c737 -
-min-size 1 --max-size 2 --tags
Key=Name,Value=A1EcommerceASG,PropagateAtLaunch=true  --
default-cooldown 300 --desired-capacity 1
```

   Amazon's auto scaling service doesn't return any response. To check whether the auto scaling group is created, you need to issue another command line:

```
aws autoscaling describe-auto-scaling-groups
```

2. The next step is to create policies for increasing and decreasing the instances in a group. Two policies will be created, one for scaling out and the other for scaling in. You only need the name of the auto scaling group created in step 1 (`A1EcommerceASG`). Create a policy that will remove instances from the group:

```
aws autoscaling put-scaling-policy --policy-name scalein-policy
--auto-scaling-group-name A1EcommerceASG --scaling-adjustment -1
--adjustment-type ChangeInCapacity
```

   The Amazon auto scaling service returns the Amazon Resource Name (ARN). The ARN will be required to create and/or remove capacity alarm:

```
{
  "PolicyARN": "arn:aws:autoscaling:us-east-
    1:193603752452:scalingPolicy:8b199b0a-88e8-4864-a18e-
      2599282cc909:autoScalingGroupName/A1EcommerceASG:
        policyName/scalein-policy"
}
```

   Now create a policy that will add instances to the group:

```
aws autoscaling put-scaling-policy --policy-name scaleout-
policy --auto-scaling-group-name A1EcommerceASG --scaling-
adjustment 1 --adjustment-type ChangeInCapacity
```

The Amazon auto scaling service returns the ARN. The ARN will be required to create/remove capacity alarm:

```
{
  "PolicyARN": "arn:aws:autoscaling:us-east-
    1:193603752452:scalingPolicy:7d63fd35-19ad-4aa3-9379-
      28d7a33d3701:autoScalingGroupName/A1EcommerceASG:
        policyName/scaleout-policy"
}
```

3. Now we have to create alarms and associate them with the policies. We will create two alarms, one to trigger scale-out and the other to trigger scale-in. Create an alarm that triggers the scale-out policy when CPU utilization exceeds 60 percent for 5 minutes:

```
aws cloudwatch put-metric-alarm --alarm-name scale-out --
alarm-description "Alarm when CPU exceeds 60 percent" --
metric-name CPUUtilization --namespace AWS/EC2 --statistic
Average --period 300 --evaluation-periods 1 --threshold 60 --
comparison-operator GreaterThanThreshold  --unit Percent   --
dimensions Name=AutoScalingGroupName,Value=A1EcommerceASG --
alarm-actions arn:aws:autoscaling:us-east-
1:193603752452:scalingPolicy:7d63fd35-19ad-4aa3-9379-
28d7a33d3701:autoScalingGroupName/A1EcommerceASG:policyName/sc
aleout-policy
```

Create an alarm that triggers a scale-in policy when the CPU utilization falls below 30 percent for 5 minutes and for four consecutive periods:

```
aws cloudwatch put-metric-alarm --alarm-name scale-in --alarm-
description "Alarm when CPU falls below 30 percent" --metric-
name CPUUtilization --namespace AWS/EC2 --statistic Average --
period 300 --evaluation-periods 4 --threshold 30 --comparison-
operator LessThanThreshold  --unit Percent   --dimensions
Name=AutoScalingGroupName,Value=A1EcommerceASG --alarm-actions
arn:aws:autoscaling:us-east-
1:193603752452:scalingPolicy:8b199b0a-88e8-4864-a18e-
2599282cc909:autoScalingGroupName/A1EcommerceASG:policyName/sc
alein-policy
```

Amazon's cloud watch service doesn't return any response. To check whether the alarms are created, you need to issue another command line:

```
aws cloudwatch describe-alarms
```

4. To receive e-mail notifications from the auto scaling group, topics in SNS have to be created and the auto scaling group has to be configured. Two SNS topics will be created, one to route events when the auto scaling group scales out and another when it scales in. Create the two topics:

```
aws sns create-topic --name A1EcommerceASG-ScaleIn
aws sns create-topic --name A1EcommerceASG-ScaleOut
```

The response from the Amazon SNS service is ARN. The ARN will be required to create the subscriptions. Use the following code:

```
{
  "TopicArn": "arn:aws:sns:us-east-
    1:193603752452:A1EcommerceASG-ScaleOut"
}
{
  "TopicArn": "arn:aws:sns:us-east-
    1:193603752452:A1EcommerceASG-ScaleIn"
}
```

Now, add subscribers to these topics using the following command:

```
aws sns subscribe --topic-arn arn:aws:sns:us-east-
1:193603752452:A1EcommerceASG-ScaleIn --protocol email --
notification-endpoint xxxx@gmail.com
aws sns subscribe --topic-arn arn:aws:sns:us-east-
1:193603752452:A1EcommerceASG-ScaleOut --protocol email --
notification-endpoint xxxx @gmail.com
```

The response from the SNS service is the current confirmation status, as described in the following code:

```
{
  "SubscriptionArn": "pending confirmation"
}
```

The list of topics can be queried using the following command:

```
aws sns list-topics
```

The list of subscribers can be queried using the following command:

```
aws sns list-subscriptions
```

5. The last step is choosing the auto scaling events to be routed to the topics. The command line expects the ARN of the topics and the name of the auto scaling group. Route all events during scaling in to the scale-in topic. Use the following command:

```
aws autoscaling put-notification-configuration --auto-scaling-
group-name A1EcommerceASG --topic-arn arn:aws:sns:us-east-
1:193603752452:A1EcommerceASG-ScaleIn --notification-type
autoscaling:EC2_INSTANCE_LAUNCH
autoscaling:EC2_INSTANCE_LAUNCH_ERROR
autoscaling:EC2_INSTANCE_TERMINATE
autoscaling:EC2_INSTANCE_TERMINATE_ERROR
```

   Route all events during scaling out to the scale-out topic using the following command:

```
aws autoscaling put-notification-configuration --auto-scaling-
group-name A1EcommerceASG --topic-arn arn:aws:sns:us-east-
1:193603752452:A1EcommerceASG-ScaleOut --notification-type
autoscaling:EC2_INSTANCE_LAUNCH
autoscaling:EC2_INSTANCE_LAUNCH_ERROR
autoscaling:EC2_INSTANCE_TERMINATE
autoscaling:EC2_INSTANCE_TERMINATE_ERROR
```

   The notification can be queried using the following command:

```
aws autoscaling describe-notification-configurations
```

Amazon GUI console can be used to verify the resources created via the command line in the preceding steps.

Summary

In this chapter, we reviewed some of the strategies you can follow for achieving scalability for your cloud application. 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 an extended section on setting up auto scaling for our sample application.

In the next chapter, we will shift our focus to strategies to achieve high availability for your application. 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 for our 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 a good practice, 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, ensuring there are no single points of failure. In order to address availability requirements, we will use the AWS infrastructure (Availability Zones and Regions), AWS Foundation Services (EC2 instances, Storage, Security and Access Control, Networking), and the AWS PaaS services (DynamoDB, RDS, CloudFormation, and so on). In addition to availability, we will describe several approaches used for disaster recovery. We will also show you how to implement high availability for our sample application.

In this chapter, you will 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 disaster recovery strategies

Defining availability objectives

Achieving high availability can be costly. Therefore, it is important to ensure that you align your application 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 and services do not need to achieve the highest levels of availability possible; at the same time ensure 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 can turn into a single point of failure for your entire application. Availability is something you should consider early on in your application design process, as it can be hard to retrofit it later. Key among these would be your database and application architecture (for example, RESTful architecture). In addition, it is important to understand that availability objectives can influence and/or impact your design, development, test, and running your system on the cloud.

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

The 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 would include DNS or domain services, load balancers, web and application servers, database servers, application services-related failures, and data center-related failures. You will need to ensure you have a mitigation strategy for each of these types and points of failure. It is highly recommended that you automate and implement detailed audit trails for your recovery strategy, and thoroughly test as many of these processes as possible.

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 such as:

- VPC
- Amazon Route 53
- Elastic Load Balancing, auto-scaling
- Redundancy
- Multi-AZ and multi-region deployments

Setting up VPC for high availability

In this section, we describe a common VPC setup scenario 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 **disaster recovery** (**DR**) site. Leverage AWS's global presence to select the best regions and availability zones to match your 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 in a different country depending on your specific requirements. 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 such as the directory services will usually reside in the private subnets.

Ensure you chose different sets of IP addresses across the different regions for the multi-region 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 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 over 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). It is imperative you consult with your network specialists through all of these tasks.

An ELB is configured in the primary region to route traffic across multiple availability zones; however, you need not necessarily commission the ELB for your secondary site at this time. This will help you avoid costs for the ELB in your DR or secondary site. However, always weigh these costs against the total cost/time required for recovery. It might be worthwhile to just commission the extra ELB and keep it running.

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. Again, consult your network specialists for these tasks.

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 simplest guideline here is to never run a single instance in a production environment. The simplest approach to improving greatly from a single server scenario 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.

ELB uses the least outstanding requests routing algorithm to spread HTTP/HTTPS requests across healthy instances. This algorithm favors instances with the fewest outstanding requests.

Even though it is not recommended to have different instance sizes 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).

There is no bandwidth charge for cross-zone traffic (if you are using an ELB).

Instances that fail can be seamlessly replaced using auto scaling while other instances continue to operate.

Though auto-replacement of instances works really well, storing application state or caching locally on your instances can be hard to detect problems.

Instance failure is detected and the traffic is shifted to healthy instances, which then carries the additional load. Health checks are used to determine the health of the instances and the application. TCP and/or HTTP-based heartbeats can be created for this purpose. It is worthwhile implementing health checks iteratively to arrive at the right set that meets your goals. In addition, you can customize the frequency and the failure thresholds as well. Finally, if all your instances are down, then AWS will return a 503.

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 to run your application stack in more than one zone to achieve high availability. However, be mindful of component level dependencies across zones and cross-zone service calls leading to substantial latencies in your application or application failures during availability zone failures. For sites with very high request loads, a 3-zone configuration might 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 better customer experience.

In the event of a zone failure, there are several challenges in a Multi-AZ configuration, resulting from the rapidly shifting traffic to the other AZs. In such situations, the load balancers need to expire connections quickly and lingering connections to caches must be addressed. In addition, careful configuration is required for smooth failover by ensuring all clusters are appropriately auto scaled, avoiding cross-zone calls in your services, and avoiding mismatched timeouts across your architecture.

ELBs can be used to balance across multiple availability zones. Each load balancer will contain one or more DNS records. The DNS record will contain multiple IP addresses and DNS round-robin can be used to balance traffic between the availability zones. You can expect the DNS records to change over time. 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.

Regional availability or regional redundancy

ELB and Amazon Route 53 have been integrated to support a single application across multiple regions. Route 53 is AWS's highly available and scalable DNS and health checking service. Route 53 supports high availability architectures by health checking load balancer nodes and rerouting traffic to avoid the failed nodes, and by supporting implementation of multi-region architectures. In addition, Route 53 uses **Latency Based Routing** (**LBR**) to route your customers to the endpoint that has the least latency. If multiple primary sites are implemented with appropriate health checks configured, then in cases of failure, traffic shifts away from that site to the next closest 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 the cache fill time (as we might need to default 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. For example, 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.

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

The failover happens entirely within the Amazon Route 53 data plane.

Depending on your availability objectives, there is an additional strategy (using Route 53) that you might 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. Furthermore, you will need to configure Route 53 to point to the S3 storage, where your static site resides. 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 layers of your application architecture.

The auto healing feature of AWS OpsWorks provides a good recovery mechanism from instance failures. All OpsWorks instances have an agent installed. If an agent does not communicate with the service for a short duration, then OpsWorks considers the instance to have failed. If auto healing is enabled at the layer and an instance becomes unhealthy, then OpsWorks first terminates the instance and starts a new one as per the layer configuration.

In the application layer, we can also do cold starts from preconfigured images or a warm start from scaled down instances for your 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 your primary region while the standby servers need not be launched in your secondary region until you actually need them. However, 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 such as Oracle, SQL Server, or MongoDB running on EC2 instances. AWS services come with HA built-in, while using 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 share your databases, which is easy to do. However, resharding your databases while taking in live traffic can be a very complex undertaking and present availability risks. Choosing to use the AWS DynamoDB service in such a situation offloads this work to AWS, thereby resulting in higher availability out of the box.

AWS provides many different data replication options and we will discuss a few of those in the following several paragraphs.

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 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 can be 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 are using SQL Server, you can leverage the SQL Server Always-on feature for synchronous and asynchronous replication across the nodes. If the volume of data is high, then you can also use the SQL Server log shipping to first upload your data to Amazon S3 and then restore into your SQL Server instance on AWS. A similar approach in case of Oracle databases uses OSB Cloud Module and RMAN. 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 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 will 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 the AWS cloud.

Designing your application services to be stateless and following a micro services-oriented architecture approach 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 then spread information across multiple AZs for high availability. Another design principle is to rigorously implement exception handling in your application code, and in 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; store shared data; stream real-time data; and so on. There are a host of highly available AWS services that can be used for implementing such mobile application functionality. For example, you can use Amazon Cognito to authenticate users, Amazon Mobile Analytics for analyzing user behavior and tracking retention, Amazon SNS for push notifications and Amazon Kinesis for streaming real-time data. In addition, other AWS services such as S3, DynamoDB, IAM, and so on can also be effectively used to complete most mobile application scenarios.

For mobile applications, you need to be especially sensitive about latency issues; hence, it is important to leverage AWS regions to get as close to your customers as possible.

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 of 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. Such packaged applications can be sourced from AWS Marketplace. 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 for achieving high availability on the 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 might 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 such as 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** (**RPO**).

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

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 the Direct Connect facility to set up a high throughput and low latency connection between your data center and the AWS cloud for your backups. If 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 compared to 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 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 configuration on AWS only or a hybrid of the on-premise and AWS cloud infrastructure. In cases of a 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 during disasters (at full 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 a result of any failures.

Testing a disaster recovery strategy

It is imperative to thoroughly test your recovery strategy end to end, to ensure 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 of 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. 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. You can, however, 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.

Setting up high availability

This section introduces configuring AWS infrastructure to support high availability for our application. Most of Amazon's high-level services are designed for high availability and fault tolerance such as Elastic Load Balancer (ELB), Simple Storage Service (S3), Simple Queue Service (SQS), Simple Notification Service (SNS), Relation 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) provide constructs such as availability zones, elastic IP addresses, and snapshots to design high availability and fault tolerant applications. Remember hosting an application on the cloud does not make it fault-tolerant or highly available.

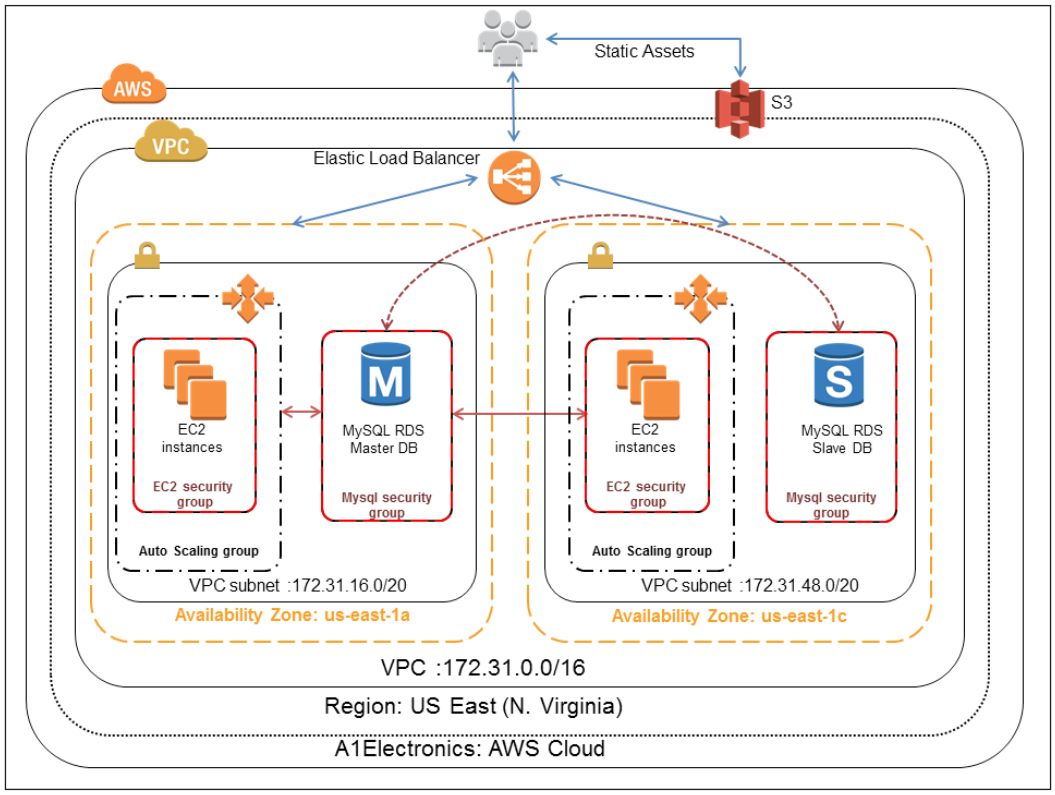
We will be architecting for high availability and not for fault tolerance. The difference between them is that there is no service interruption in the case of fault-tolerant services; whereas, there is minimal service interruption in cases of high availability.

The AWS high availability architecture

Let's start by designing generic high availability architecture for an Amazon region. High availability can be architected in many different ways depending upon the services used and the requirements. Here we present a very generic architecture .The key for achieving high availability is to avoid **single point of failure** (**SPOF**) that is, the application will fail when any one of the components or services that make up the system fails. This implies that we have to cover for all the Amazon services which can fail in a region which the application uses, in our case, these are as follows:

- Availability zone (AZ)
- EC2 instances (EC2)
- Elastic Load Balancer (ELB)
- Relation Database Service (RDS)

In the preceding list of Amazon services, ELB and RDS are already designed for high availability and need to be configured to support high availability as per the architecture. Let's begin by analyzing the high availability architecture. Let's take a look at the following figure:



The application is hosted in the US East region of the AWS cloud. This architecture is designed to handle failures within a region and not across regions.

- **Availability Zone**: An availability zone is equivalent to a datacenter. 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 failures. The decision on how many availability zones to host the application zone depends on how critical the application is and the economics of hosting. This removes the SPOF if using a single AZ.

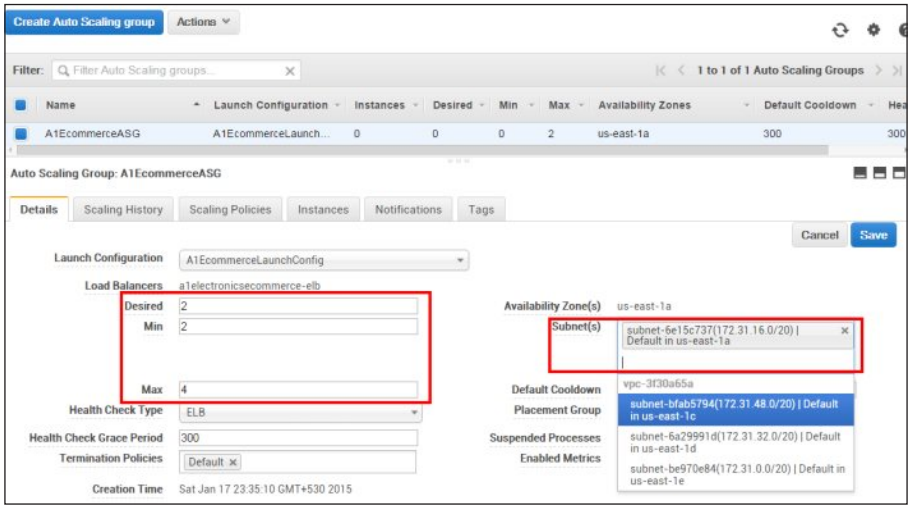
- **Elastic Load Balancer**: All the traffic is routed via the ELB to the applications. This piece of infrastructure is fault-tolerant by design. ELB needs to be configured for routing traffic to the application hosted in different AZ's. In addition, ELB performs health checks on all the EC2 instances registered with it and only routes traffic to the healthy EC2 instances. This removes the SPOF at the load balancer tier.
- **EC2 instances**: An EC2 instance is the most vulnerable component in the chain. It is the most complex in terms of the software components it has, which is your application plus the supporting software stack. Failure in either will make your application unavailable. To cover for this, an **auto scaling group** (**ASG**) is used that monitors and launches the EC2 based on the configuration of alarms, maximum, minimum, and desired EC2 instances per availability zone. This removes 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 availability zones, one of them being a master and the other being a standby replica or slave instance. The master and standby database 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; under the hood, 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 salve RDS instance. Multi-AZ deployment is designed for 99.95 percent availability. In addition, RDS can be configured to take a snapshot of the database at regular intervals, which helps during disaster recovery. This removes the SPOF at the database tier.
- **Simple Storage Service (S3)**: S3 is a highly available service for storing static assets. Amazon S3 is designed for 99.99 percent availability and 99.999999999 percent of durability of objects over a year. All the files uploaded to the application should be stored in S3. Even if the EC2 instances fails, the uploaded file is not lost and another EC2 instance can process if required. It is good practice to store all the static assets such as images/scripts of the application into S3 as it takes the load off your EC2 instances.
- **Virtual Private Cloud (VPC)**: Amazon automatically creates a VPC for a region for all the accounts created after March 18, 2013. By default, subnets are created for all availability zones in a region. You can create up to 200 subnets per region, which can be increased on request.

For a high availability setup, the existing AWS services that were provisioned and configured under *Chapter 3*, *AWS Components, Cost Model, and Application Development Environments* and *Chapter 4*, *Designing for and Implementing Scalability* need to reconfigured 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 the availability zones. We can follow the steps listed here to launch the EC2 application:

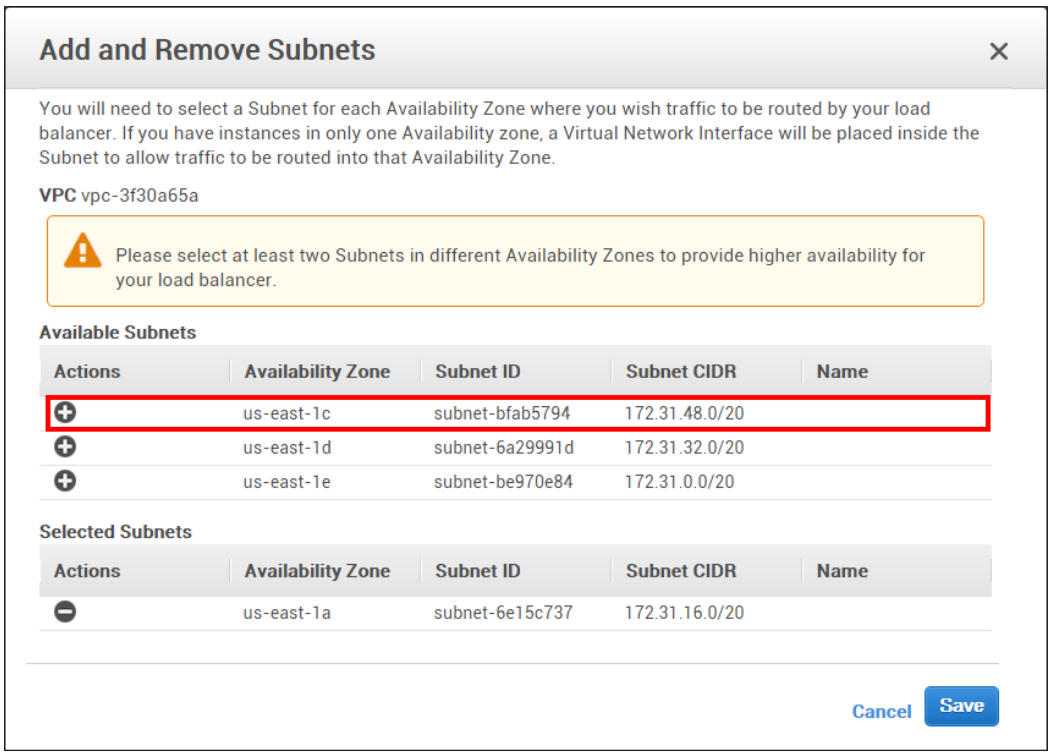
1. From the EC2 dashboard navigation pane, click on **Auto Scaling Groups**. Select the **A1EcommerceASG** auto scaling group and then click on the **Edit** button in the **Details** tab.
2. From our architecture diagram, the second availability zone selected is **us-east-1c** where the EC2 instances will be launched. Select the subnet associated with the **us-east-1c** availability zone from the dropdown in the Subnet(s). This configures the ASG for high availability.
3. For high availability, the minimum number of instances running in the auto scaling group needs to be two, one EC2 instance per availability zone. Modify the **Desired** and **Min** number of instances to `2` and **Max** to a number greater or equal to `4`.
4. Save the changes. As soon as you save the changes, the auto scaling group will start a new EC2 instance in the **us-east-1c** availability zone. Let's take a look at the following screenshot:



HA support for ELB

The next step is configuring the ELB to route the traffic to EC2 instances in the added availability zone that is **us-east-1c**. Perform the following steps:

1. From the EC2 dashboard navigation pane, click on **Load Balancers**. Select the **a1electronicsecommerce-elb** load balancer and then click on **Edit Availability Zones** in the **Instances** tab.
2. Click on the **us-east-1c** availability zone to add it to the ELB. Click on **Save**. The ELB starts routing the traffic to the EC2 instances in the us-east-1c zone.

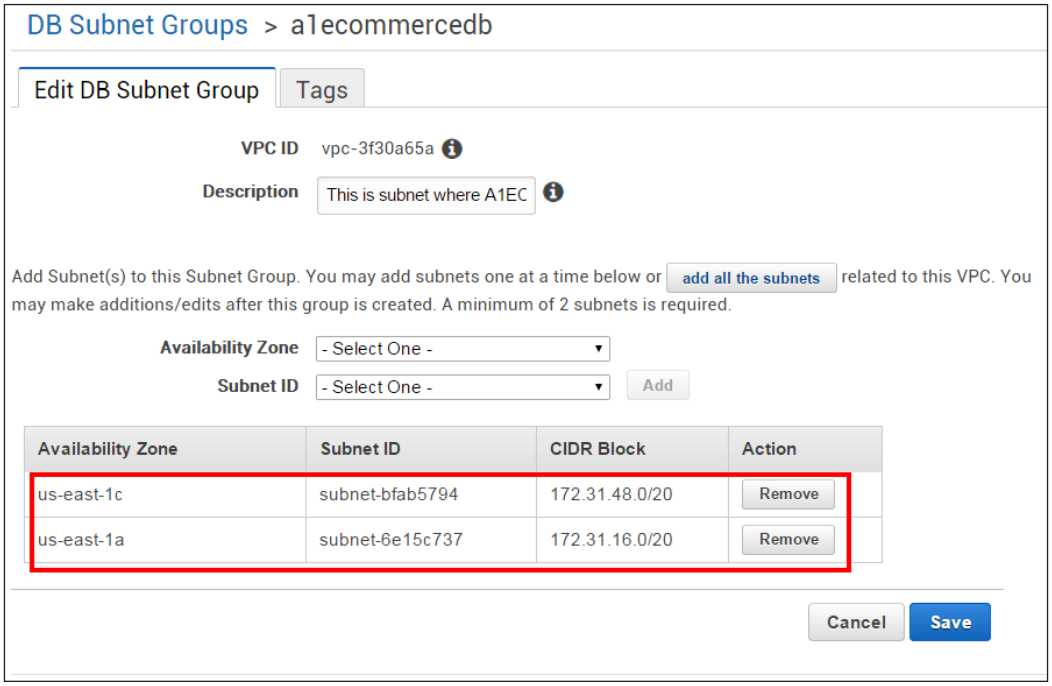


HA support for RDS

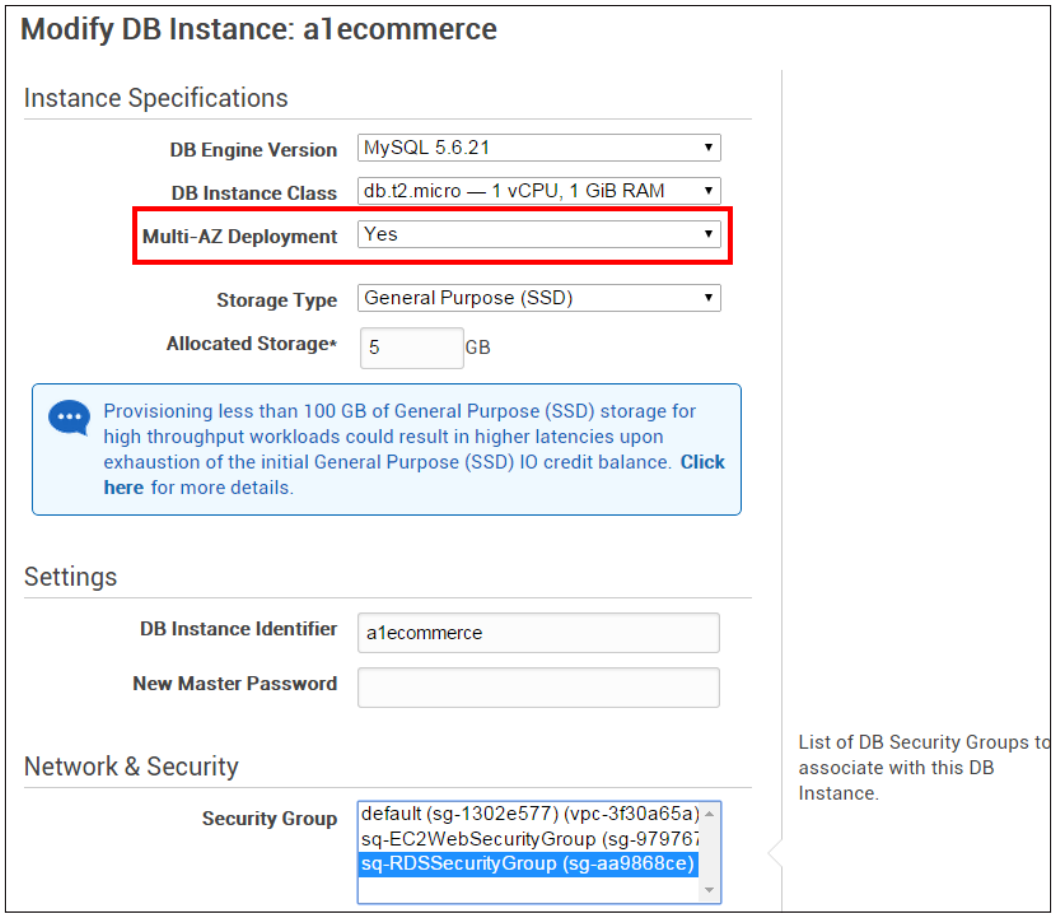
As explained RDS provides high availability via Multi-AZ. To setup Multi-AZ, RDS needs to be configured for two things—one, the availability zone where the slave RDS instance will be launched and instantiated and the second is the Multi-AZ support for the current RDS instance.

Perform the following steps:

1. From the RDS dashboard navigation pane, click on **Subnet Groups**. Select **a1ecommercedb** and click on **Edit**. Make sure there are a minimum of two availability zones in the subnet group. The two availability zones as per the architecture diagram are **us-east-1a** and **us-east-1c**.

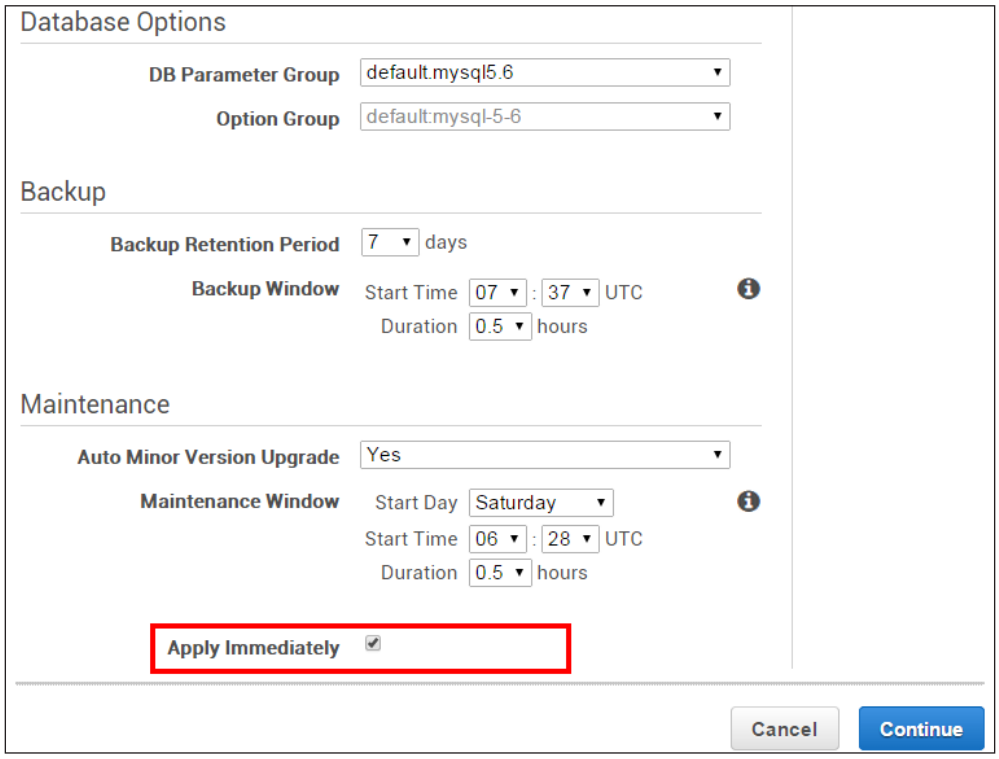


2. From the RDS dashboard navigation pane, click on **Instances**. Select the **a1ecommerce** database instance and click on **Modify** from the **Instance Action** dropdown. Select **Yes** from the **Multi-AZ Deployment** dropdown, this will instantiate the slave database instance in the us-east-1c availability zone.



Make sure you check the **Apply Immediately** checkbox; if not, your changes will be scheduled for the next maintenance cycle.

Let's take a look at the following screenshot:



Summary

In this chapter, we reviewed some of the strategies you can follow for achieving high availability in your cloud application. We emphasized the importance of both designing your application architecture for availability and using the 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 for designing and implementing security for your cloud application. We will review some approaches for application security using the 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 startup, 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 the AWS services including IAM, CloudTrail, and CloudWatch. We will also show you how to implement security for our sample application.

In this chapter, we will cover the following topics:

- 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, key management service, and configuring SSL
- Securing data-at-rest – Amazon S3 and RDS

Defining security objectives

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 you organization. While larger companies may have well-defined security controls already defined for their on-premise 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 produce security-related documentation to meet varying demands 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 the bases and measure success against a set of well-defined metrics. Mapping your implementation against the control framework allows you to produce evidence of due diligence to your customers. In addition, you should budget for the expenses and effort required for conducting regular audits. In some cases, be prepared to share these audit reports with your major customers.

Implementation costs can vary widely based on security mechanisms; 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 requirements and risks.

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 security responsibilities

AWS security operates on a shared responsibility model comprising of parts managed by you and other parts managed by AWS. For example, you will need to implement your own security controls for users and roles, policies and configuration, applications and data (storage, in-transit, and at-rest) and for firewalls, network configuration, and the operating system.

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).

AWS has a whole host of industry recognized compliance certifications and standards such as **Payment Card Industry** (**PCI**), NIST, SSAE, ISO, and so on. Hence, we will keep our focus on your responsibilities in this chapter.

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 (into high/medium/low categories) the assets you need to protect. This is often a non-trivial undertaking in large enterprises. Assets related 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 Describe 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 or network ACLs, or the ability to turn on CloudTrail on all your AWS accounts. In addition, we typically implement infrastructure as code on the AWS cloud. Security is now baked into the whole deployment process. For example, when code is deployed on a new EC2 instance, the OS hardening happens as part of the build pipeline.

The AWS IAM service is central to implementing security for your applications on the AWS cloud. Some of the main activities and best practices for AWS IAM are listed as follows:

- Use IAM to create users, groups, and roles and assign permissions.
- Manage permissions using groups. You assign permissions to groups and then assign individuals to them. While assigning permissions to groups, always follow the principle of granting 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 groups that have 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 might be a combination of people within your organization, 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 access, temporarily or otherwise. Restrict privileged access further using the 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 the 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 more 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. Still other best practices relate to using 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`.

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

Implementing identity lifecycle management

Establishing a robust identity lifecycle management is often considered late in the development lifecycle by organizations offering SaaS applications on a global basis. For example, how do you keep track of 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 the easiest, from an account management perspective, to have a feature within your application to create an application administrator role per customer who in turn is responsible for managing users within their organization.

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-premise directory services to the AWS cloud using Direct Connect.

Tracking the AWS API activity using CloudTrail

AWS CloudTrail is a web service for recording the 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 to AWS resources, 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 a more comprehensive threat analysis and mitigation. However, ensure your logging activity is scalable and does not unduly impact the performance of your application. For example, you can use SQS with auto scaling based on the queue depth for the logging activity. In addition, you can also use products such as 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 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 like HP Fortify (available as a SaaS service or an on-premise product) or Veracode (SaaS service) can be used to identify vulnerabilities within your application code. These enterprise security tools might be expensive but they are great for prevention of OWASP top ten type vulnerabilities in your application and promoting secure coding practices in your development teams.

It is important to schedule a penetration test with specialists both within your organization and 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 that you allow sufficient time in your project schedule for 2-3 rounds of testing and remediation work.

Reviewing and auditing security configuration

It is important 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 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 on-going 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 might get defined, permissions might 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 describe the features and walk you through the process of setting up security for our sample application. This will include using IAM roles, 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 such as facilities, hardware, network and some software like 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.

AWS IAM – Securing your Infrastructure

AWS Identity and Access Management (IAM) is a web service that enables you to manage users, groups, and user permissions within the AWS infrastructure. This allows for central control of users, groups, user access, and security credentials. As there are a plethora of services being offered by AWS, there is a need of securely accessing these services by authorized users. IAM defines concepts, constructs, and services to achieve this. IAM solves the following issues:

- **Credential Scoping**: This 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 the entire S3.
- **Credential Distribution**: This facilitates the distribution and rotation of credentials to users, AWS services, instances, and to applications in a secure manner.
- **Managing 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 totality is beyond the scope of this book and probably would need a book on its own. In this section, only the pertinent IAM concepts and services are discussed, which cover a general web hosting use case.

IAM roles

A role is a set of permissions that grant 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 services such as EC2 assume a role; there is no need to provide credentials to an EC2 instance. This solves a very important issue that is of 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 Model, 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 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 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 either on the EC2 or hardcoded it into the application. The instance metadata can be queried from the running EC2 instance from the command line by making a call to:

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

This returns the following metadata that can be queried:

```
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
```

To query the temporary security credentials for a role execute the following command from the running EC2 instance command line:

```
curl http://169.254.169.254/latest/meta-data/iam/security-
  credentials/ec2Instaces
```

Note that `ec2Instaces` is the name of the role assigned to your EC2 instance. The response will be a temporary security credential that the AWS SDK uses to access the resource:

```
{
  "Code" : "Success",
  "LastUpdated" : "2014-11-12T03:15:30Z",
  "Type" : "AWS-HMAC",
  "AccessKeyId" : "ASIAICXSVRXJIG56YDRA",
  "SecretAccessKey" : "bH6Q+s6bLB9CRMrQBRrqDVx7aCm+fdDRfITWBevO",
  "Token" : "AQoDYXdzEFQa0ANZNtrSNX5uqJa0jYEmK/93OEKYJmjgHm+qqWYKIT
CBL//p+6l7De7PzJrYcmJRM2uLuOJ32ejjrA02bFRhP21MvVkU6xOM960Se1VMtmXUF
p+3seSke/7bBblbtEOtbOWKcAo3vEXCCVuF61vQruH760Ak5kTwowaQokRwHlWG+71+
Hn4dd1OqqBvHQw/ISI+MSXYDmcjgX1xPnZ6TNaW8UfeKtEac8Msr3ZfSYDkohxFZrKC/
JB30tDRRN1WG093sPU8wSZ3ljYJW37xH8L2q5i9tdwrxOz28Kr6O5qU8jPGZPJSeUqGJkB/
L0wK6G92drLQoE4kylybNeF9R2X7aIYw4vsuputEuptAOmWNH0W43LIScZEE0r9es8BbF6
mQpel+epT/Y0VpphrG9TSJSQ4U64vFjCK+dQhTiktrBcx0Fvwa+yGopanOHAa9MpNWT5BwS3
vULxAgAhamXtsgds+U1rMu4M8C2+fAZCyYE9VxpQNgBPSecynLOM3oa4ar+8BGHZztEB24x
aqFrp36C/4V5IcIJyOgRhfohmAAYslOlCWN3Tl6mJFA2I5DHFb2nRYmAaq7whL5gjMXeW1+
PWgzE6GMb5C/xeD8nBKEFsyDAiMqnBQ==",
  "Expiration" : "2014-11-12T09:50:26Z"
}
```

The temporary credentials are automatically rotated and have an expiry date and time associated with it. The application has to query the instance metadata for the new credentials before the current credential expires; if AWS SDK is being used within an application, then it manages it transparently and no credential key refresh logic is necessary. You are not charged for making the instance metadata calls. For more information on the metadata, please refer to `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-metadata.html`.

AWS Key Management Service

We all have used encrypted data in some application or the other, and the biggest challenge has always been knowing how to effectively hide the encryption key, the key with which the data is encrypted within the application or the OS by using different mechanisms. However, at 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 for a single application. Imagine you have tens of applications running on the cloud, the challenge of the key distribution and the effort to keep the key secret multiplies exponentially because a programmer interacts with many others. In addition, social engineering could be used to get the master key, or an unauthorized user compromises your network since the master key is in clear text. With KMS, the master key is never released but still enables you to encrypt and decrypt data.

AWS key management service manages the following issues:

- **Encryption for all your applications**: This manages encryption keys used to encrypt data stored by your applications, regardless of where you store it. KMS provides an SDK for programmatic integration of encryption and key management.
- **Centralized Key Management**: This provides centralized control of your encryption keys, presents a single view into all of the key usage, allows for creation of keys, implements key rotation, creates usage policies, and enables logging.
- **Integration with AWS services**: This is integrated with other AWS services such as S3, Redshift, EBS, and RDS to make it easy to encrypt data you store with these services.
- **Built-in Auditing**: This logs all API calls from KMS to AWS CloudTrail. It 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**: This is a fully managed service. AWS handles availability, physical security, and hardware maintenance of the underlying infrastructure.
- **Low Cost**: Each key costs $1/month plus you pay only $0.03 for 10,000 usage requests.

Let's get started with the process of creating a master key, and then we will use it to encrypt and decrypt data. As a good security practice, create and use different keys for different AWS services such as S3, RDS and within your applications.

Creating the KMS key

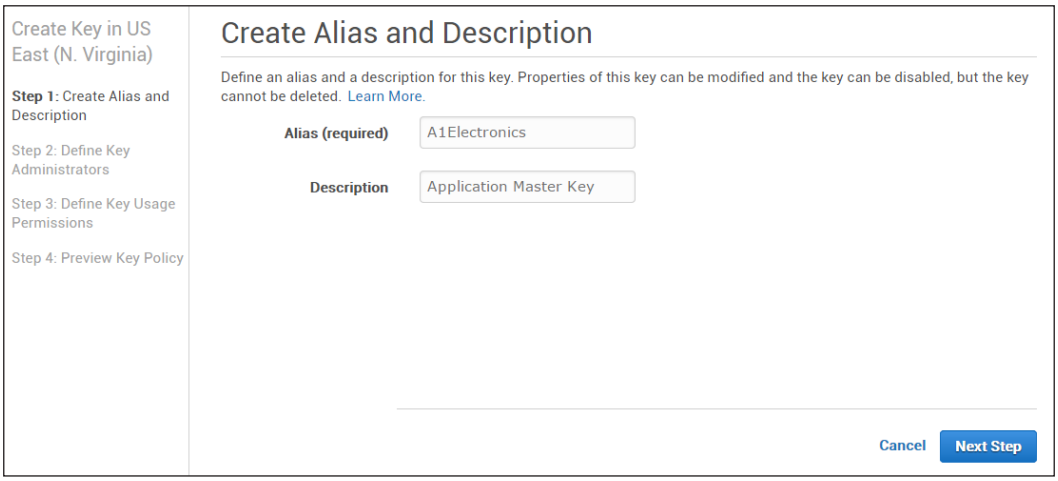
From the IAM dashboard navigation pane, click on **Encryption Keys** and then on **Create Key** to create a new master encryption key.

Once created, a key cannot be deleted; it can only be enabled or disabled.

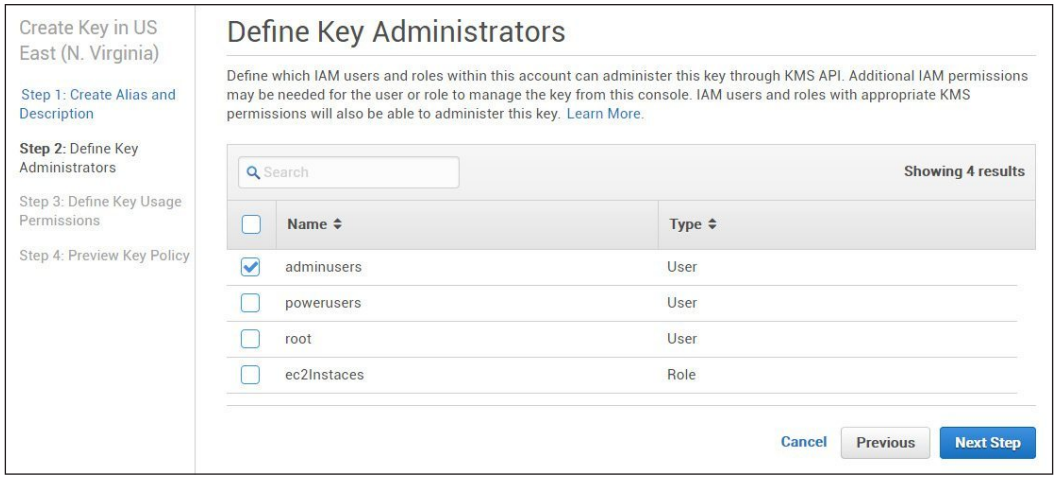
After creating a key, perform the following steps:

1. The first step is to create a key for a region. In this step, we will create an alias and describe it as follows:
    - **Alias**: 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 these are reserved by AWS 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.

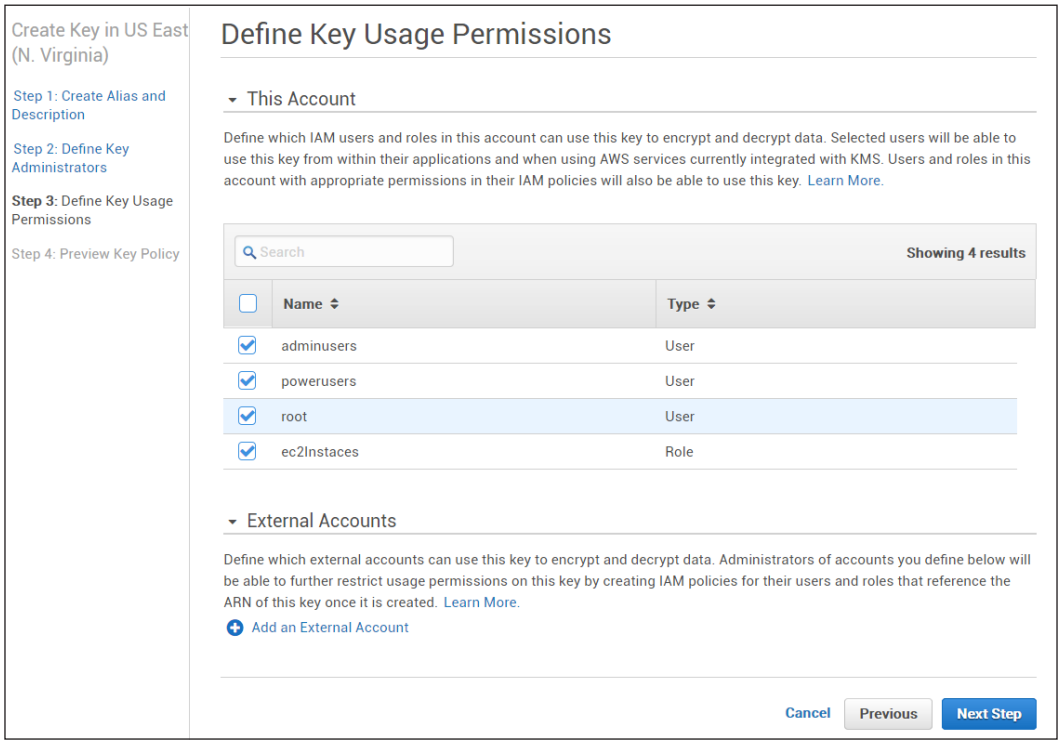
2. Click on **Next Step**, which will configure the users who administer the key.



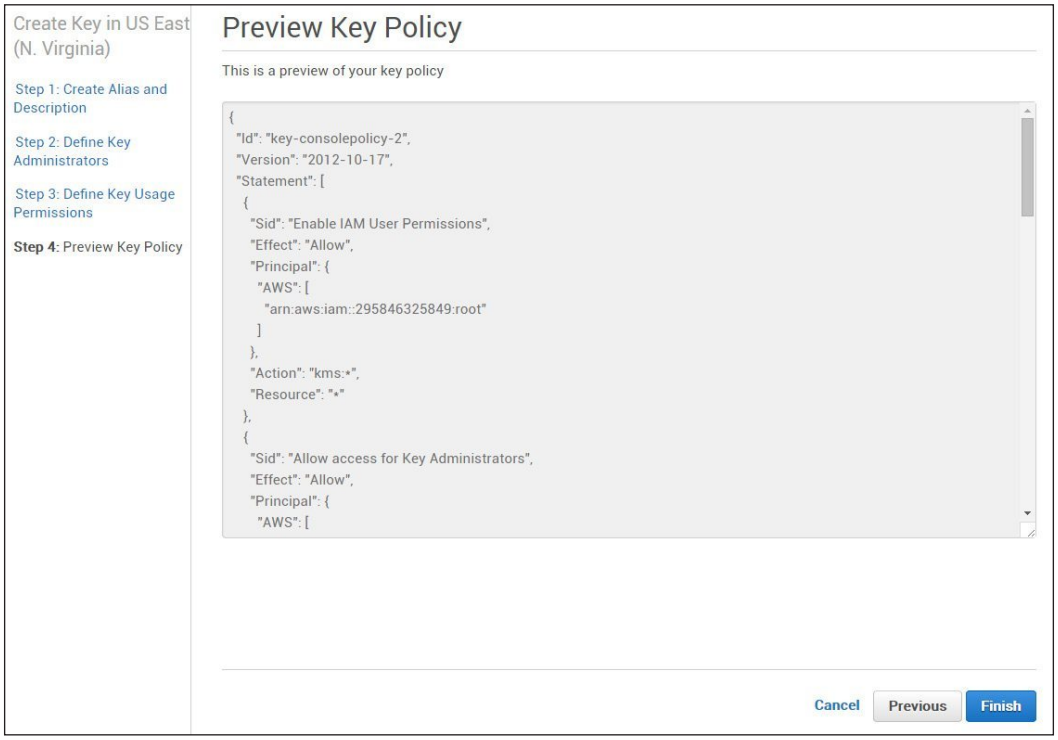
3. The next step is to associate the users/roles who will have administration rights on this key. The administration rights allows enabling or disabling of a key, rotation of keys, and adding of users/roles that can use the key. In our example, **adminusers** is selected as the IAM group. Click on **Next Step**; this assigns users to the key, as shown in the following screenshot:



4. The next step is to assign rights to the IAM users/roles. Usage rights in this context means to encrypt and decrypt data using this key. It is a good practice to assign rights to roles instead of users, as it helps to centralize user management around roles. Click on **Next Step** to review the key policy. Let's take a look at the following screenshot:



5. You can now review the policy before creating one. Click on **Finish** to create a new master key. Note that the result of this wizard is a JSON policy file. This JSON file can also be externally created and edited and should be stored in your SCM tool. It is a good practice to test the policy on the IAM policy simulator at `https://policysim.aws.amazon.com/home/index.jsp?#`. Let's take a look at the following screenshot:



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 an encrypted format. Here is a class 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 `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. As a good security practice, remove the credentials if you are running it within the EC2 instance; the AWS sdk will query and use the credentials from the instance metadata.

Let's take a look at the following code:

```
public class KMSClient {
  private String keyId = "arn:aws:kms:us-east-
    1:993603752452:key/2b0c514c-0ea9-48cc-8c70-a8e60c4724be";
  private AWSCredentials credentials;
  private AWSKMSClient kms;

    public KMSClient() {
      credentials =  new BasicAWSCredentials("AKIAJYLXXXXX",
        "FCCCCSS2wFq+w5bHrKUsYfNHUW/KFm8rJMYi7kmm");
      kms = new AWSKMSClient(credentials);
      kms.setEndpoint("kms.us-east-1.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

In application security settings, we look at:

- 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, such as passwords and keys used by the application. We have already covered this previously in the *Using the KMS key* section.

Transport security

While transporting data over HTTP, security is provided by an SSL. SSL is widely used on the Internet to authenticate a service to a client, and then to provide encryption to the transport channel. Configuring the ELB to accept SSL certificates will secure the transport channel between the user's browsers 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, whose purpose is to sign certificates for network entities it has authenticated using secure means. Normally, you would create a CSR and have the CSR signed by the CA. We will not use a commercial CA to sign a certificate but instead use a self-signed certificate. As a result, 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, a secure transport channel is created between the browser and the ELB. If you own a domain then you can access `www.startssl.com` to obtain free SSL certificates.

Generating self-signed certificates

OpenSSL is used to create the keys and certificates and to make sure you have it installed on your development machine. The example shown here is for a Linux machine. From the command line, execute the following command:

```
openssl req -x509 -newkey rsa:2048 -keyout key.pem -out cert.pem
  -days 3650 –nodes
```

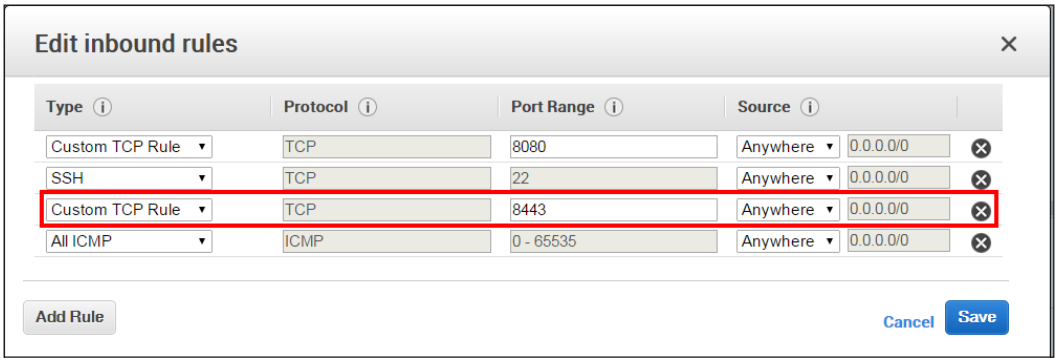
This will create a 2048 bit RSA private key `key.pem` and this private key is used to sign the certificate `cert.pem` file. While generating the signed certificate, make sure you enter the correct information for `Common Name` (for example, server FQDN or your name); here, we have used the ELB public DNS name:

```
Generating a 2048 bit RSA private key
.+++
..+++
writing new private key to 'key.pem'
-----
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-721149061.us-east-1.elb.amazonaws.com
Email Address []:admin@a1lectronics.com
```

Configure ELB for SSL

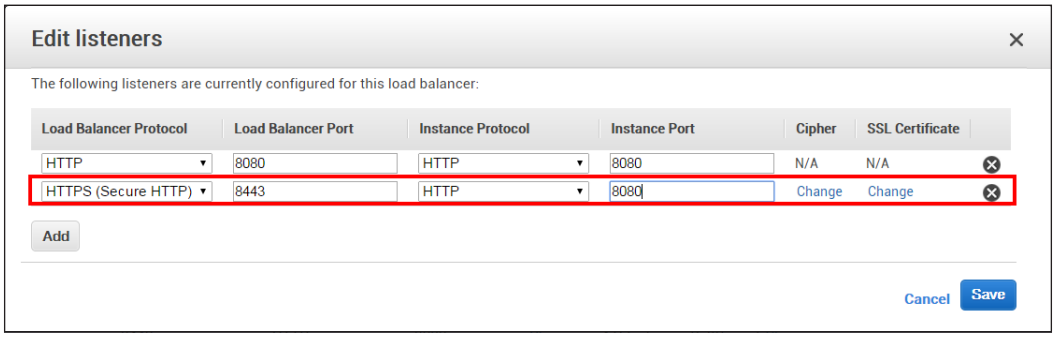
The next step is to configure the ELB to support SSL. Here, the SSL connection will be terminated at the load balancer. The connection between the ELB and your EC2 instance will be unsecured. The standard HTTPS port is 443; instead, we use port 8443, as using port 443 requires configuring the OS and the default Apache Tomcat configuration files. Perform the following steps:

1. The first step is to configure the security group to add a custom TCP rule to accept data on port 8443. From the EC2 dashboard, navigate to **Load Balancers**, click on the **Security** tab, and then click on **Security Group ID** associated with the ELB. In our example, this is **sq-EC2WebSecurityGroup**. The click action will navigate to the **Security Groups** pane. Click on **Edit** in the **Inbound** tab to add the TCP rule and accept data on port 8443. Delete **Custom TCP Rule** on **Port Range 8080** as it is being replaced by the **8443** port, as shown in the following screenshot:



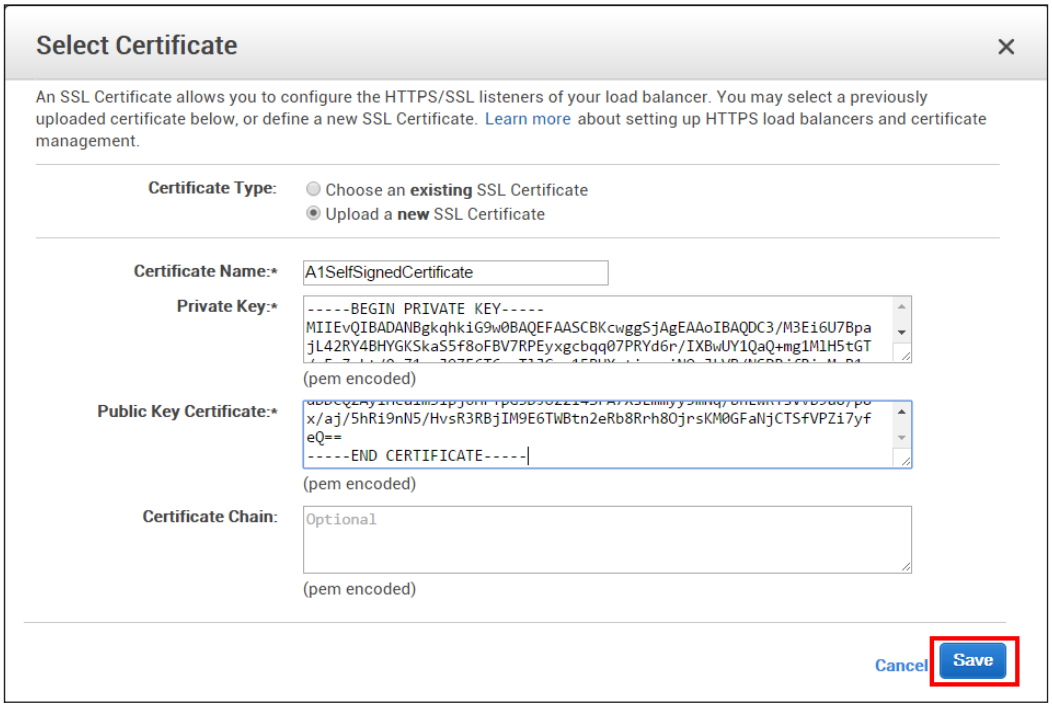
2. The next step is to add load balancer protocol (HTTPS) and listener port and configure the private and the public key on the ELB. From the EC2 dashboard, navigate to **Load Balancers**, click on the **Listeners** tab, and then click on **Edit**:
    - From **Load Balancer Protocol**, select the **HTTPS** protocol
    - Set **Load Balancer Port** to `8443`; this is the port we added to our security group in our previous step

- From **Instance Protocol**, select **HTTP**; this is the protocol between the ELB and the EC2 instances
- Set **Instance Port** to `8080`; this is the port that the Tomcat is listening on
- From **Load Balancer Protocol**, delete the **HTTP** protocol as it is not needed anymore



3. The next step is to associate the SSL certificate with the ELB. Click on **Change** under **SSL Certificate**. The following are the properties:
   - **Certificate Type**: Make sure the radio button **Upload a new SSL Certificate** is selected.
   - **Certificate Name**: Enter the name of the certificate for your reference; this will be reflected in the ELB dashboard **Listeners** tab.
   - **Private Key**: Copy the contents of `key.pem` and paste it in the edit box.
   - **Public Key Certificate**: Copy the contents of the `cert.pem` file and paste it in the edit box.

- Click on the **Save** button. This will configure ELB to support the SSL protocol. Test the URL on the browser using the HTTPS protocol. Let's take a look at the following screenshot:



Secure 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, Dynamo DB and so on. To secure data-at-rest, symmetric encryption is used, that is, the data is encrypted with an encryption key—the data is secured as long as the encryption key is secure so all the effort is directed to keep this encryption key secure. AWS provides **Key Management Service** (**KMS**) to resolve the issues related with management and storage of encryption key as described in the previous section. This service is also used to secure the data-at-rest. Encryption of data at rest is a key component of regulations such as HIPPA, PCI DSS,SOC1, 2, 3, and so on. In this section, walkthroughs to secure the data-at-rest for RDS and S3 are presented.

Secure data on S3

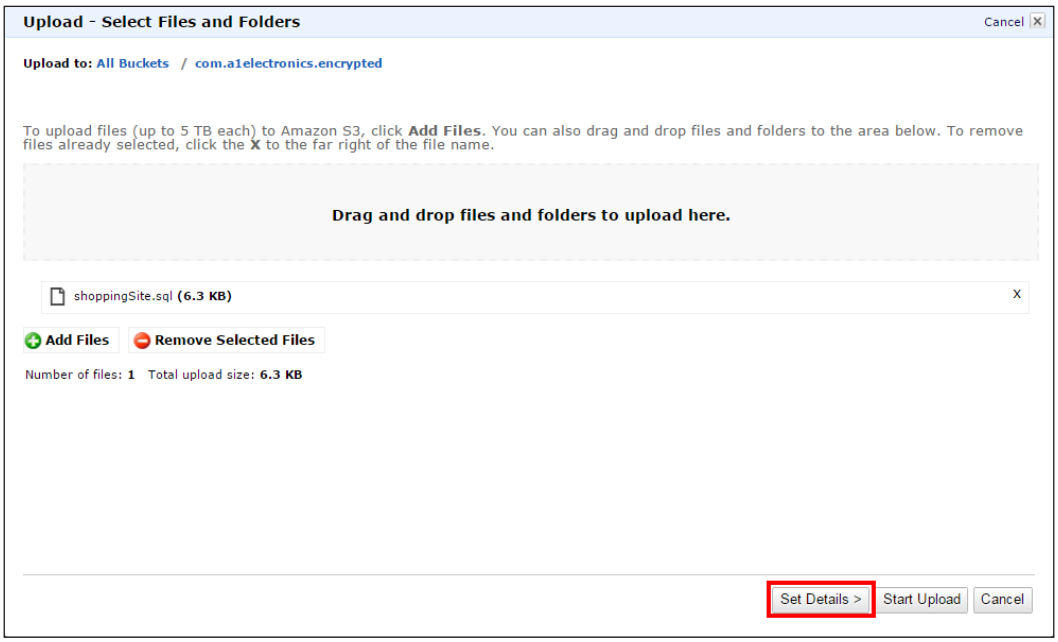
To secure the data-at-rest within S3 broadly, there are two options, as follows:

- **Server-Side Encryption**: Amazon S3 encrypts your object before saving, and decrypt's it when you download 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 the encryption of the object before uploading it to Amazon S3, and for decrypting the object after it has been downloaded. The client is responsible for the encryption/decryption process and management of encryption keys.

Here, we show you two ways by which you can achieve server-side encryption—one, by using the S3 console and the other by uploading a file to S3 via Java AWS SDK. We will not go through the client-side encryption.

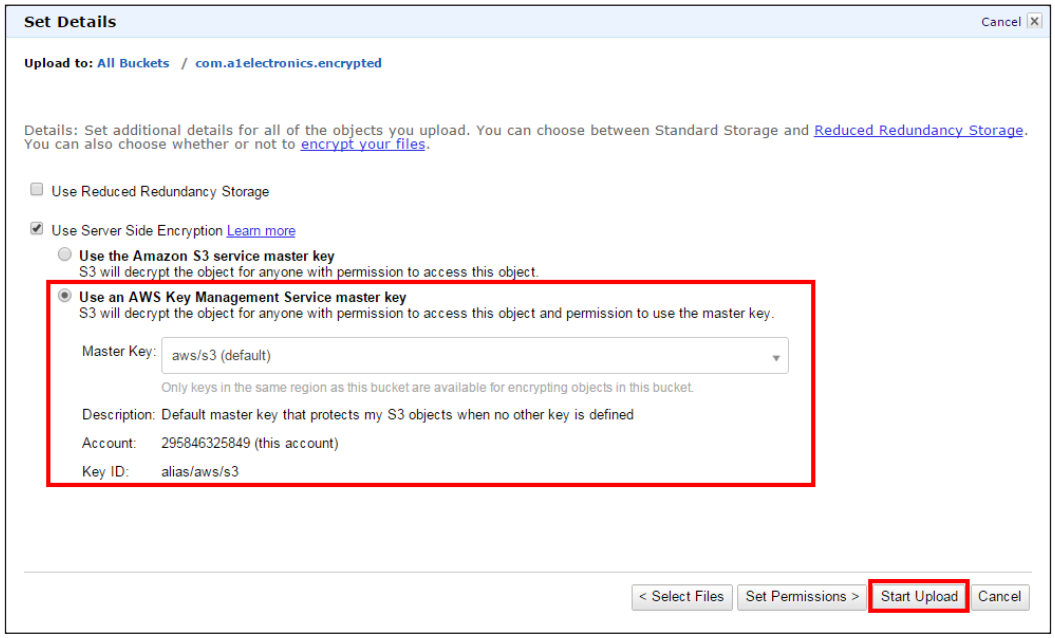
Using the S3 console for server-side encryption

The easiest way to secure data on S3 is via the S3 console. Select the bucket where the file is to be uploaded, click on the **Upload** button; this presents a pop up window to upload files, as shown in the following screenshot:



Click on **Set Details** to set the encryption options.

Select the check box **Use Server Side Encryption** and then click on **Use an AWS Key Management Service master key** to select the key to encrypt the file with. There is an option of selecting **Master Key** for encryption available with the Key Management Service. Click on **Start Upload** to upload the selected file, as shown in the following screenshot:



Using Java SDK for server-side encryption

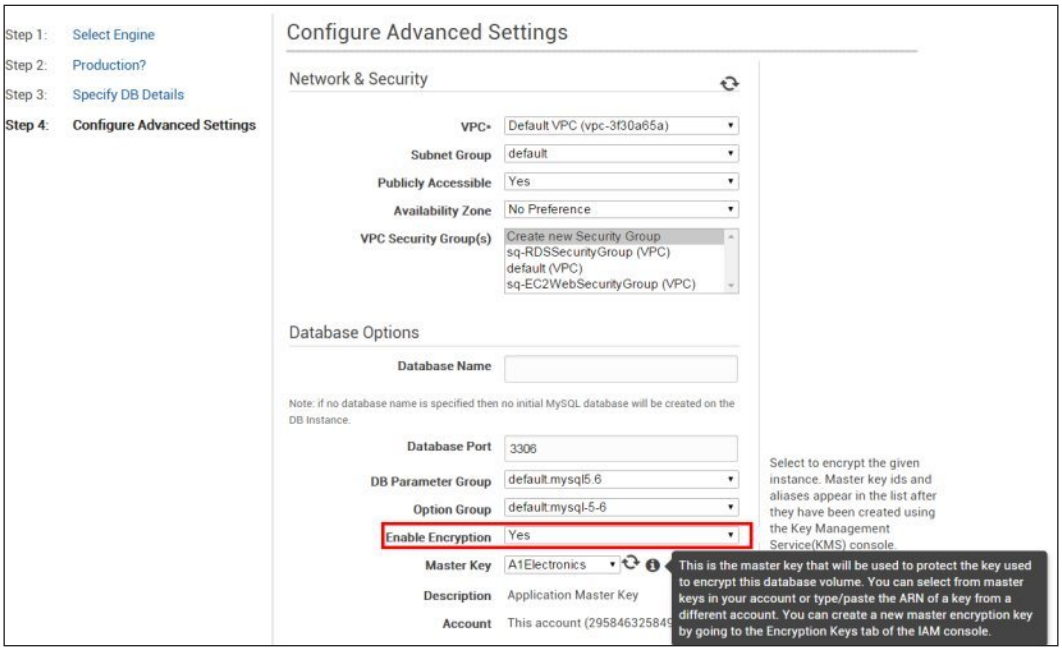
Here is a code snippet that will upload code from your application and instruct S3 to encrypt the file; this is not the same as a client encrypting the file and uploading it to S3, since the code only passes the location of the encryption key within the KMS and the actual process of encryption is handled by S3. The variable `keyId` needs to be initialized with the KMS key identifier with which the S3 will encrypt the file. The key identifier of master keys is located within the KMS dashboard:

```
public class S3 {
  /** The s3 secret key. */
    private String secretKey;
  /** The s3 access key. */
    private String accessKey;
  /** The s3 amazon s3 client. */
    AmazonS3 amazonS3Client;
  /** The s3 client options. */
```

```
  S3ClientOptions s3ClientOptions;
/** KMS encryption key id **/
    private String keyId;

  public S3(String secretKey,String accessKey) {

    this.secretKey=secretKey;
    this.accessKey=accessKey;
    this.endPoint="s3-external-1.amazonaws.com"; // US Standard
      * N. Virginia only
    this.keyId="2b0c514c-0ea9-48cc-8c70-a8e60c4724be "; // KMS
      Encryption KeyID for S3

    amazonS3Client = new AmazonS3Client(new
      BasicAWSCredentials(this.accessKey,this.secretKey));
    s3ClientOptions = new S3ClientOptions();
    s3ClientOptions.setPathStyleAccess(true);
    amazonS3Client.setS3ClientOptions(s3ClientOptions);
    amazonS3Client.setEndpoint(this.endPoint);
    this.accessKey=null;this.secretKey=null;
  }


  /**
   * To upload a file to a particular s3 bucket with a key.
   *
   * @param file file to upload.
   * @param object for this file.
   * @param s3Bucket the s3 bucket name.
   */
  public void uploadFileEncrypted(File file, String object,
    String s3Bucket) throws AmazonServiceException,
      AmazonClientException, InterruptedException {
    TransferManager s3TransferManager = new
      TransferManager(amazonS3Client);
    PutObjectRequest putObjectRequest = new
      PutObjectRequest(s3Bucket, object, file).
        withSSEAwsKeyManagementParams(new
          SSEAwsKeyManagementParams(this.keyId));
    putObjectRequest.setStorageClass("REDUCED_REDUNDANCY");
    Upload upload = s3TransferManager.upload(putObjectRequest);
    upload.waitForCompletion();
    s3TransferManager.shutdownNow(false);
    s3TransferManager=null;
  }

}
```

Secure data on RDS

The RDS service secures the database by encrypting the database volume with the specified encryption key from the KMS. Note, 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 doesn't need the encryption key to decrypt the data. If an unauthorized user gets a hold of the database volume, it will be of no real value to him, since it is encrypted, and without the encryption key, it cannot be decrypted. The option to encrypt the database volume is available while creating the database under **Configure Advanced Settings**. This encrypts the RDS read replicas, automated backups, the underlying database storage, and database snapshots. Let's take a look at the following screenshot:



Summary

In this chapter, we reviewed some of the strategies you can follow for achieving security in your cloud application. We emphasized on the best practices of implementing security using the 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 backups and restores and application monitoring and troubleshooting. We will also include a hands-on section that will walk you through these processes for our sample application.

7
Deploying to Production and Going Live

In this chapter, we will focus on getting your application live in the cloud. As an enterprise, or a start-up, you want to ensure that your applications are deployed and supported appropriately to best serve your customers. We will discuss tools, approaches, and the best practices in deployment and operations that ensure smooth functioning of your applications in production environments. We will also show you how to deploy a sample application in a production environment.

In this chapter, we will cover the following topics:

- 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 has 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 the 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 stack, 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 OpsWorks, AWS CodeDeploy, AWS CloudTrail, and AWS CloudWatch. We describe some of these in greater detail in the following sections.

Besides AWS tools and services, it is also important that we upgrade our skills and try to stay as current as possible with 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, the infrastructure engineers, specialists, and administrators focus more on organizational network architecture; infrastructural policies; templates, generic patterns, frameworks and models; AWS service usage guidelines and principles; cloud security; and so on.

We strongly recommend you actively engage Amazon architects throughout your 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, procedures, and so on. This will help your team's technical understanding of the cloud environment, aid rapid on-boarding of new team members, and help establish standards and guidelines in your organization.

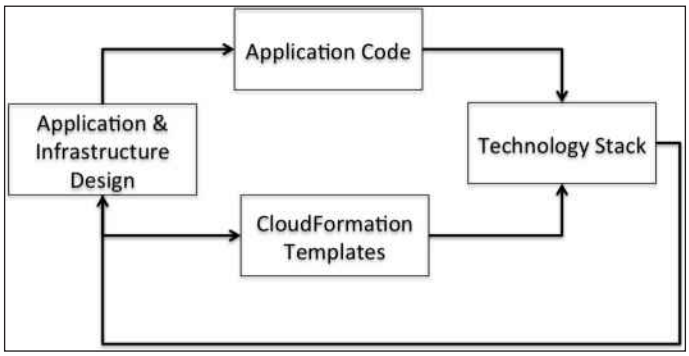
Creating and managing AWS environments using CloudFormation

Your primary goal for deployments includes minimizing the overall time and effort required for it, while having predictability, flexibility, and visibility into 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.

There are several factors at play, simultaneously, while setting up a DevOps pipeline. The layers in your architecture are interdependent, and you want a very high degree of automation and agility. On a different note, there are defined processes and procedures to be followed for production migrations and application upgrades. CloudFormation provides you the orchestration required for fulfilling most of these requirements in a declarative and parameterized manner while managing 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. In case you need to make a change from outside of CloudFormation, then have a process in place to make the appropriate changes in the CloudFormation template before any subsequent stack updates. Ensure that you protect your stacks from accidental or inadvertent changes by strictly managing changes or updates to your templates using 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 chances of errors during 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 code and templates. CloudFormation templates are ideal for provisioning and replication of your application's technology stack across your environments, that is, development, test, staging, and production. 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 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, OS, libraries, and your application packages and/or code. You can define 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. CloudFomation templates that refer to other templates result in nested stacks or a tree of stacks. This is typically done to drive as much reusability as possible in your deployment processes. For example, if you have several websites sharing common requirements in terms of their load balancing and auto scaling features, then you can create a template for your ELB and auto scaling groups, and reuse it across multiple stacks. If your stacks are complex, then you can also use AWS OpsWorks. This option will allow you to leverage a number of predefined stacks and Chef recipes.

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 might include different lifecycles 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 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 CloudFront templates is very similar to software development practices. For example, you will need to develop, conduct code reviews, maintain repositories, version control, test, run, and maintain them. In addition, when you hit errors you will need to debug and fix your code.

In order to minimize the errors and the time taken to develop the production quality, CloudFormation templates ensure that you:

- Validate the template (checks for structure and API usage, JSON syntax, presence of circular dependencies, and so on). The CloudFormation console automatically validates your template after you specify the appropriate input parameters. Alternatively, you can use the AWS CLI, that is, the `validate-template` command or the AWS CloudFormation API to validate your templates.
- 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 your stack creation process so in case of any errors, you will get to 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, number of EC2 instances, RDS storage limits, and so on. There are default limits for AWS services per AWS account. You can request changes to these limits for the services that allow them. One way to verify these limits is to use Trusted Advisor. Trusted Advisor displays your usage and limits for each of the services in a specific region.
- Leverage CloudFormation: Here Init is used to declaratively specify the packages to be installed, users to be created, specific configuration scripts to be executed, and so on. Never include secret keys and access keys in your CloudFormation templates. You can leverage IAM roles to achieve the same result.

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.

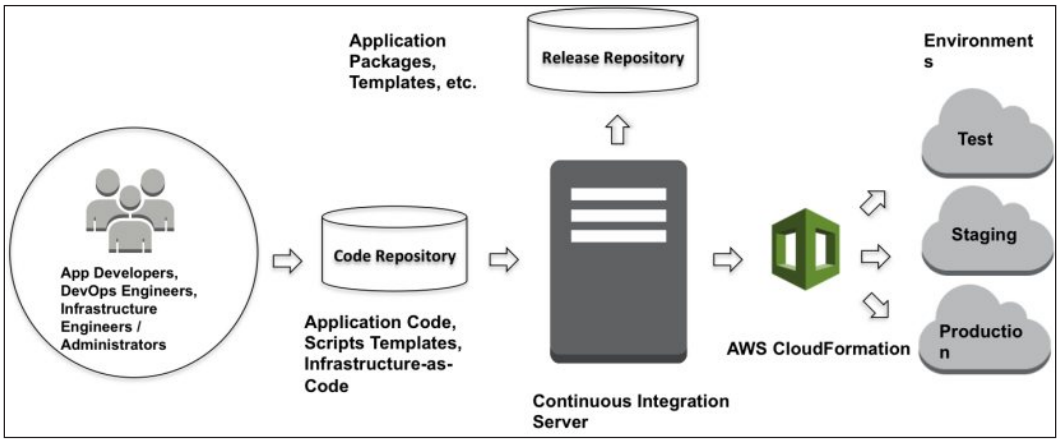
There are other AWS tools that can help you with creating and updating your CloudFormation templates. For example, you can use AWS Config for detecting changes in the stack made from outside of CloudFormation. You can also use CloudFormer to create CloudFormation templates from existing resources in 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 has reduced to minutes, at least in enterprises that have embraced some of these practices. In large enterprises, the average number of deployments across their application portfolio now runs into a couple of hundreds 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.

It is absolutely essential to have automated tests for your application and infrastructure in highly automated DevOps pipelines.

The following diagram illustrates a DevOps pipeline incorporating code repositories, a continuous integration environment, AWS CloudFormation, and application environments. Automated testing in the test and staging environments is almost mandatory for rapid deployments of a well-tested application to the production environment.



CloudFormation is a key part of your DevOps pipeline enabling faster production releases. For example, as shown in the figure, you can setup 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

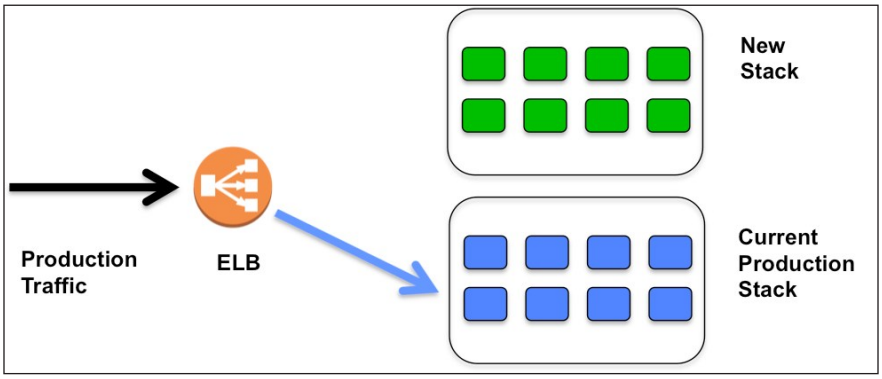
There are primarily two main approaches to updating your stack—in-place and blue-green approach. Each of these approaches has their own pros and cons, and you should select the approach that is most suitable for your specific situation. You can also start with one approach and then move over to a different approach depending on your business needs.

As the name suggests, 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 migration of data and the application state is much simpler than the blue-green approach.

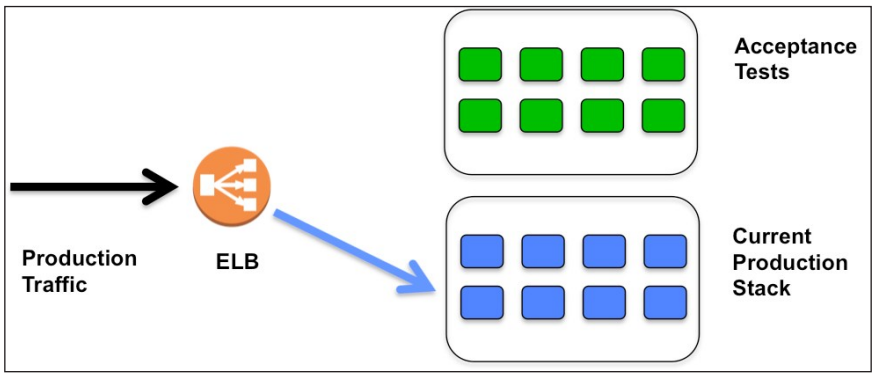
In the blue-green approach, you take the new template and create a completely new and separate stack (from your currently running stack). After you have verified that the new stack is running as per your requirements, you switch production traffic over to it.

The main steps in a blue-green deployment are illustrated in a series as shown in the following steps:

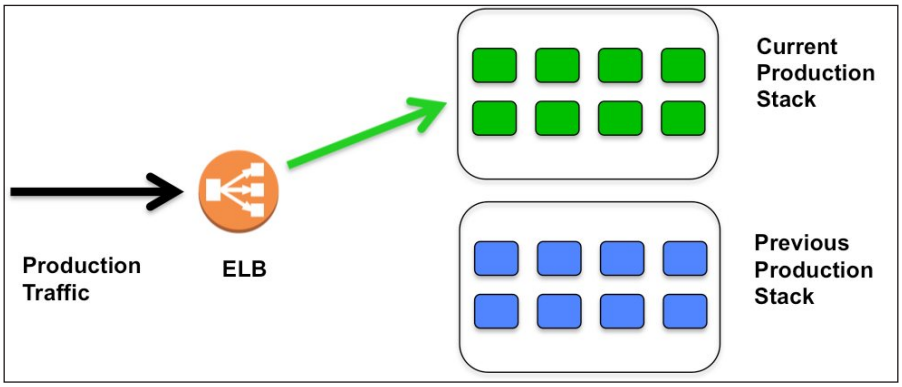
1. Instances with the production stack are labeled blue and the instances hosting the new stack are labeled green. The blue fleet is currently serving all of the production traffic.



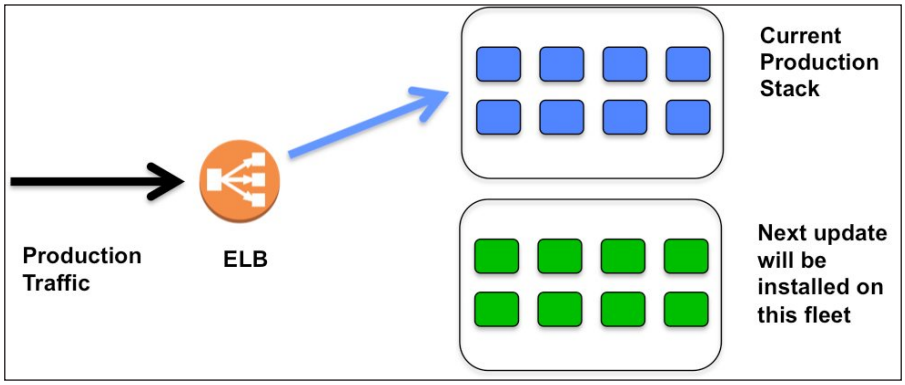
2. Verification and acceptance tests for the new stack are conducted on the green fleet, while the blue fleet continues to serve the production traffic.



3. After the acceptance tests are successfully cleared, the production traffic is switched over to the green fleet.



4. The green fleet is then labeled blue and it is serving all of the production traffic. The fleet that was originally Blue is now labeled Green.



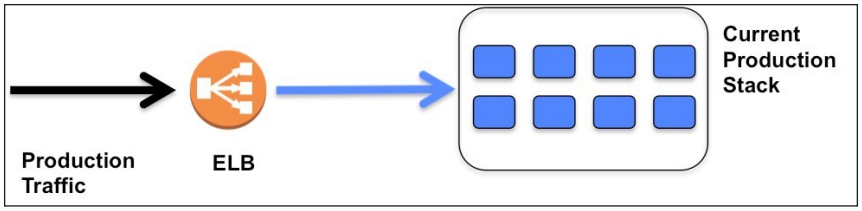
The primary advantage of the blue-green approach is that you are not touching the currently running stack at all. You also have the option to fallback to the old stack at any time, easily. However, blue-green deployments are expensive, as this approach requires you to spin up a duplicate set of instances.

Each of the stack update options has certain desirable characteristics, and you can combine them, appropriately, to evolve an approach that works best for you.

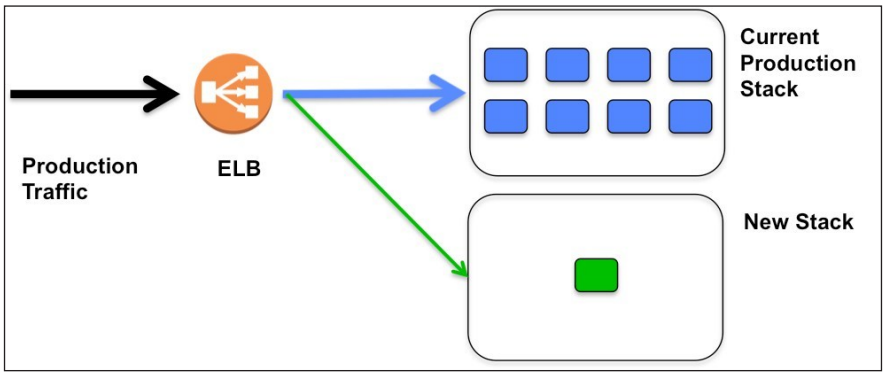
There are several variants of the blue-green approach that address some of the shortcomings of the traditional blue-green deployment approach discussed previously.

An approach that uses a mix of the green and blue instances (with a single ELB) might be useful for certain applications, and is described as follows:

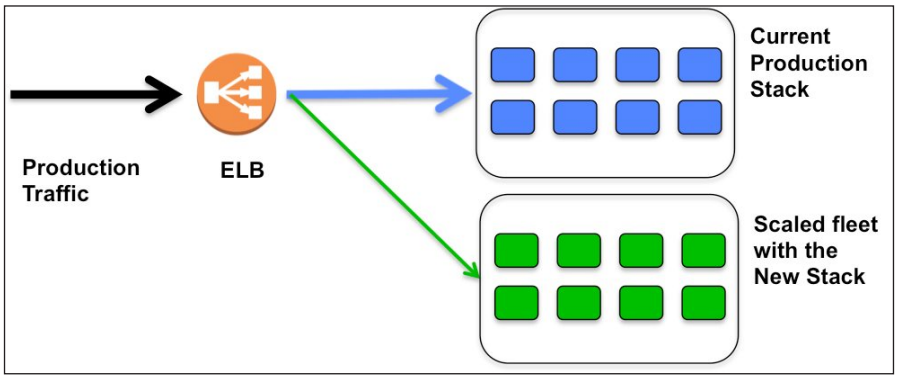
1. The starting state is a single fleet of instances labeled blue (belonging to an auto scaling group). These instances serve all of the production traffic.



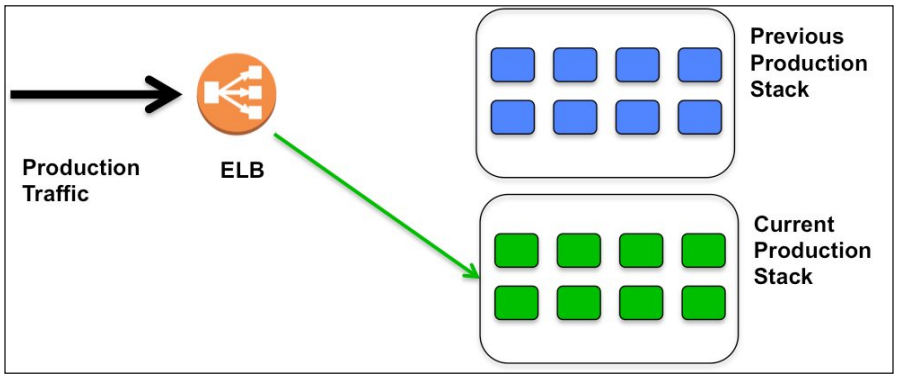
2. In this stage, you create a new instance hosting the new stack (in a separate auto scaling group with a max size of 1). While the blue fleet is serving a majority of production traffic, the green instance also starts serving some of the production traffic.



3. In the next step, you scale up the green fleet.



4. Finally, you switch over to the green fleet.



At this stage, you can shutdown the blue fleet and your green fleet is now the designated blue fleet. This approach takes into consideration that acceptance tests might not be complete or comprehensive. At the same time, there are no DNS changes or ELB warm up required.

A variant of the blue-green approach that works well within an auto scaling group are 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 might also be required for AWS services that are not currently supported by CloudFormation, or if you have a requirement to provision on-premise resources as a part of your stack. Two ways of including such resources in your CloudFormation stack are discussed here.

In the first approach, to achieve 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. On 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 a SNS topic that are subsequently picked up by a provisioning application (that you have to write) to subscribe to the third-party resource. This approach is not as robust as the previous one because CloudFormation is not aware of failures during the provisioning of the third-party service.

Using CloudWatch for monitoring

Amazon CloudWatch enables 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. CloudWatch provides APIs to retrieve 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, specific messages or custom metrics published by the application, and so on. In addition, you can also use the logs to correlate system status with change events such as when AWS CloudFormation is used to roll out 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, or 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 set it up 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 on 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 initially to ingest the logging messages, and then use 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 of 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 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 startups, the cloud might represent their first and only backup solution.

There are third-party solutions such as those from 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 least disruption while you continue to use your existing processes. Cloud storage represents unlimited capacity, so you don't 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 first want to classify data that needs to be kept versus data that need not to be kept. Next, you might classify your data as long term data to be kept for compliance reasons that is unlikely to be restored, very high volume data to be transferred from on-premise storage to the cloud, data shared by multiple applications (document shares, image repositories, and so on), highly available data, data that can be aggregated and then kept in a summarized form, data related to online collaboration applications, and so on. This classification can help you choose appropriate solutions for their storage and backup. AWS provides different storage classes, that is, S3, Glacier, and EBS, to meet varied data lifecycle management requirements. To further reduce costs, you can enable the reduced redundancy option, for less sensitive data, to store fewer copies of your data on the S3 storage. 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 backup 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 restoring. Other guidelines might recommend building stateless services and storing all data on Amazon S3 and leveraging services such as SQS based on assuming that all instances are temporary or will 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 are taking increasing responsibilities for their data.

While evaluating costs 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. In addition, cloud-based solutions provide an easy and convenient solution to offsite backup requirements.

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 nonfunctional requirements) and accepted by the business, all your templates for automated production infrastructure provisioning and deployment scripts tested and ready to go, and backup policies and disaster recovery strategies documented and tested.

For the first few deployments, it is 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. However, it is imperative to fully automate the deployments and the associated verification steps, as soon as you have worked out the wrinkles in your end-to-end deployment process.

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 for sufficient time in your schedule for a couple of iterations of the penetration test. It is a good practice not to do this for the first time after you have deployed in production and management is impatient to go live. You will get through the penetration tests a lot quicker if you have done them a couple times before in you dev, test, and staging environments. This will ensure that you have addressed most of the vulnerability issues before reaching production. After you have cleared the penetration test you should be officially live and actively serving your customers.

At this time, you might 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, tweaks to your infrastructure, and so on.

In the next section, we will walk you through the deployment related activities for our sample application.

Setting up for production

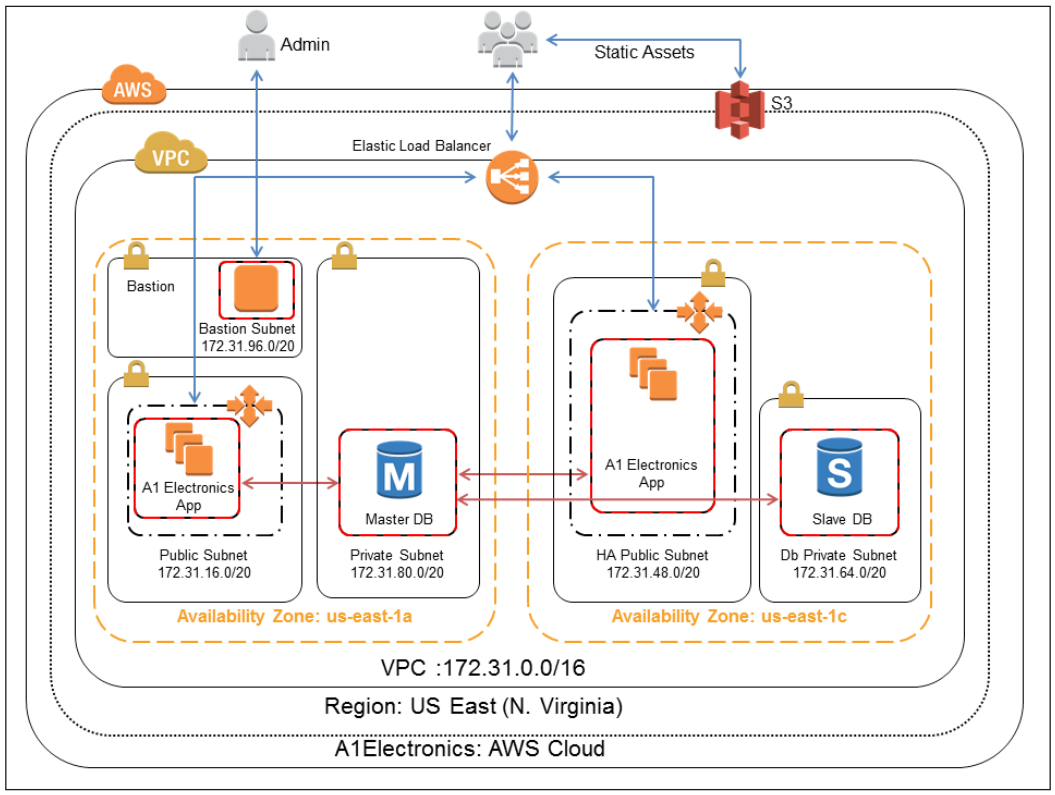
This is it! The final section where AWS will be configured to host the application for production deployment. The key issues in the production setup include health monitoring of the application, disaster recovery, security, costs, configuration management, and process repeatability.

The AWS production deployment architecture

The first step is 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 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 our production setup have been discussed previously.

The following figure represents the deployment architecture for the 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 (firewall) 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. For example, we host the RDS MYSQL database server in a private subnet (172.31.80.0/20) that accepts connections on port 3306 only. This allows access only from the two public subnets (172.31.16.0/20 and 172.31.48.0/20). The VPC is created on 172.31.0.0/16 and the subnets created within the VPC are listed as follows:

- Subnet at 172.31.96.0/20 hosts the bastion host and accepts SSH connections from trusted sources only.
- Public subnets at 172.31.16.0/20 and 172.31.48.0/20 hosting the EC2 instances in the auto scaling group for the application. It accepts HTTP and HTTPS connections from the load balancer security group. The two subnets are in two different availability zones to support HA.
- Private subnets at 172.31.80.0/20 and 172.31.64.0/20 to host the database servers and accept MYSQL connections only from defined web and bastion security groups.

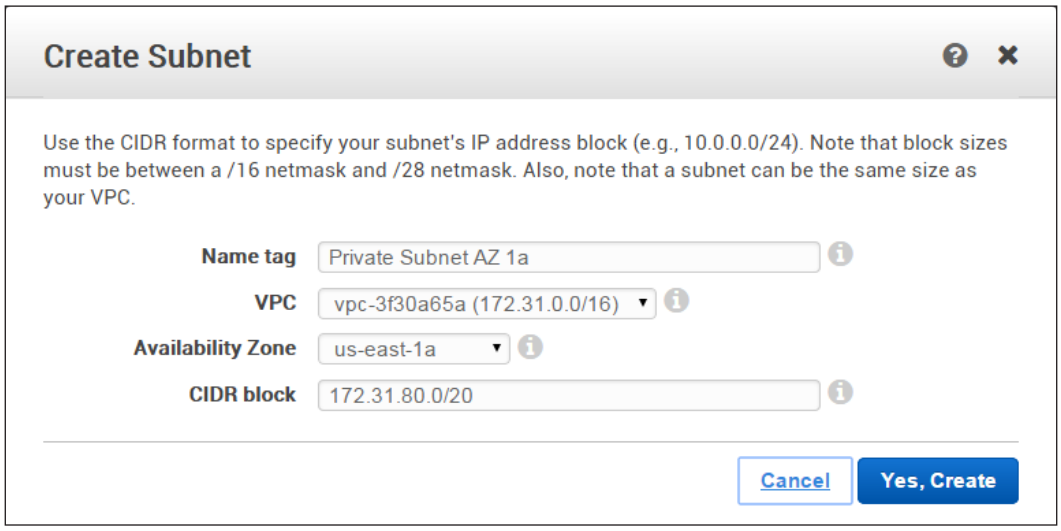
We configured two subnets, 172.31.16.0/20 in AZ us-easts-1a region and 172.31.48.0/20 in AZ us-east-1c region, in *Chapter 5*, *Designing for and Implementing High Availability*. We will continue to use them as our public subnets.

Private subnet

Any EC2 running on a private subnet can be accessed from another EC2 instance within a VPC network or over a VPN network and are not accessible via the public Internet. Each VPC has a default Internet gateway associated with it whose purpose is to route traffic to the Internet. A new subnet is always created as a public subnet and can be changed to a private subnet by assigning its route table to a private route table. Perform the following steps:

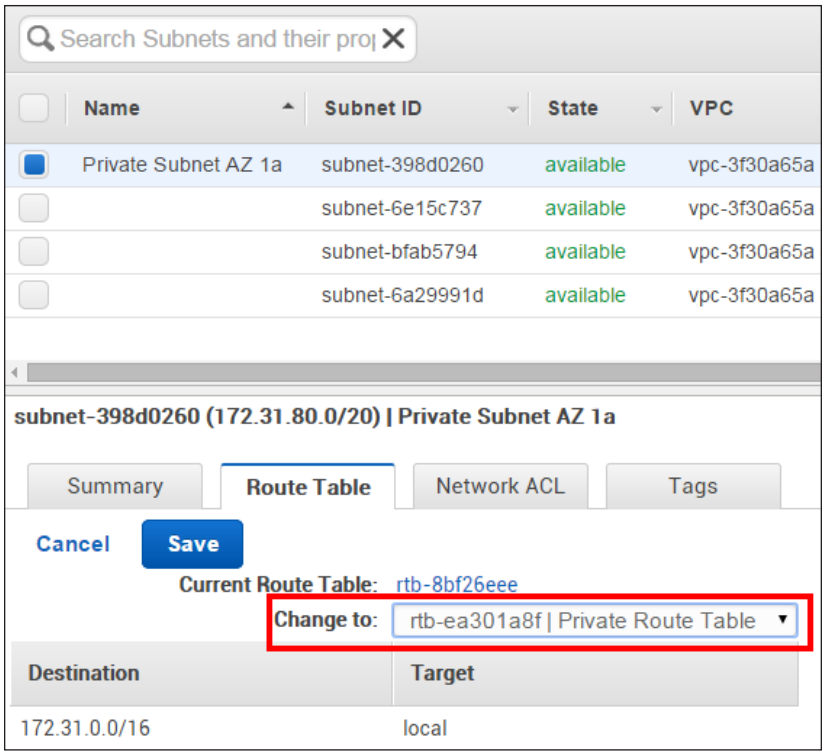
1. The first step is to create a private route table. From the VPC dashboard, navigate to **Route Tables** and click on **Create Route Table**:
    - In the **Create Route Table** popup, assign the name of the route table in the **Name tag**

2. The next step is to create a subnet. From the VPC dashboard, navigate to **Subnets** and then click on **Create Subnets**:
    - **Name tag**: Specify a name for the subnet. This name will be reflected in the VPC dashboard.



    - **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.
    - **Availability Zone**: This is the availability zone in which the subnet will be created. From the dropdown, select **us-east-1a**; this is one of the two private subnets, the other one will be created in the us-east-1c availability zone as per the deployment architecture.
    - **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 4096 hosts).

3. The last step is to associate the private route table created in step 1 to the subnet created in step 2. From the VPC dashboard, navigate to **Subnets** and click on the subnet created in step 2. Navigate to the **Route Table** tab in the bottom pane and click on edit, as shown in the following screenshot:



4. From the **Change To** dropdown, select the route created in step 1.

Similarly, create another subnet Private Subnet with the CIDR block 172.16.48.0/16 in the availability zone us-east-1c and assign the private route table to it (created in step 1).

Bastion subnet

Create another subnet named bastion, with the CIDR block 172.16.96.0/20 in the 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 addresses 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 connection 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 are recommended security groups and their inbound and outbound rules. Please refer to *Chapter 2, Designing Cloud Applications – An Architect's Perspective*.

- The 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	This accepts HTTP traffic from anywhere.
0.0.0.0/0	TCP	8443	This accepts 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.

- Web security group recommended rules: Apply this security group to EC2 instances running in a public network 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	This accepts HTTP traffic from the load balancer.
ID of ELB security group	TCP	8443	This accepts HTTPS traffic from the load balancer.
ID of Bastion security group	TCP	22	This allows SSH traffic from the bastion network.

Outbound			
Destination (CIDR)	**Protocol**	**Port Range**	**Comments**
ID of Database security group	TCP	3306	This allows MYSQL access to the database servers assigned to database security group
0.0.0.0/0	TCP	80	Allow the EC2 instances to connect to the Internet on HTTP port
0.0.0.0/0	TCP	443	Allow the EC2 instances to connect to the Internet on HTTPS port

- Bastion Security Group Recommended Rules: Apply this security group to EC2 instances running in the bastion network.

Inbound			
Source (CIDR)	**Protocol**	**Port Range**	**Comments**
MyIP	TCP	22	This accepts the SSH connection for your fixed static IP. It implies that you can connect to the bastion sever only from this 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**
0.0.0.0/0	TCP	0..65535	Allow to connect on any ports on the Internet and also to the private database security group.

- Database security group recommended rules: Apply this security group to EC2 instances running in a 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	This accepts MYSQL connections from the web application running in the public network.
ID of Bastion security group	TCP	3306	This allow MYSQL access to the database servers for administering MYSQL database.

Outbound			
Destination (CIDR)	**Protocol**	**Port Range**	**Comments**
None			None

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 a good practice to build your cloud infrastructure via code as it is repeatable and can be versioned.

AWS provides services such as CloudFormation, AWS OpWorks, and Code Deploy. CloudFormation focuses on providing foundational capabilities for the full breadth of the 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. CodeDeploy automates application deployment in a reliable and efficient manner to a fleet of EC2 instances and instances running on-premise.

Setting up CloudFormation

While working with CloudFormation you define the AWS resources you need and then wire them together as per your architecture.

The template JSON file includes the following sections; only the `Resources` section is mandatory while the rest are all optional.

The structure of a CloudFormation script consists of the following:

```
{
  "AWSTemplateFormatVersion" : "",
  "Description" : "",
  "Parameters" : {
  },
  "Mappings" : {
  },
  "Conditions" : {
  },
  "Resources" : {
  },
  "Outputs" : {
  }
}
```

- **AWSTemplateFormatVersion**: This defines the capabilities of the template. Only one version has been defined so far and the value for it is `2010-09-09`.
- **Description**: This is free text; use it to include comments for your template. It should always follow AWSTemplateFormatVersion. The maximum size of this free text is 1,024 characters.
- **Parameters**: This 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 a 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 where you define your AWS resources and wire them together. This section is mandatory.
- **Outputs**: In this section, you declare the values to be returned on creation of each AWS resource. This is useful, for example, to find the ELB URL or a RDS endpoint.

Since a full CloudFormation template of the production deployment architecture can run into hundreds of lines of JSON code, we will only present the creation of key AWS resources is presented; however, the complete script (`a1ecommerceaws.json`) is available for download in the source code repository. The code snippets below might not match the script line by line as the emphasis here is more on clarity than on structure.

The key AWS resources are explained below:

- **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"}
          ]
      }
}
```

- **Subnets**: As an example, only one is presented here for the public subnet. Remember to replace CIDR Block with 172.31.16.0/24 and `VPCId` with the default VPC identifier, for example, vpc-3f30a65a and `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-east-1a",
    "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-east1a and us-east-1c:

```
"RDSSubnetGroup":{
"Type" : "AWS::RDS::DBSubnetGroup",
"Properties":{
  "DBSubnetGroupDescription":" Availability Zones for
    A1EcommerceDB",
  "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::Auto scaling::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:

```
"WebServerScaleUpPolicy":{
  "Type":"AWS::Auto scaling::ScalingPolicy",
  "Properties":{
    "AdjustmentType":"ChangeInCapacity",
    "Auto scalingGroupName":{
      "Ref":"WebServerGroup"
    },
    "Cooldown":"60",
    "ScalingAdjustment":"1"
  }
},"
WebServerScaleDownPolicy":{
  "Type":"AWS::Auto scaling::ScalingPolicy",
  "Properties":{
    "AdjustmentType":"ChangeInCapacity",
    "Auto scalingGroupName":{
      "Ref":"WebServerGroup"
    },
```

```
      "Cooldown":"60",
      "ScalingAdjustment":"-1"
    }
}
```

- **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 greater than 90 percent for a period of 5 minutes and an EC2 instance is removed whenever the average CPU load is less than 70 percent for a period of 5 minutes. For 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":"Auto scalingGroupName",
      "Value":{"Ref":"WebServerGroup" }
    }
    ],
    "ComparisonOperator":"GreaterThanThreshold"
  }
},
"CPUAlarmLow":{
},
```

- **Auto Scaling Group**: Finally, the auto scaling group itself is configured to send messages to the SNS topic on launch and termination of EC2 instances:

```
"WebServerGroup":{
  "Type":"AWS::Auto scaling::Auto scalingGroup",
  "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":[
        "auto scaling:EC2_INSTANCE_LAUNCH",
        "auto scaling:EC2_INSTANCE_LAUNCH_ERROR",
        "auto scaling:EC2_INSTANCE_TERMINATE",
        "auto scaling:EC2_INSTANCE_TERMINATE_ERROR"
      ]
    }
  }
},
```

Executing the CloudFormation script

The next step is to execute the CloudFromation script in order to create the AWS resources. There are two ways by which you can achieve this; the first one is by using the Amazon web console and the other by using the Amazon command line from your development machine. Before you execute the script make sure you delete all the AWS resources except the AMI Image you have in the N. Virginia region. By default the CloudFormation script is configured for the N. Virginia region. Cross check the following parameters in the script before you execute the script:

- Substitute the value of `KeyPairName` in the script with the EC2 instance key pair created in *Chapter 3*, *AWS Components, Cost Model, and Application Development Environments* under *Creating EC2 instance key pairs* if required
- Substitute the value of `AMIImageId` in the script with an AMI instance ID created in *Chapter 4*, *Designing for and Implementing Scalability* under *Creating an AMI* if required
- Substitute the values of `DBName`, `DBUser`, and `DBPassword` in the script with the ones created in *Chapter 3*, *AWS Components, Cost Model, and Application Development Environments* under *Amazon Relational Database Service* if required

Now we are ready to execute the CloudFormation script.

Via the command line

The simplest way of executing the CloudFormation script is via the Amazon command line tools that were installed in *Chapter 4*, *Designing for and Implementing Scalability* under *Scripting auto scaling*. If you already have the AWS command line tools installed and configured, only two commands need to be fired, the first one to validate the CloudFormation script is as follows:

```
aws cloudformation validate-template  --template-body file://
a1ecommerceaws.json --region=us-east-1
```

To create the CloudForamtion stack, the second command is as follows:

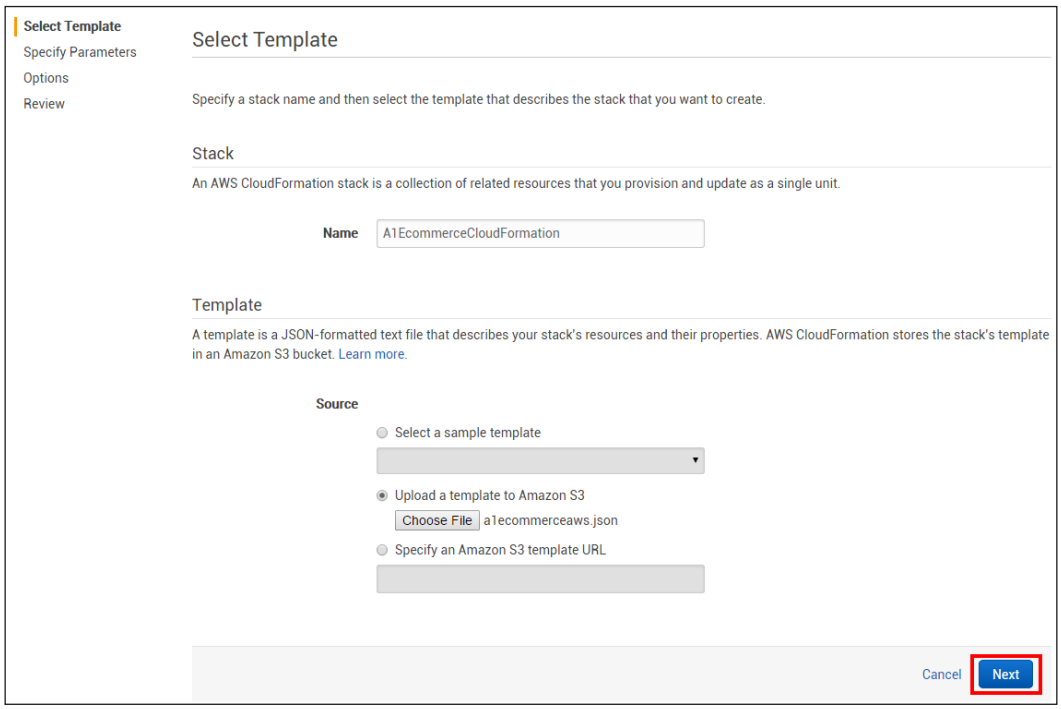
```
aws cloudformation create-stack --stack-name a1ecommerce  --template-body
file://a1ecommerceaws.json --region=us-east-1
```

The progress of the CloudFormation stack can be monitored via the CloudFormation dashboard via the Amazon web console.

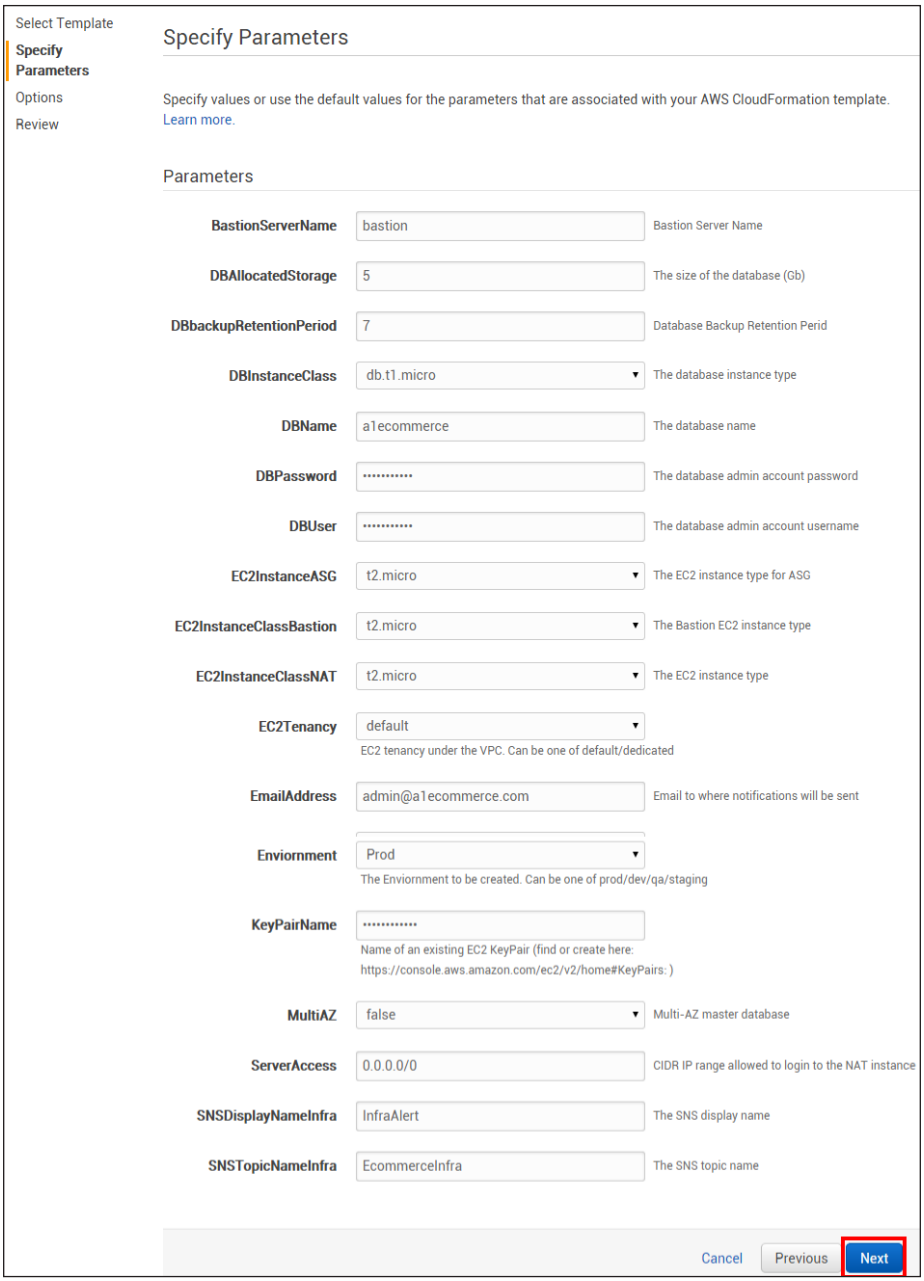
Via the Amazon web console

There is another option to create the CloudFormation stack via the CloudFormation dashboard. Navigate to the CloudFormation dashboard from the Amazon web console and click on **Create Stack**.

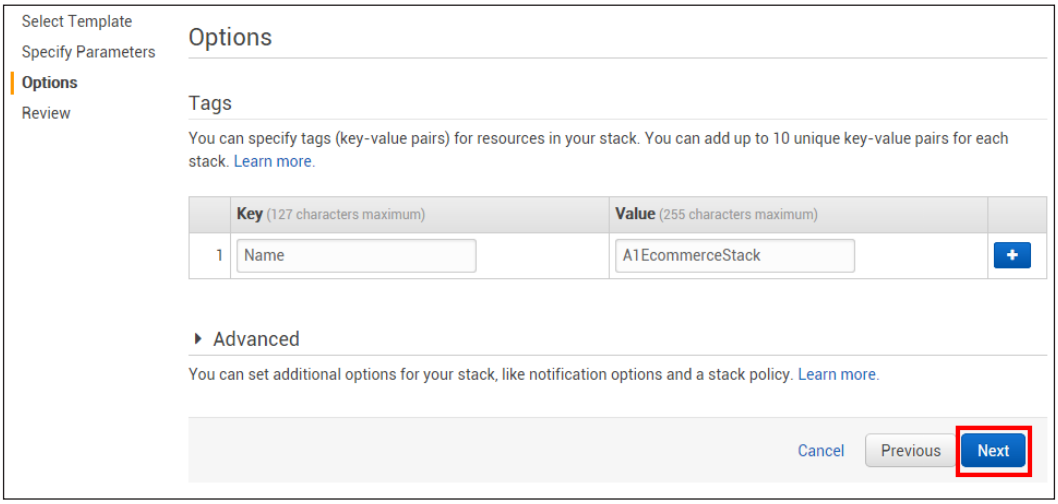
- **Select Template**: The first step is to name the CloudFormation stack and upload the template script to S3. Another option is to provide the path of the template script in S3; this option is useful if a single file is used to create stacks across regions or availability zones within a region.



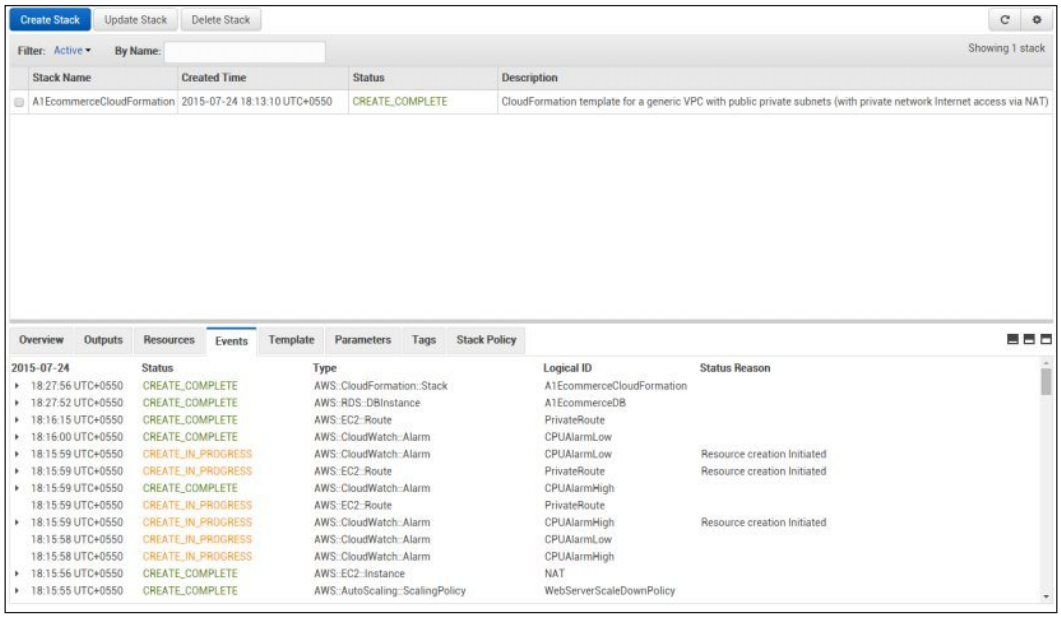
- **Specify Parameters**: This screen allows you to override the default parameters in the template script. Cross check to verify that the values of parameters for **KeyPairName**, **AMIImageId**, **DBName**, **DBUser**, and **DBPassword** are legit.



- **Options**: On this screen, you can specify Tags for your stack and configure certain other advanced options. For instance, you can send notifications of CloudFormation events to an Amazon SNS topic. In addition, you can specify a timeout period for stack creation, enable rollbacks, and specify stack policies for protection of your AWS resources during stack updates.



- **Review**: This screen lists your selected options and parameter overrides for review before creating the stack. Click on **Create** to start the CloudFormation stack. The AWS resources created by the script can be monitored under the **Events** tab in the CloudFormation dashboard:



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 auto scaling 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 OS log their data to; this makes it very convenient to search, view, analyze, and take action on the logs in real time from a centralized console. You have an option of either rolling out your own centralized logging infrastructure using the open source ELK stack (Elasticsearch, Logstash, and Kibana) or subscribing to one of the many third-party logging as a service provider. 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. To do this, perform the following steps:

1. Install the AWS command client library as described in *Chapter 4*, *Designing for and Implementing Scalability*, under 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.
    - Download the CloudWatch logging agent:

      ```
      wget https://s3.amazonaws.com/aws-cloudwatch/downloads/
      latest/awslogs-agent-setup.py
      ```

    - 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 CloudWatch is defined in step 4; make sure the file by the name exists:

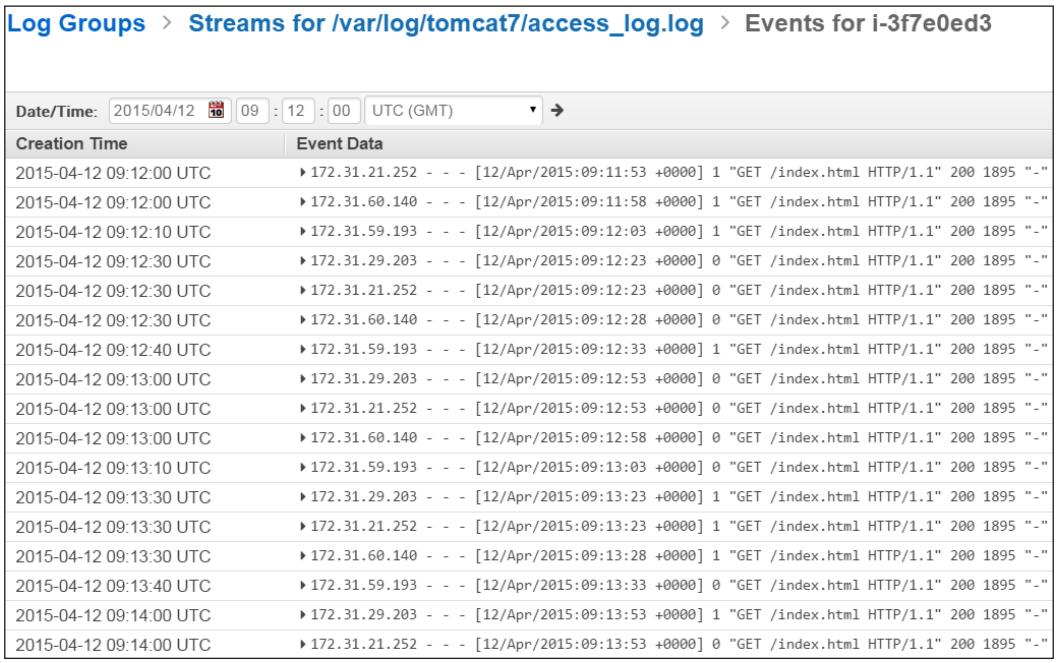
      ```
      sudo python ./awslogs-agent-setup.py --region --us-east-1
      Launching interactive setup of CloudWatch Logs agent ...
      Step 1 of 5: Installing pip ...DONE
      Step 2 of 5: Downloading the latest CloudWatch Logs agent
      bits ... DONE
      Step 3 of 5: Configuring AWS CLI ...
      AWS Access Key ID [None]:
      AWS Secret Access Key [None]:
      Default region name [us-east-1]:
      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
```

- After configuring and installing the CloudWatch instance in your base AMI image, start the logging agent:

```
sudo service awslogs start
```

- 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. Drill down by clicking on the **Log Groups** entry, a list of all the EC2 instances that are configured and are logging to the selected Log Group. Further drilling down and by clicking on any one of the EC2 instances will display the log data:



Summary

In this chapter, we reviewed some of the strategies you can follow for the 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.

Index

A

B

C

D

E

F

G

H

I

T

U

V

W

Thank you for buying

Learning AWS

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

About Packt Enterprise

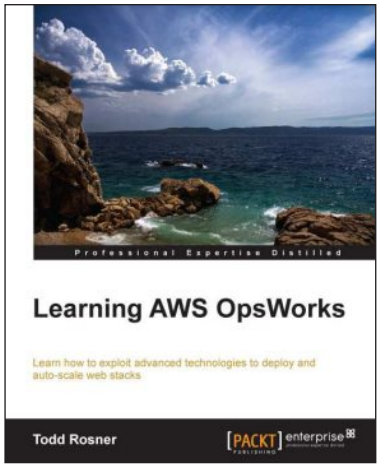
In 2010, Packt launched two new brands, Packt Enterprise and Packt Open Source, in order to continue its focus on specialization. This book is part of the Packt Enterprise brand, home to books published on enterprise software – software created by major vendors, including (but not limited to) IBM, Microsoft, and Oracle, often for use in other corporations. Its titles will offer information relevant to a range of users of this software, including administrators, developers, architects, and end users.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] enterprise
professional expertise distilled
PUBLISHING

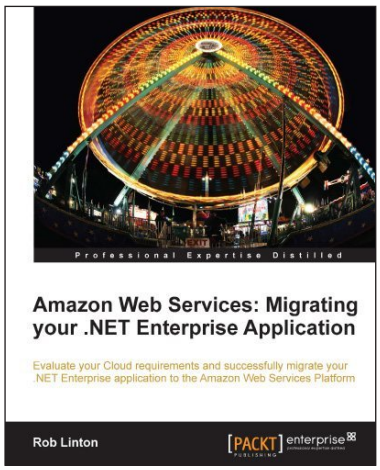


Learning AWS OpsWorks

ISBN: 978-1-78217-110-2 Paperback: 126 pages

Learn how to exploit advanced technologies to deploy and auto-scale web stacks

1. Discover how a DevOps cloud management solution can accelerate your path to delivering highly scalable infrastructure and applications.
2. Learn about infrastructure automation, auto-scaling, and distributed architecture using a Chef-based framework.
3. Includes illustrations, details, and practical examples for successful scaling in the cloud.



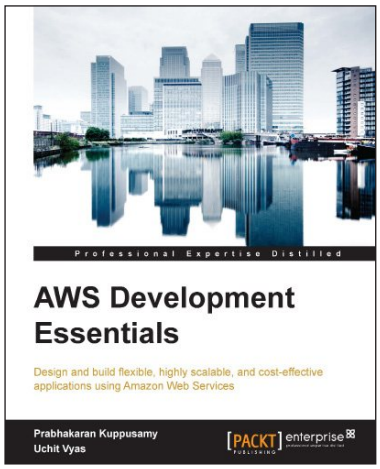
Amazon Web Services: Migrating your .NET Enterprise Application

ISBN: 978-1-84968-194-0 Paperback: 336 pages

Evaluate your Cloud requirements and successfully migrate your .NET Enterprise application to the Amazon Web Services Platform

1. Get to grips with Amazon Web Services from a Microsoft Enterprise .NET viewpoint.
2. Fully understand all of the AWS products including EC2, EBS, and S3.
3. Quickly set up your account and manage application security.

[PACKT] enterprise
professional expertise distilled
PUBLISHING



AWS Development Essentials

ISBN: 978-1-78217-361-8 Paperback: 226 pages

Design and build flexible, highly scalable, and cost-effective applications using Amazon Web Services

1. Integrate and use AWS services in an application.
2. Reduce the development time and billing cost using the AWS billing and management console.
3. This is a fast-paced tutorial that will cover application deployment using various tools along with best practices for working with AWS services.

RESTful Services with ASP.NET Web API [Video]

ISBN: 978-1-78328-575-4 Duration: 02:04 hours

Get hands-on experience of building RESTful services for the modern Web using ASP.NET Web API

1. Apply your current ASP.NET knowledge to make your Web APIs more secure and comply to the global standard in order to make your service RESTful.
2. Explore the possibilities of extending your Web APIs by making use of message handlers, filters, and media formatters.
3. Comprehensive examples to help you build an end-to-end working solution for a real-use case.

Please check **www.PacktPub.com** for information on our titles