

QA 76

.625

.V574

2011

Copy 1

HF GROUP - N 02

1/3/2012

2493112 24



12-B126

Survive Your Success

QA 76
.625
.V574
2011
Copy 1

ramming

Amazon EC2



O'REILLY®

*Jurg van Vliet
& Flavia Paganelli*

Programming Amazon EC2

Programming Amazon EC2

Jurg van Vliet and Flavia Paganelli

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Programming Amazon EC2

by Jurg van Vliet and Flavia Paganelli

Copyright © 2011 I-MO BV. All rights reserved.
Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://my.safaribooksonline.com>). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Editors: Mike Loukides and Julie Steele
Production Editor: Adam Zaremba
Copyeditor: Amy Thomson
Proofreader: Emily Quill

Indexer: John Bickelhaupt
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

Printing History:

February 2011: First Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *Programming Amazon EC2*, the image of a bushmaster snake, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Library of Congress



2011 377697

ISBN: 978-1-449-39368-7

[LSI]

1297357886

Table of Contents

Foreword	ix
Preface	xiii
1. Introducing AWS	1
From 0 to AWS	1
Biggest Problem First	2
Infinite Storage	3
Computing Per Hour	4
Very Scalable Data Store	5
Optimizing Even More	6
Going Global	7
Growing into Your Application	7
Start with Realistic Expectations	8
Simply Small	8
Growing Up	9
Moving Out	10
“You Build It, You Run It”	11
Individuals and Interactions: One Team	12
Working Software: Shared Responsibility	12
Customer Collaboration: Evolve Your Infrastructure	13
Responding to Change: Saying Yes with a Smile	13
In Short	14
2. Starting with EC2, RDS, and S3/CloudFront	15
Setting Up Your Environment	16
Your AWS Account	16
Command-Line Tools	17
AWS Management Console	19
Other Tools	20
Choosing Your Geographic Location, Regions, and Availability Zones	21

Choosing an Architecture	21
Creating the Rails Server on EC2	22
Creating a Key Pair	23
Finding a Suitable AMI	23
Setting Up the Web/Application Server	25
RDS Database	35
Creating an RDS Instance (Launching the DB Instance Wizard)	36
Is This All?	39
S3/CloudFront	41
Setting Up S3 and CloudFront	41
Static Content to S3/CloudFront	43
Making Backups of Volumes	45
Installing the Tools	46
Running the Script	46
In Short	49
3. Growing with S3, ELB, Auto Scaling, and RDS	51
Preparing to Scale	52
Setting Up the Tools	54
S3 for File Uploads	54
User Uploads for Kulitzer (Rails)	54
Elastic Load Balancing	55
Creating an ELB	56
Difficulties with ELB	59
Auto Scaling	60
Setting Up Auto Scaling	60
Auto Scaling in Production	64
Scaling a Relational Database	66
Scaling Up (or Down)	66
Scaling Out	68
Tips and Tricks	69
Elastic Beanstalk	70
In Short	72
4. Decoupling with SQS, SimpleDB, and SNS	73
SQS	73
Example 1: Offloading Image Processing for Kulitzer (Ruby)	74
Example 2: Priority PDF Processing for Marvia (PHP)	77
Example 3: Monitoring Queues in Decaf (Java)	81
SimpleDB	85
Use Cases for SimpleDB	87
Example 1: Storing Users for Kulitzer (Ruby)	88
Example 2: Sharing Marvia Accounts and Templates (PHP)	91

Example 3: SimpleDB in Decaf (Java)	95
SNS	99
Example 1: Implementing Contest Rules for Kulitzer (Ruby)	100
Example 2: PDF Processing Status (Monitoring) for Marvia (PHP)	105
Example 3: SNS in Decaf (Java)	108
In Short	111
5. Managing the Inevitable Downtime	113
Measure	114
Up/Down Alerts	114
Monitoring on the Inside	114
Monitoring on the Outside	118
Understand	122
Why Did I Lose My Instance?	122
Spikes Are Interesting	122
Predicting Bottlenecks	124
Improvement Strategies	124
Benchmarking and Tuning	124
The Merits of Virtual Hardware	125
In Short	127
6. Improving Your Uptime	129
Measure	129
EC2	130
ELB	131
RDS	132
Using Dimensions from the Command Line	133
Alerts	134
Understand	136
Setting Expectations	136
Viewing Components	137
Improvement Strategies	138
Planning Nonautoscaling Components	138
Tuning Auto Scaling	138
In Short	138
7. Managing Your Decoupled System	141
Measure	141
S3	142
SQS	142
SimpleDB	149
SNS	152
Understand	153

Imbalances	154
Bursts	154
Improvement Strategies	154
Queues Neutralize Bursts	155
Notifications Accelerate	155
In Short	156
8. And Now...	157
Other Approaches	157
Private/Hybrid Clouds	158
Thank You	158
Index	159

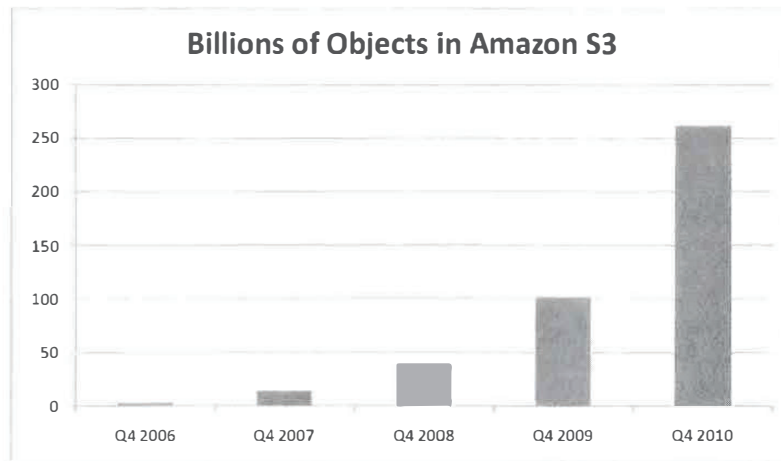
Foreword

March 14, 2006, was an important day, even though it is unlikely that it will ever become more than a footnote in some history books. On that day, Amazon Web Services launched the first of its utility computing services: the Amazon Simple Storage Service (Amazon S3). In my eyes that was the day that changed the way IT was done; it gave everyone access to an ultra-reliable and highly scalable storage service without having to invest tens of thousands of dollars for an exclusive enterprise storage solution. And even better, the service sat directly on the Internet, and objects were directly HTTP addressable.

The motivation behind the launch of the service was simple: the AWS team had asked itself what innovation could happen if it could give everyone access to the same scalable and reliable technologies that were available to Amazon engineers. A student in her dorm room could have an idea that could become the next Amazon or the next Google, and the only thing that would hold her back was access to the resources needed to fulfill that potential. AWS aimed at removing these barriers and constraints so people could unleash their innovation and focus on building great new products instead of having to invest in infrastructure both intellectually and financially.

Today, Amazon S3 has grown to store more than 260 billion objects and routinely runs more than 200,000 storage operations per second. The service has become a fundamental building block for many applications, from enterprise ERP log files to blog storage, streaming videos, software distribution, medical records, and astronomy data.

By routinely running over 200,000 storage operations per second, Amazon S3 is a marvel of technology under the covers. It is designed to support a wide range of usage scenarios and is optimized in very innovative ways to make sure every customer gets great service, regardless of whether he is streaming videos or just housing some home photos. One of my colleagues had a great analogy about how the Amazon S3 software had to evolve: it was like starting with a single-engine Cessna that had to be rebuilt into a Boeing 747 while continuing to fly and continuously refueling, and with passengers that changed planes without noticing it. The Amazon S3 team has done a great job of making the service something millions and millions of people rely on every day.



Following Amazon S3, we launched Amazon Simple Queue Service (Amazon SQS), and then Amazon Elastic Compute Cloud (Amazon EC2) just a few months later. These services demonstrated the power of what we have come to call Cloud Computing: access to highly reliable and scalable infrastructure with a utility payment model that drives innovation and dramatically shortens time to market for new products. Many CIOs have told me that while their first motivation to start using AWS was driven by the attractive financial model, the main reason for staying with AWS is that it has made their IT departments agile and allowed them to become enablers of innovation within their organization.

The AWS platform of technology infrastructure services and features has grown rapidly since that day in March 2006, and we continue to keep that same quick pace of innovation and relentless customer focus today.

Although AWS, as well as its ecosystem, has launched many tools that make using the services really simple, at its core it is still a fully programmable service with incredible power, served through an API. Jurg and Flavia have done a great job in this book of building a practical guide for how to build real systems using AWS. Their writing is based on real experiences using each and every one of the AWS services, and their advice is rooted in building foundations upon which applications on the AWS platform can scale and remain reliable. I first came in contact with them when they were building Decaf, an Android application used to control your AWS resources from your mobile device. Since then, I have seen them help countless customers move onto the AWS platform, and also help existing customers scale better and become more reliable while taking advantage of the AWS elasticity to drive costs down. Their strong customer focus makes them great AWS partners.

[illegible]

AWS will continue to innovate on behalf of our customers, and there are still very exciting things to come.

—Werner Vogels

Preface

Thank you for picking up a copy of this book. Amazon Web Services (AWS) has amazed everyone: Amazon has made lots of friends, and all its “enemies” are too busy admiring AWS to do much fighting back. At the moment, there is no comparable public Infrastructure as a Service (IaaS); AWS offers the services at a scale that has not been seen before. We wrote this book so you can get the most out of AWS’ services. If you come from conventional hardware infrastructures, once you are on AWS, you won’t want to go back.

AWS is not easy; it combines skills of several different (established) crafts. It is different from traditional systems administration, and it’s not just developing a piece of software. If you have practiced one or both of these skills, all you need is to be inquisitive and open to learning.

Our background is in software engineering. We are computer scientists with extensive software engineering experience in all sorts of different fields and organizations. But the cloud in general and AWS in particular caught our interest some years ago. We got serious about this by building Decaf, an Android smartphone application that manages Amazon EC2 (Elastic Compute Cloud) accounts. We were finalists in the Android Developer Challenge in 2009. We will use Decaf to illustrate various AWS services and techniques throughout this book.

Around the same time, in early 2010, we decided we wanted to build applications on AWS. We founded 9Apps and set out to find a select group of partners who shared our development interests. Our expertise is AWS, and our responsibility is to keep it running at all times. We design, build, and operate these infrastructures.

Much of our experience comes from working with these teams and building these applications, and we will use several of them as examples throughout the book. Here is a short introduction to the companies whose applications we will use:

Directness

Directness helps customers connect brands to businesses. With a set of tools for making surveys and collecting, interpreting, and presenting consumers’ feedback, this application is very successful in its approach and works with a number of international firms. The problem is scaling the collection of customer responses,

transforming it into usable information, and presenting it to the client. Directness can only grow if we solve this problem.

Kulitzer

Kulitzer is a web application that allows users to organize creative contests. Users can invite participants to enter the contest, an audience to watch, and a jury to pick a winner. Technically, you can consider Kulitzer a classical consumer web app.

Layar

Layar is an augmented reality (AR) smartphone browser that is amazing everyone. This application enriches the user's view of the world by overlapping different objects or information in the camera view, relevant to the location. For example, users can see what people have been tweeting near them, the houses that are for sale in the neighborhood, or tourist attractions near where they are walking.

The Layar application continues to win prize after prize, and is featured in many technical and mainstream publications. Layar started using Google App Engine for its servers, but for several reasons has since moved to AWS.

Marvia

Ever needed to create some "print ready" PDFs? It's not an easy task. You probably needed desktop publishing professionals and the help of a marketing agency, all for a significant price tag. Marvia is an application that can dramatically reduce the effort and cost involved in PDF creation. It allows you to create reusable templates with a drag-and-drop web application. Or you can integrate your own system with Marvia's API to automate the generation of leaflets and other material.

Publitas

Publitas does the opposite of what Marvia does, in a way. It lets you transform your traditional publication material to an online experience. The tool, called ePublisher, is very feature-rich and is attracting a lot of attention. You can input your material in PDF format to the application and it will generate online content. You can then enrich the content with extra functionality, such as supporting sharing in social networks and adding music, video, search, and print. The challenge with the Publitas software is that its existing customers are established and well-known businesses that are sometimes already so powerful that exposure ratings resemble those of a mass medium like television.

Audience

Of course, we welcome all readers of this book, and we hope it inspires you to get into AWS and utilize it in the best possible way to be successful. But we set out to write this book with a particular purpose: to be an AWS guide for building and growing applications from small to "Internet scale." It will be useful if you want to host your blog or small web application, but it will also help you grow like Zynga did with Farmville. (Some say Zynga is the fastest growing company in the world.)

This book does not focus on detail; for example, we are not going to tell you exactly which parameters each command receives, and we are not going to list all the available commands. But we will show you the approach and implementation. We rely on examples to illustrate the concepts and to provide a starting point for your own projects. We try to give you a sense of all AWS functionality, which would be nearly impossible if we were to show the details of every feature.

To get the most out of this book, you should be comfortable with the command line, and having experience writing software will be useful for some of the chapters. And it certainly wouldn't hurt if you know what Ubuntu is (or CentOS or Windows 2003, for that matter) and how to install software. But most of all, you should simply be curious about what you can do with AWS. There's often more than one way of doing things, and since AWS is so new, many of those ways have not yet been fully explored.

If you are a seasoned software/systems engineer or administrator, there are many things in this book that will challenge you. You might think you know it all. Well, you don't!

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width bold

Shows commands or other text that should be typed literally by the user.

Constant width italic

- * Shows text that should be replaced with user-supplied values or by values determined by context.



This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Programming Amazon EC2* by Jurg van Vliet and Flavia Paganelli. Copyright 2011 I-MO BV, 978-1-449-39368-7."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

Safari® Books Online



Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at <http://my.safaribooksonline.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

<http://www.oreilly.com/catalog/9781449393687>

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, courses, conferences, and news, see our website at <http://oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://www.youtube.com/oreillymedia>

Acknowledgments

There are many people we would like to thank for making this book into what it is now. But first of all, it would never have been possible without our parents, Aurora Gómez, Hans van Vliet, Marry van Vliet, and Ricardo Paganelli.

Right from the start we have been testing our ideas with many friends and colleagues; their early feedback shaped this book significantly. Thanks to Adam Dorell, Arjan van Woensel, Björn van Vliet, Dirk Groten, Eduardo Röhr, Eric Hammond, Federico Mikaelian, Fleur van Vliet, Grant Wilson, Indraneel Bommisetty, Joerg Seibel, Khalil Seyedmehdi, Marten Mickos, Matt Miesnieks, Pambo Pascalides, Pedro Moranga Gonçalves, Pim Dernelen, Roy Chandra, Steven van Wel, Werner Vogels, Wouter Broekhof, and Zoran Kovačević.

Of course, you need “strange eyes” going over every detail and meticulously trying out examples to find errors. Our technical reviewers, Anthony Maës, Ben Immanuel, Graziano Obertelli, and Menno van der Sman, did just that.

And finally, there is the wonderful and extremely professional team at O'Reilly. Without Mike, Julie, and all the others there wouldn't even have been a book. To Amy Thomson, Adam Zaremba, Julie Steele, Mike Loukides, Sumita Mukherji, and the rest we met and worked with, thank you!

Introducing AWS

From 0 to AWS

By the late 1990s, Amazon had proven its success—it showed that people were willing to shop online. Amazon generated \$15.7 million in sales in 1996, its first full fiscal year. Just three years later, Amazon saw \$1.6 billion in sales, and Jeff Bezos was chosen Person of the Year by Time magazine (<http://www.time.com/time/subscriber/personoftheyear/archive/stories/1999.html>). Realizing its sales volume was only 0.5% that of Wal-Mart, Amazon set some new business goals. One of these goals was to change from shop to platform.

At this time, Amazon was struggling with its infrastructure. It was a classic monolithic system, which was very difficult to scale, and Amazon wanted to open it up to third-party developers. In 2002, Amazon created the initial AWS, an interface to programmatically access Amazon's features. This first set of APIs is described in the wonderful book *Amazon Hacks* (<http://oreilly.com/catalog/9780596005429>) by Paul Bausch (O'Reilly), which still sits prominently on one of our shelves.

But the main problem persisted—the size of the Amazon website was just too big for conventional (web) application development techniques. Somehow, Jeff Bezos found Werner Vogels (now CTO of Amazon) and lured him to Amazon in 2004 to help fix these problems. And this is when it started for the rest of us. The problem of size was addressed, and slowly AWS transformed from “shop API” to an “infrastructure cloud.” To illustrate exactly what AWS can do for you, we want to take you through the last six years of AWS evolution (see Figure 1-1 for a timeline). This is not just a historical journey, but also a friendly way to introduce the most important components for starting with AWS.

AWS has two unique qualities:

- It doesn't cost much to get started. For example, you don't have to buy a server to run it.

- It scales and continues to run at a low cost. For example, you can scale elastically, only paying for what you need.

The second quality is by design, since dealing with scale was the initial problem AWS was designed to address. The first quality is somewhat of a bonus, but Amazon has really used this quality to its (and our) advantage. No service in AWS is useless, so let's go through them in the order they were introduced, and try to understand what problems they were designed to solve.

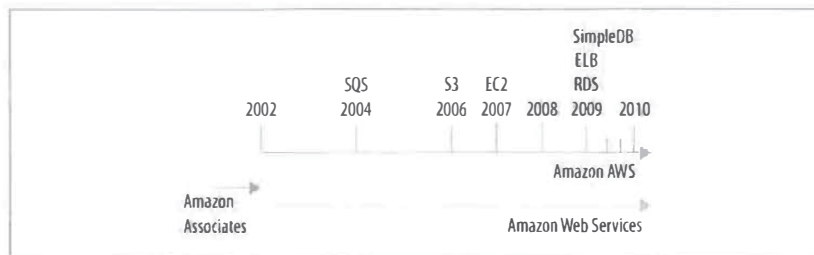


Figure 1-1. Timeline of AWS

Biggest Problem First

If your system gets too big, the easiest (and perhaps only) solution is to break it up into smaller pieces that have as few dependencies on each other as possible. This is often referred to as *decoupling*. The first big systems that applied this technique were not web applications; they were applications for big corporations like airlines and banks. These applications were built using tools such as CORBA (<http://www.omg.org/>) and the concept of “component-based software engineering.” Similar design principles were used to coin the more recent term service-oriented architecture or SOA (http://en.wikipedia.org/wiki/Service-oriented_architecture) which is mostly applied to web applications and their interactions.

Amazon adopted one of the elements of these broker systems, namely *message passing*. If you break up a big system into smaller components, they probably still need to exchange some information. They can pass messages to each other, and the order in which these messages are passed is often important. The simplest way of organizing a message passing system, respecting order, is a queue (Figure 1-2). And that is exactly what Amazon built first in 2004: Amazon Simple Queue Service or SQS (<http://aws.amazon.com/sqs/>).

By using SQS, according to AWS, “developers can simply move data between distributed components of their applications that perform different tasks, without losing messages or requiring each component to be always available.” This is exactly what Amazon needed to start deconstructing its own monolithic application. One interesting feature of SQS is that you can rely on the queue as a buffer between your components,

implementing *elasticity*. In many cases, your web shop will have huge peaks, generating 80% of the orders in 20% of the time. You can have a component that processes these orders, and a queue containing them. Your web application puts orders in the queue, and then your processing component can work on the orders the entire day without overloading your web application.

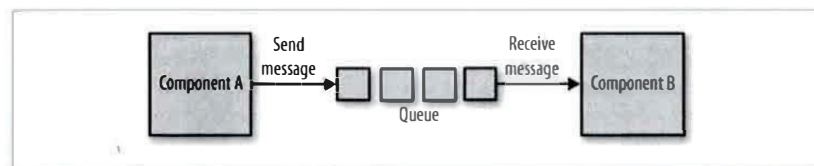


Figure 1-2. Passing messages using a queue

Infinite Storage

In every application, storage is an issue. There is a very famous quote attributed to Bill Gates that 640 K “ought to be enough for anybody.” Of course, he denies having said this, but it does hit a nerve. We all buy hard disks believing they will be more than enough for our requirements, but within two years we already need more. It seems there is always something to store and there is never enough space to store it. What we need is infinite storage.

To fix this problem once and for all, Amazon introduced Amazon Simple Storage Service or S3 (<http://aws.amazon.com/s3/>). It was released in 2006, two years after Amazon announced SQS. The time Amazon took to release it shows that storage is not an easy problem to solve. S3 allows you to store objects of up to 5 terabytes, and the number of objects you can store is unlimited. An average DivX is somewhere between 600 and 700 megabytes. Building a video rental service on top of S3 is not such a bad idea, as Netflix realized.

According to AWS, S3 is “designed to provide 99.999999999% durability and 99.99% availability of objects over a given year.” This is a bit abstract, and people often ask us what it means. We have tried to calculate it ourselves, but the tech reviewers did not agree with our math skills. So this is the perfect opportunity to quote someone else. According to Amazon Evangelist Jeff Barr, this many 9s means that, “If you store 10,000 objects with us, on average we may lose one of them every 10 million years or so.” Impressive! S3 as a service is covered by a service level agreement (SLA), making these numbers not just a promise but a full contract.

S3 was extremely well received. Even Microsoft was (or is) one of the customers using S3 as a storage solution, as advertised in one of the announcements of AWS: “Global enterprises like Microsoft are using Amazon S3 to dramatically reduce their storage costs without compromising scale or reliability” (<http://aws.amazon.com/about-aws/whats-new/2006/07/11/amazon-simple-storage-service-amazon-s3---continuing-suc>

cesses/). In only two years, S3 grew to store 10 billion objects. In early 2010, AWS reported to store 102 billion objects in S3. Figure 1-3 illustrates the growth of S3 since its release.

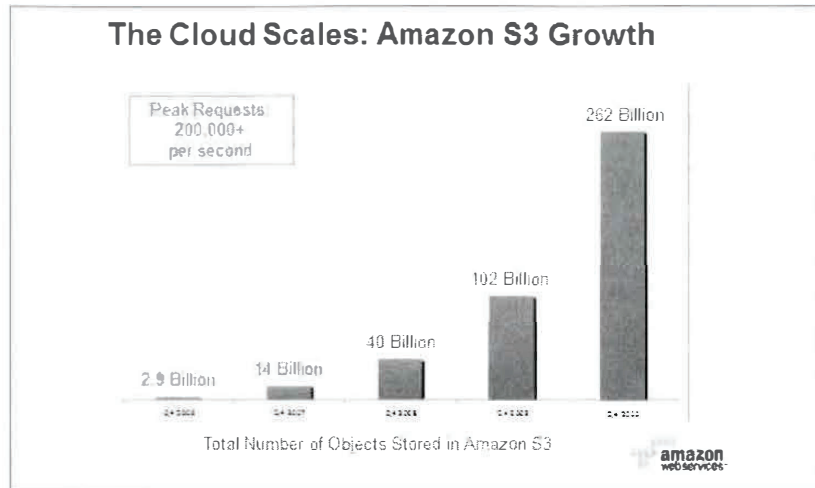


Figure 1-3. S3's huge popularity expressed in objects stored

Computing Per Hour

Though we still think that S3 is the most revolutionary of services because no one had solved the problem of unlimited storage before, the service with the most impact is undoubtedly Amazon Elastic Compute Cloud or EC2 (<http://aws.amazon.com/ec2/>). Introduced as limited beta in the same year that S3 was launched (2006), EC2 turned computing upside down. AWS used XEN virtualization to create a whole new cloud category, Infrastructure as a Service, long before people started googling for IaaS. Though server virtualization already existed for quite a while, buying one hour of computing power in the form of a Linux (and later Windows) server did not exist yet.

Remember, Amazon was trying to decouple, to separate its huge system into components. For Amazon, EC2 was the logical missing piece of the puzzle because Amazon was in the middle of implementing a strict form of SOA. In Amazon's view, it was necessary to change the organization. Each team would be in charge of a functional part of the application, like wish lists or search. Amazon wanted each (small) team not only to build its own infrastructure, but also for developers to operate their apps themselves. Werner Vogels said it in very simple terms: "You build it, you run it."

In 2007, EC2 was opened to everyone, but it took more than a year before AWS announced general availability, including SLA. There were some very important features added in the meantime, most of them as a result of working with the initial community of EC2 users. During this period of refining EC2, AWS earned the respect of the development community. It showed that Amazon listened and, more importantly, cared. And this is still true today. The Amazon support forum is perhaps its strongest asset.

By offering computing capacity per hour, AWS created elasticity of infrastructures from the point of view of the application developer (which is also our point of view.) When it was this easy to launch servers, which Amazon calls *instances*, a whole new range of applications became reachable to a lot of people. Event-driven websites, for example, can scale up just before and during the event and can run at low capacity the rest of the time. Also, computational-intensive applications, such as weather forecasting, are much easier and cheaper to build. Renting one instance for 10,000 hours is just as cheap as renting 10,000 instances for an hour.

Very Scalable Data Store

Amazon's big system is decoupled with the use of SQS and S3. Components can communicate effectively using queues and can share large amounts of data using S3. But these services are not sufficient as glue between the different applications. In fact, most of the interesting data is structured and is stored in shared databases. It is the relational database that dominates this space, but relational databases are not terribly good at scaling, at least for commodity hardware components. Amazon introduced Relational Database Server (RDS) recently, sort of "relational database as a service," but its own problem dictated that it needed something else first.

Although normalizing data is what we have been taught, it is not the only way of handling information. It is surprising what you can achieve when you limit yourself to a searchable list of structured records. You will lose some speed on each individual transaction because you have to do more operations, but you gain infinite scalability. You will be able to do many more simultaneous transactions. Amazon implemented this in an internal system called Dynamo, and later, AWS launched Amazon SimpleDB (<http://aws.amazon.com/simpledb/>).

It might appear that the lack of joins severely limits the usefulness of a database, especially when you have a client-server architecture with dumb terminals and a mainframe server. You don't want to ask the mainframe seven questions when one would be enough. A browser is far from a dumb client, though. It is optimized to request multiple sources at the same time. Now, with a service specially designed for many parallel searches, we have a lot of possibilities. By accessing a user's client ID, we can get her wish list, her shopping card, and her recent searches, all at the same time.

There are alternatives to SimpleDB, and some are more relational than others. And with the emergence of big data, this field, also referred to as NoSQL, is getting a lot of attention. But there are a couple of reasons why it will take time before SimpleDB and others will become successful. The most important reason is that we have not been taught to think without relations. Another reason is that most frameworks imply a relational database for their models. But SimpleDB is incredibly powerful. It will take time, but slowly but SimpleDB will surely find its place in (web) development.

Optimizing Even More

The core principle of AWS is optimization, measured in hardware utilization. From the point of view of a cloud provider like AWS, you need economies of scale. As a developer or cloud consumer, you need tools to operate these infrastructure services. By listening to its users and talking to prospective customers, AWS realized this very point. And almost all the services introduced in this last phase are meant to help developers optimize their applications.

One of the steps of optimization is creating a service to take over the work of a certain task. An example we have seen before is S3, which offers storage as a service. A common task in web (or Internet) environments is load balancing. And just as with storage or queues, it would be nice to have something that can scale more or less infinitely. AWS introduced a service called Elastic Load Balancing or ELB (<http://aws.amazon.com/elasticloadbalancing/>) to do exactly this.

When the workload is too much for one instance, you can start some more. Often, but not always, such a group of instances doing the same kind of work is behind an Elastic Load Balancer (also called an ELB). To manage a group like this, AWS introduced Auto Scaling (<http://aws.amazon.com/autoscaling/>). With Auto Scaling you can define rules for growing and shrinking a group of instances. You can automatically launch a number of new instances when CPU utilization or network traffic exceeds certain thresholds, and scale down again on other triggers.

To optimize use, you need to know what is going on; you need to know how the infrastructure assets are being used. AWS introduced CloudWatch to monitor many aspects of the infrastructure assets. With CloudWatch, it is possible to measure metrics like CPU utilization, network IO, and disk IO over different dimensions like an instance or even all instances in one region.

AWS is constantly looking to optimize from the point of view of application development. It tries to make building web apps as easy as possible. In 2009, it created RDS, a managed MySQL service, which eases the burden of optimization, backups, scaling, etc. Early in 2010, AWS introduced the high availability version of RDS. AWS also complemented S3 with CloudFront, a very cheap content delivery network, or CDN. CloudFront now supports downloads and streaming and has many edge locations around the world.

Going Global

AWS first launched on the east coast of the United States, in northern Virginia. From the start, the regions were designed with the possibility of failure in mind. A region consists of *availability zones*, which are physically separate data centers. Zones are designed to be independent, so failure in one doesn't affect the others. When you can't use this feature of AWS, because it can harden your application.

While AWS was adding zones to the US East region, it also started building new regions. The second to come online was Europe, in Ireland. And after that, AWS opened another region in the US, on the west coast in northern California. One highly anticipated new region was expected (and hinted at) in Asia Pacific. And in April 2010, AWS opened region number four in Singapore.

Growing into Your Application

In 2001, the Agile Manifesto (http://en.wikipedia.org/wiki/Agile_Manifesto) for software development was formulated because a group of people felt it was necessary to have more lightweight software development methodologies than were in use at that time. Though this movement has found its place in many different situations, it can be argued that the Web was a major factor in its widespread adoption. Application development for the Web has one major advantage over packaged software: in most cases it is distributed exactly once. Iterative development is much easier in such an environment.

Iterative (agile) infrastructure engineering is not really possible with physical hardware. There is always a significant hardware investment, which almost always results in scarcity of these resources. More often than not, it is just impossible to take out a couple of servers to redesign and rebuild a critical part of your infrastructure. With AWS, you can easily build your new application server, redirect production traffic when you are ready, and *terminate* the old servers. For just a few dollars, you can upgrade your production environment without the usual stress.

This particular advantage of clouds over physical hardware is important. It allows for applying an agile way of working to infrastructures, and lets you iteratively grow into your application. You can use this to create room for mistakes, which are made everywhere. It also allows for stress testing your infrastructure and scaling out to run tens or even hundreds of servers. And, as we did in the early days of Layar (<http://www.layar.com/>), you can move your entire infrastructure from the United States to Europe in just a day.

In the following sections, we will look at the AWS services you can expect to use in the different iterations of your application.

Start with Realistic Expectations

When asking the question, “Does the application have to be highly available?”, the answer is usually a clear and loud “yes.” This is often expensive, but the expectation is set and we work very hard to live up to it. If you ask the slightly different question, “Is it acceptable to risk small periods of downtime provided we can restore quickly without significant loss of data?”, the answer is the same, especially when it becomes clear that this is much less expensive. Restoring quickly without significant loss of data is difficult with hardware, because you don’t always have spare systems readily available. With AWS, however, you have all the spare resources you want. Later, we’ll show you how to install the necessary command-line tools, but all you need to start five servers is:

```
$ ec2-run-instances ami-480df921 -n 5
```

When it is necessary to handle more traffic, you can add servers—called *instances* in EC2—to relieve the load on the existing infrastructure. After adjusting the application so it can handle this changing infrastructure, you can have any number of instances doing the same work. This way of scaling—*scaling out*—offers an interesting opportunity. By creating more instances doing the same work, you just made that part of your infrastructure highly available. Not only is your system able to handle more traffic or more load, it is also more resilient. One failure will no longer bring down your app.

After a certain amount of scaling out, this method won’t work anymore. Your application is probably becoming too complex to manage. It is time for something else; the application needs to be broken up into smaller, interoperating applications. Luckily, the system is agile and we can isolate and extract one component at a time. This has significant consequences for the application. The application needs to implement ways for its different parts to communicate and share information. By using the AWS services, the quality of the application only gets better. Now entire components can fail and the app itself will remain functional, or at least responsive.

Simply Small

AWS has many useful and necessary tools to help you design for failure. You can assign Elastic IP addresses to an instance, so if the instance dies or you replace it, you reassign the Elastic IP address. You can also use Elastic Block Store (EBS) volumes for instance storage. With EBS, you can “carry around” your disks from instance to instance. By making regular *snapshots* of the EBS volumes, you have an easy way to back up your data. An instance is launched from an *image*, a read-only copy of the initial state of your instance. For example, you can create an image containing the Ubuntu operating system with Apache web server, PHP, and your web application installed. And a boot script can automatically attach volumes and assign IP addresses. Using these tools will allow you to instantly launch a fresh copy of your application within minutes.

Most applications start with some sort of database, and the most popular database is MySQL. The AWS RDS offers MySQL as a service. RDS offers numerous advantages like backup/restore and scalability. The advantages it brings are significant. If you don't use this service, make sure you have an extremely good reason not to. Scaling a relational database is notoriously hard, as is making it resilient to failure. With RDS, you can start small, and if your traffic grows you can scale up the database as an immediate solution. That gives you time to implement optimizations to get the most out of the database, after which you can scale it down again. This is simple and convenient: priceless. The command-line tools make it easy to launch a very powerful database:

```
$ rds-create-db-instance kultzter \  
  --db-instance-class db.m1.small \  
  --engine MySQL5.1 \  
  --allocated-storage 5 \  
  --db-security-groups default \  
  --master-user-password Sdg_5hh \  
  --master-username arjan \  
  --backup-retention-period 2
```

Having the freedom to fail (occasionally, of course) also offers another opportunity: you can start searching for the boundaries of the application's performance. Experiencing difficulties because of increasing traffic helps you get to know the different components and optimize them. If you limit yourself in infrastructure assets, you are forced to optimize to get the most out of your infrastructure. Because the infrastructure is not so big yet, it is easier to understand and identify the problem, making it easier to improve. Also, use your freedom to play around. Stop your instance or scale your RDS instance. Learn the behavior of the tools and technologies you are deploying. This approach will pay back later on, when your app gets critical and you need more resources to do the work.

One straightforward way to optimize your infrastructure is to offload the "dumb" tasks. Most modern frameworks have facilities for working with media or static subdomains. The idea is that you can use extremely fast web servers or caches to serve out this static content. The actual dynamics are taken care of by a web server like Apache, for example. We are fortunate to be able to use CloudFront. Put your static assets in an S3 *bucket* and expose them using a CloudFront *distribution*. The advantage is that you are using a full-featured content delivery network with edge locations all over the world. But you have to take into account that a CDN caches aggressively, so change will take some time to propagate. You can solve this by implementing invalidation, building in some sort of versioning on your assets, or just having a bit of patience.

Growing Up

The initial setup is static. But later on, when traffic or load is picking up, you need to start implementing an infrastructure that can scale. With AWS, the biggest advantage you have is that you can create an elastic infrastructure, one that scales up and down depending on demand. Though this is a feature many people want, and some even

expect out of the box, it is not applicable to all parts of your infrastructure. A relational database, for example, does not easily scale up and down automatically. Work that can be distributed to identical and independent instances is extremely well suited to an elastic setup. Luckily, web traffic fits this pattern, especially when you have a lot of it.

Let's start with the hard parts of our infrastructure. First is the relational database. We started out with an RDS instance, which we said is easily scalable. It is, but, unaided, you will reach its limits relatively quickly. Relational data needs assistance to be fast when the load gets high. The obvious choice for optimization is caching, for which there are solutions like Memcached. But RDS is priceless if you want to scale. With minimum downtime, you can scale from what you have to something larger (or smaller):

```
$ rds-modify-db-instance kulitzer \  
--db-instance-class db.m1.xlarge \  
--apply-immediately
```

We have a strategy to get the most out of a MySQL-based data store, so now it is time to set up an elastic fleet of EC2 instances, scaling up and down on demand. AWS has two services designed to take most of the work out of your hands:

- Amazon ELB
- Amazon Auto Scaling

ELB is, for practical reasons, infinitely scalable, and works closely with EC2. It balances the load by distributing it to all the instances behind the load balancer. The introduction of *sticky sessions* (sending all requests from a client session to the same server) is recent, but with that added, ELB is feature-complete. With Auto Scaling, you can set up an autoscaling *group* to manage a certain group of instances. The autoscaling group launches and terminates instances depending on triggers, for example on percentage of CPU utilization. You can also set up the autoscaling group to add and remove these instances from the load balancer. All you need is an image that launches into an instance that can independently handle traffic it gets from the load balancer.

ELB's scalability comes at a cost. The management overhead of this scaling adds latency to the transactions. But in the end, human labor is more expensive, and client performance does not necessarily need ultra low latencies in most cases. Using ELB and Auto Scaling has many advantages, but if necessary, you can build your own load balancers and autoscaling mechanism. All the AWS services are exposed as APIs. You can write a daemon that uses CloudWatch to implement triggers that launch/terminate instances.

Moving Out

The most expensive part of the infrastructure is the relational database component. None of the assets involved here scales easily, let alone automatically. The most expensive operation is the join. We already minimized the use of joins by caching objects,

but that is not enough. All the big boys and girls try to get rid of their joins altogether. Google has BigTable and Amazon has SimpleDB, both of which are part of what is now known as NoSQL. Other examples of NoSQL databases are MongoDB and Cassandra, and they have the same underlying principle of not joining.

The most radical form of minimizing the use of joins is to decouple, and a great way to decouple is to use queues. Two applications performing subtasks previously performed by one application are likely to need less severe joins. Internally, Amazon has implemented an effective organization principle to enforce this behavior. Amazon reorganized along the lines of the functional components. Teams are responsible for everything concerning their particular applications. These decoupled applications communicate using Amazon SQS and Amazon Simple Notification Service (SNS), and they share using Amazon SimpleDB and Amazon S3.

These teams probably use MySQL and Memcached and ELB to build their applications. But the size of each component is reduced, and the traffic/load on each is now manageable once more. This pattern can be repeated again and again, of course.

“You Build It, You Run It”

Perhaps by chance, but probably by design, AWS empowers development teams to become truly agile. It does this in two ways:

- Getting rid of the long-term aspect of the application infrastructure (investment)
- Building tools to help overcome the short-term aspect of operating the application infrastructure (failure)

There is no need to distinguish between building and running, and according to Werner Vogels (<http://queue.acm.org/detail.cfm?id=1142065>), it is much better than that:

Giving developers operational responsibilities has greatly enhanced the quality of the services, both from a customer and a technology point of view. The traditional model is that you take your software to the wall that separates development and operations, and throw it over and then forget about it. Not at Amazon. You build it, you run it. This brings developers into contact with the day-to-day operation of their software. It also brings them into day-to-day contact with the customer. This customer feedback loop is essential for improving the quality of the service.

This lesson is interesting, but this particular change in an organization is not always easy to implement. It helped that Vogels was the boss, though it must have cost him many hours, days, and weeks to convince his colleagues. If you are not the boss, it is even more difficult, though not impossible. As we have seen before, AWS offers ways to be agile with infrastructures. You can tear down servers, launch new ones, reinstall software, and undo entire server upgrades, all in moments.

Individuals and Interactions: One Team

In bigger organizations, there is an IT department. Communication between the organization and its IT department can be difficult or even entirely lacking. The whole activity of operating applications can be surrounded with frustration, and everyone feels powerless. Smaller companies often have a hosting provider, which can be very similar to an IT department. A hosting provider tends to be a bit better than an IT department because you can always threaten to replace it. But the lock-in is significant enough to ignore these issues; for a small company it is generally more important to focus on development than to spend time and energy on switching hosting providers.

Let's start with one side: the IT department or hosting provider. Its responsibility is often enormous. IT department members have to make decisions on long-term investments with pricetags that exceed most product development budgets. These investments can become difficult projects with a huge impact on users. At the same time, the IT department has to make sure everything runs fine 24/7. It is in a continuous battle between dealing with ultra long term and ultra short term; there seems to be nothing in between.

Now for the development team. The work of the development team is exactly in between the long term and the short term. The team is asked to deliver in terms of weeks and months, and often makes changes in terms of days. During the development and testing phases, bugs and other problems are part of the process, part of the team's life. But once in production, the application is out of the team's hands, whether they like it or not.

Organizations can handle these dynamics by creating complex processes and tools. Because each group typically has no understanding of the other's responsibilities, they tend to formalize the collaboration/communication between the teams, making it impersonal. But as the Agile Manifesto states, in developing software *individuals and interactions* are more valuable than processes and tools. With AWS, the investment part of infrastructures is nonexistent. And AWS helps you manage the ultra short term by providing the tools to recover from failure. With AWS, you can *merge the responsibility of running the application with the responsibility of building it*. And by doing this, you turn the focus on the people and their interactions instead of on creating impersonal and bureaucratic processes.

Working Software: Shared Responsibility

Deploying software means moving the application from development to the "other side," called *production*. Of course, the other side—the IT department in the traditional structure—has already committed to a particular SLA. As soon as the application is moved, the IT department is on its own. As a consequence, members want or need to know everything necessary to run the application, and they require documentation to do so.

This documentation is an SLA itself. If there is a problem related to the software that is not included in the documentation, fingers will point to the development team. The documentation becomes a full description of every aspect of the application, for fear of liability.

But in the end, there is only one thing that matters, and that is whether the application is running. This is not very difficult to determine if the responsibility is shared; the team members will quickly discuss a solution instead of discussing who is to blame. So the thing to do is to build working software together, as a team. Remove the SLAs and merge the functions of two teams into one. When something doesn't work, it needs to be fixed—it does not always have to be debated first. Documentation in this context becomes less important as a contract between parts, and becomes an aid to keep the application running.

Customer Collaboration: Evolve Your Infrastructure

Wherever IT is present, there is an SLA. The SLA is regarded as a tool in managing the process of IT infrastructure, where the bottom line is the number of nines. In reality it is a tool designed to facilitate cooperation, but is often misused for the purpose of deciding who is responsible for problems: development or operations.

It can be difficult to negotiate this contract at the time of application development. There is a huge difference between “we need to store audio clips for thousands of customers” and “storage requirements are estimated to grow exponentially from 500 GB to 5 TB in three years.” The problem is not so much technical as it is that expectations (dreams, often) are turned into contract clauses.

You can change contract negotiation into *customer collaboration*. All you need to do is merge the two responsibilities: building and running the application becomes a shared challenge, and success is the result of a shared effort. Of course, in this particular example it helps to have Amazon S3, but the point is that requirements change, and collaboration with the customer is better suited for handling those changes than complex contract negotiations.

Responding to Change: Saying Yes with a Smile

At the end of the project, just two weeks before launch, the CEO is shown a sneak preview of the new audio clip platform. She is very excited, and proud of the team effort. The meeting is positive and she is reassured everything is planned for. Even if the dreams of millions of customers come true, the platform will not succumb to its success because it's ready to handle a huge number of users.

In the evening, she is telling her boyfriend about her day. She shares her excitement and they both start to anticipate how they would use the new platform themselves. At some point he says, “Wouldn't it be great to have the same platform for video clips?”

Of course, he doesn't know that this whole project was based on a precondition of audio-only; neither does the CEO.

In the morning, she calls the project manager and explains her idea. She is still full of energy and says enthusiastically, "the functionality is 100% perfect, only we want audio *and* video." The project manager knows about the precondition, and he also knows that video files are significantly bigger than audio files. However, the CEO doesn't want to hear buts and objections about moving away from the plan; she wants this product to change before launch.

In Short

In this chapter we walked you through the recent history of AWS. We showed how each of the AWS services was created to solve a particular problem with Amazon's platform. We gave you a brief overview of the different AWS services you can use to build, monitor, and scale your cloud infrastructure. And finally, we talked about how developing with AWS is naturally agile and allows you to make the infrastructure building and running part of the development process.

In the rest of the book, we'll show how all these services actually work, so get ready to stop reading and start doing! In the next chapter, we will start with migrating a simple web application to AWS using EC2, RDS, S3, and CloudFront.

Growing with S3, ELB, Auto Scaling, and RDS

We have done quite a lot in just a couple of chapters. We have explored what it means to design, build, and operate virtual infrastructures on AWS. We have looked at the opportunities it provides, and we have moved a real-world application, Kulitzer.com, to AWS. Although we have done many things that are usually very difficult on physical infrastructures, we have not yet looked at the biggest benefit of AWS: an *elastic* infrastructure that scales with demand.

With growing traffic, our initial setup will soon be insufficient. We know we can *scale up* with bigger machines, but we prefer to *scale out*. Scaling up is OK in certain situations. If you have a fixed upper limit in traffic—for example, if your users are internal to an organization—you can safely assume it is not necessary to go through the trouble of implementing load balancing and/or autoscaling. But if your application is public, and perhaps global, this assumption can be costly. Scaling up is also manual on AWS, requiring reboots and temporary unavailability of your instance. The same goes for the opposite, scaling down (Figure 3-1).

Scaling out, on the other hand, requires changes to your application. You have to make sure you can have several instances doing the same work, preferably on their own. If the instances can operate in their own context, it doesn't matter if you have three or seven instances. If this is possible, we can handle traffic with just the right number of instances. If the traffic increases, we launch some more instances; if the traffic decreases, we terminate some of the instances.

We don't have to worry about scaling out the presentation layer (recall the three-tier architecture introduced in the previous chapter). We use S3/CloudFront, which does all the scaling out and scaling in for us. And even if you chose to implement your own infrastructure for this, it is not that hard. Making the logic layer elastic is a bit harder, but not impossible. You basically have to *make sure all of the data the application server works on that needs to be shared by other application servers does not reside on the application server itself*. Think of sessions, for example, or user content. But the data

layer is more problematic. For traditional Relational Database Management Systems (RDBMS), you will probably require a strategy of scaling up *and* scaling out. But introducing elasticity is almost impossible in this situation.

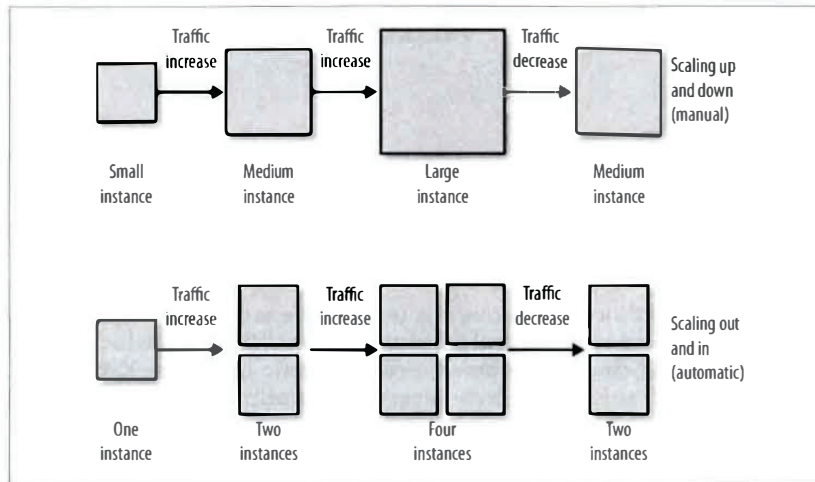


Figure 3-1. Scaling up versus scaling out

Preparing to Scale

Scaling an application on AWS is essentially the same as scaling any other infrastructure. As said before, if you use S3/CloudFront as your presentation tier, you don't have to worry about that. For the logic tier (Apache, in our case), you have to move out the data/information to shared storage. If you chose to implement your own presentation layer with thin HTTP servers like Nginx or Lighttpd, you can follow the same principle.

You have several options to move *filesystem-style data* to a shared storage solution (Table 3-1). Remember that an EBS volume can only be attached to one instance at a time, so it is not an option. The obvious choice if you don't have a Storage Area Network (SAN)/Network Attached Storage (NAS) at your disposal used to be to implement something like Network File System (NFS) or Server Message Block (SMB), but they are not very scalable. An alternative is to move this data to a database, relational or otherwise. In both cases, you don't really solve the scalability problem, you just move it to another part of your infrastructure. Most of the time, the shared data you want to store on the filesystem is written once and read often (or never). For this use case, S3 is ideal, if you can live with *eventual consistency*. In other words, it might take a little while before your update on an object becomes visible for reading.

You can store the *nonfilesystem-style data* somewhere else. This is session information or other data that needs to be shared by several instances, for example. In the next chapter we will look at AWS solutions like SimpleDB and SQS. For now, we'll just move it to the database.

Then we are left with the *database*. As said before, the problem with a relational database when trying to scale is the join, or join-like operations, which are the most expensive ones. Of course, you can handle many joins if you replicate the database and perform these operations on one or more slaves, but the joins stay—they are not removed. The most radical way to scale the database without getting rid of joins is to minimize their execution. The easiest and most versatile way is caching. The popular choice for caching is Memcached.

Modern web application development frameworks make it very easy to utilize these techniques. Rails is not any different, as you'll see. Figure 3-2 shows a high-level view of the different components of Kulitzer, our Rails sample application.

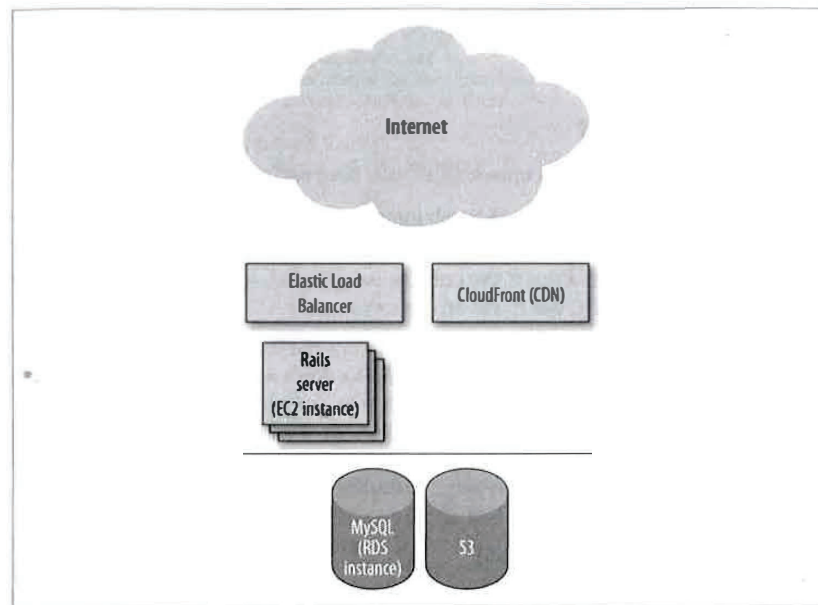


Figure 3-2. Kulitzer architecture

You will also see that RDS, the MySQL service provided by Amazon, can easily be scaled up and down if needed (up to a certain limit, of course).

Table 3-1. Preparing to scale: options for sharing data

Presentation tier	Filesystem-style data	Nonfilesystem-style data
CDN, e.g., CloudFront over S3	S3	Database
	Database	SimpleDB
	SAN, NAS	SQS
	NFS, SMB (not very scalable)	

Setting Up the Tools

Before we start diving into these scaling techniques, it is a good idea to install some tools. Download the command-line tools for the following:

- ELB (<http://aws.amazon.com/developertools/2536>)
- Auto Scaling (<http://aws.amazon.com/developertools/2535>)
- CloudWatch (<http://aws.amazon.com/developertools/2534>)

Extract the compressed files and add the bin directories to your `initaws` file as we did in Chapter 2.

S3 for File Uploads

When you want to store user content to S3, you have two aspects to consider:

- *Latency.* S3 is accessed through a web interface, which is considerably slower than disk (EBS).
- *Eventual consistency.* Even if the content is uploaded, it is not guaranteed to be available immediately, but it will be there eventually.

For these reasons, we need to find a way to accept the user content but offload the actual upload to S3 to a separate process/thread. We don't want to keep the user waiting for this work to complete. She will be notified that it might take a while and will automatically see the content when available. For big workloads, you might consider Amazon SQS, but that is a bit too much at this moment. We'll search for our solution in the framework we use, Ruby on Rails, in the case of the Kulitzer example to follow.

User Uploads for Kulitzer (Rails)

We chose to use Rick Olson's `attachment_fu` (http://github.com/technoweenie/attachment_fu) for handling user uploads. It does S3 out of the box, so we don't need to modify the plug-in to implement the actual upload. But it doesn't do this outside of the user thread, so we need to do some extra work there. Of course, we are not the only ones with this challenge, and we found that people were using a plug-in called `8j` (`BackgroundJob`) in combination with `attachment_fu` to handle S3 file uploads. Ara

Howard's Bj uses the database to create jobs, which are run one at a time, outside of the app server's process.

So with this, the user content is uploaded to S3. The location (URL) given to the actual assets is the S3 location. But we want to serve this content from CloudFront. We know all our images come from the same CloudFront distribution, so we can implement this quickly by overriding the `image_tag` method of the `AssetTagHelper` in `config/initializers/image_tag.rb`. In this method, the S3 URL is replaced with a URL from one of the domains (randomly chosen) that points to the CloudFront distribution. It might not be the most beautiful solution, but it is easy to understand and it works. If we need something more elaborate, we will probably have to move from Bj to Amazon SQS:

```
module Kulitzer
  module CloudFrontImageTag

    S3_PREFIX = 'http://s3.amazonaws.com/media.kulitzer.com'
    CF_PREFIX = 'http://media%d.kulitzer.com'

    def image_tag(source, options = {})
      # For production, re-write url to use CloudFront
      if Rails.env.include?('production') && source.starts_with?(S3_PREFIX)
        # re-write to use a cloudfront url eg
        # http://s3.amazonaws.com/media.kulitzer.com/foo.jpg becomes
        # http://mediaZ.kulitzer.com/foo.jpg where Z = 0..3

        cf_host = CF_PREFIX % rand(4)
        source = source.sub(S3_PREFIX, cf_host)
      end

      super
    end
  end
end

ActionView::Base.send :include, Kulitzer::CloudFrontImageTag
```

Elastic Load Balancing

Wikipedia says that *load balancing* is “a technique to distribute workload evenly across two or more computers, network links, CPUs, hard drives, or other resources, in order to get optimal resource utilization, maximize throughput, minimize response time, and avoid overload.” And, Wikipedia notes, it can also “increase reliability through redundancy.” To finish the description, Wikipedia says load balancing “is usually provided by a dedicated program or hardware device.”

Amazon *Elastic Load Balancing* (ELB) is all of the above. It distributes load evenly across availability zones and across instances within those zones. ELB checks the health of the instances and will not route traffic to unhealthy instances. You have the ability to use something called *sticky sessions*, which can force a particular session to one instance.

You would need this in case the instances keep session data in a nonshared location such as local memory.

ELB is not a dedicated program or a hardware device; it is a load-balancing service. As a service, it can automatically scale its capacity depending on incoming traffic. As a result, an ELB is not referenced by an IP address, but by a fully qualified domain name. It's been said that an ELB scales best with slowly increasing/decreasing traffic, but in our experience spikes are handled quite well.



Layar is a high-profile augmented reality (AR) application for smartphones like Android and iPhone. The ongoing traffic is reasonable, but not extreme; there are just not very many AR-capable smartphones available yet (with camera, GPS, compass, and accelerometer). However, the Layar application is magic, and phone manufacturers like Samsung and Motorola use Layar to sell their latest devices.

Motorola especially is determined to sell as many Android smartphones as possible, and it often buys primetime television advertising slots. One of its ads focuses solely on Layar, and it was aired during Oprah Winfrey's popular TV show. We saw a spike in the Layar servers' traffic that was extreme, to say the least. We didn't have autoscaling yet, so our two instances were hit hard. We went from around 30 requests per second to over 800 requests per second. Any more was just not possible.

This was the first serious production test of the Layar infrastructure. It was a valuable lesson, especially with the other, even bigger events on the horizon.

Now that we are using S3 for file uploads, we are almost ready for load balancing, as we can't rely on instance storage anymore. There is one other thing we have to make sure we don't do, and that is use local files. Rails can store sessions in file storage. The default storage mechanism for sessions in Rails 2.x is CookieStore, and this is fine because it stores the session object in the client. Same for the alternative ActiveRecordStore (or "one of its derivatives," as the guide tells us) that we use for Kulitzer, which stores the session in the database.

Creating an ELB

Creating an ELB is straightforward, and you can use either the AWS Console or the command-line tools. Simply give the ELB a name (which will be part of the domain name you use for the ELB), tell it how to handle traffic (protocols to handle, such as HTTP, HTTPS, TCP, and how to route load balancer ports to other or the same ports in the instances), configure the health check (thresholds to decide if an instance is healthy or not), and add your zones and instances. As you are used to by now, the Console gives you a wizard to create an ELB (Figure 3-3).

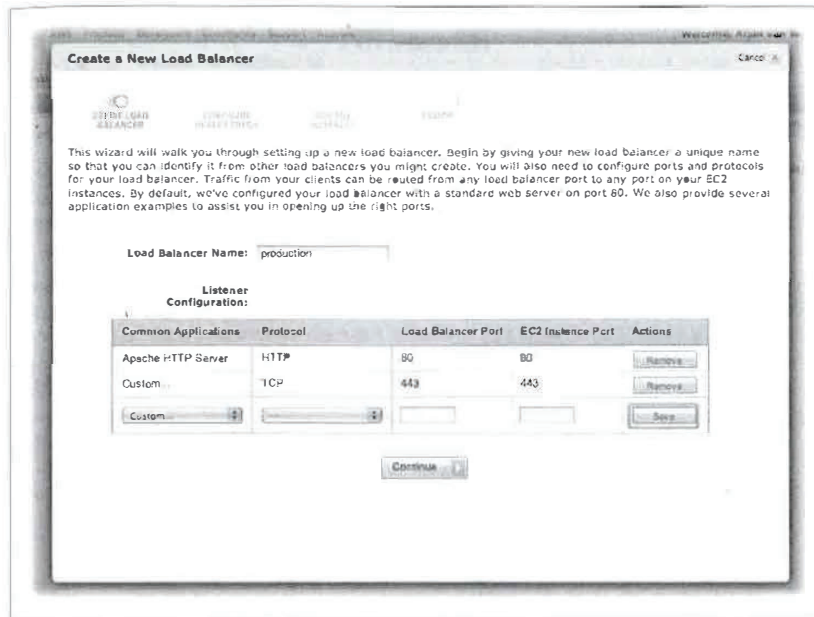


Figure 3-3. Create an Elastic Load Balancer

You can also create the ELB with the following commands:

```
$ elb-create-lb production \
  --availability-zones us-east-1b \
  --listener "protocol=HTTP, lb-port=80, instance-port=80" \
  --listener "protocol=TCP, lb-port=443, instance-port=443"

$ elb-configure-healthcheck production \
  --target "HTTP:80/" \
  --interval 30 \
  --timeout 2 \
  --healthy-threshold 6 \
  --unhealthy-threshold 2

$ elb-register-instances-with-lb production \
  --instances i-29184c43
```

ELB and HTTPS

This particular load balancer routes HTTP traffic. It does not use the HTTPS feature of ELB, it just routes port 443 TCP traffic. This way, we handle the HTTPS on the instances.

You can also have the load balancer handle HTTPS traffic, but the communication between ELB and instances then remains unencrypted HTTP. You can't secure this 100% yet through the use of something similar to security groups, but if you make use of this, it does give you a couple of advantages, including the following:

- Extra load of HTTPS traffic is on the ELB, not on the instance. And the load of the ELB is taken care of as part of the service; you don't need to pay more for this.
- It is easier to set up and maintain.
- You can have session stickiness on HTTPS sessions.

Setting this up requires some IAM command magic to add your certificates, and a slightly different invocation of the ELB commands. Here we add two different wildcard certificates: **.publitas.nl* and **.publitas.com*. We pass the private key, public key certificate, the certificate chain file, and the name of the certificate to `iam-servercertupload`. Then we create the two load balancers from the command line:

```
$ iam-servercertupload -k _publitas.nl.key \
  -b _publitas.nl.crt -c gd_bundle.crt -s _publitas_nl
$ iam-servercertupload -k _publitas.com.key \
  -b _publitas.com.crt -c gd_bundle.crt -s _publitas_com

$ iam-servercertlistbypath
arn:aws:iam::075493990647:server-certificate/_publitas_com
arn:aws:iam::075493990647:server-certificate/_publitas_nl

$ elb-create-lb publitas-nl \
  --availability-zones eu-west-1a,eu-west-1b \
  --listener "protocol=HTTP, lb-port=80, instance-port=80" \
  --listener "protocol=HTTPS, lb-port=443, instance-port=80,
    cert-id=arn:aws:iam::075493990647:server-certificate/_publitas_nl"

$ elb-create-lb publitas-com \
  --availability-zones eu-west-1a,eu-west-1b \
  --listener "protocol=HTTP, lb-port=80, instance-port=80" \
  --listener "protocol=HTTPS, lb-port=443, instance-port=80,
    cert-id=arn:aws:iam::075493990647:server-certificate/_publitas_com"
```

When you create the load balancer from the command line, you need to specify the availability zones where your instances will run. In the AWS Console, the availability zones of the instances you choose will be added automatically. If you don't choose any instances, one availability zone will be chosen for you by default.



If you work with ELBs from the Console, it appears as if you can only register an instance to one load balancer. It might be a mistake in the Console, but you can register instances to multiple load balancers at the same time. You can do this explicitly through the command line, or have Auto Scaling register multiple ELBs at the same time.

If you want to serve multiple SSL domains (actually, if you need multiple SSL certificates) through ELBs, you can't use the HTTPS feature of the load balancer. You can set up an ELB for every certificate, all pointing to the same instances.

You can clone your existing instance and make it work in a different availability zone. If you add this new instance to the load balancer, you have a nice load-balanced server farm for your application. This infrastructure is very close to our traditional infrastructures. There are many solutions to the problem of deploying an application on several servers; we use Capistrano in this case. With a little bit of work, we can scale our application up and down.



Because ELB does support load balancing on the HTTP level and TCP level, you are not restricted to balancing your web traffic. The default listeners are all web or application servers, but you can also use an ELB to set up a server farm for IMAP or POP, for example. All the principles we discussed still apply, of course.

Difficulties with ELB

Currently, there are a number of challenges with the ELB service. One of the most irritating, but logical, is that DNS does not allow CNAME records for root domains. In other words, you cannot point `kulitzer.com` to `production-1188255634.us-east-1.elb.amazonaws.com`. Some people think the only good domain name for an app is the shortest one possible, in our case `kulitzer.com`. But the hosts of most large sites rewrite their domains to the `www` alternative, probably because a service scales much easier than a device. You can use the rewrite, or forwarding, feature of your domain name registrar. We often use one of the app servers with an EIP address to do this rewriting. With Apache, you can add such a rewrite with `mod_rewrite`. We rewrite every domain name we don't want to `www.kulitzer.com` like this:

```
RewriteEngine on
RewriteCond %{HTTP_HOST} !^www\.kulitzer\.com [NC]
RewriteCond %{HTTP_HOST} !^$
RewriteRule ^/(.*) http://www.kulitzer.com/$1 [L,R]
```

Changing the configuration of a running ELB is limited. You can't add or remove different connections, for example. For this reason, just always do HTTP and HTTPS, if you have the proper certificates. The chances are small that you will want to balance other traffic later in the life of a load balancer, but a mistake with HTTP is inconvenient.

Auto Scaling

The cloud's biggest promise is elasticity: running with just the amount of computing resources your workload demands at any given time. Of course, services like SimpleDB, SQS, and CloudFront scale automatically. But for some applications, you need to be able to dynamically scale EC2 capacity, or instances. And that is what Auto Scaling does. It allows you to define conditions for increasing or decreasing the number of instances.

According to AWS, "Auto Scaling is particularly well suited for applications that experience hourly, daily, or weekly variability in usage." This is exactly what we need for Kulitzer. At peak hours, we want to increase our capacity to handle the load, and when it's not so busy, we want to run idle.

You can also use Auto Scaling to make your application more *resilient*. Say you use Auto Scaling to monitor one instance; if the instance dies, Auto Scaling can launch a new one automatically. This same technique can be applied to multiple instances distributed over different availability zones. We basically apply elasticity in case of failure, automatically resizing the group to its required size.

Auto Scaling is designed to work on its own, but can be easily integrated with an ELB. You can't use metrics of the load balancer as conditions for scaling. But it is convenient that the autoscaling group handles registration of instances with the load balancer.

Usually, it takes several minutes for an instance to launch and provision itself. Because demand cannot be met immediately, extreme spikes are more difficult to handle than gradual changes in demand. But you can tune the triggers for scaling so that spiky conditions can be handled, more or less.



After Oprah introduced Layar, we knew we should plan for these spikes. We set up autoscaling and tuned the triggers so that we would scale up relatively early and scale down relatively late.

Not everyone at Layar expected another Oprah, and we were already quite surprised to have had to handle that spike. But in May 2010, the inevitable happened. The same ad was aired in primetime—not on a talk show, but during the fifth game of the NBA finals.

Within an hour, we scaled from the two large instances we normally run to 16 large instances. The database choked (again) and we couldn't handle any more traffic. But compared to our Oprah moment, we were able to serve many more users for just a few dollars.

Setting Up Auto Scaling

Auto Scaling is a little complex. There are *autoscaling groups* that hold the instances. You have *launch configurations* that determine which instance is launched. And there

are *alarms*, borrowed from CloudWatch, that determine when a scaling action—called a *policy*—should take place. Policies simply specify that instances should be launched or terminated. All of these are dependent, making it difficult to know the order in which they can or should be changed. Working on a live Auto Scaling setup can be a bit scary. Don't be afraid, though—all Auto Scaling commands are quite verbose (not all EC2 commands are this “talkative”) and it is difficult to do things you don't intend. But there is nothing better than testing a properly set up autoscaling group by terminating an instance and seeing a new instance taking its place.

Launch configuration

A launch configuration determines what kind of instance is being launched. It sets the AMI and the type of instance that is going to be created. Here we immediately run into the most important change in our application. We need to create self-provisioning instances; upon launch, an instance needs to have or get the right version of the application.

As you don't want to change images all the time, you probably want to devise a way of passing enough information to the instance for provisioning. Depending on your way of deploying an application, you can specify the branch (Subversion) or commit (GIT) in a place where the instance can read it. You can choose to store this in a public object in S3, as the rest of the information is safely stored on the image itself. If you want even more freedom, you can specify a complete provisioning script in S3 that is executed by the instance upon startup.

With your image ready, specify the launch configuration by executing the following command:

```
$ as-create-launch-config app-server-launch-config-1 \
  --image-id ami-6e1deb07 \
  --instance-type c1.medium \
  --group web
```

Autoscaling group

An autoscaling group must have a launch configuration, which is why you created one first. Now that we have one, we can create an autoscaling group. Depending on how you specify the autoscaling group, you might be launching instances immediately.

For Kulitzer, the autoscaling group will control between two and six instances behind an ELB. We want at least two instances to add redundancy, but the database cannot handle more than six instances at this moment. The instances will be distributed over two availability zones to make our application more resilient. Remember that availability zones are physically separated, forcing instances to use different cooling, racks, and blade servers. And we force the autoscaling group to wait a bit after scaling (out or in) by using the `cooldown` option:

```
$ as-create-auto-scaling-group app-server-as-group-1 \
  --launch-configuration app-server-launch-config-1 \
```

```
--availability-zones us-east-1c,us-east-1d \
--min-size 2 \
--max-size 6 \
--default-cooldown 120 \
--load-balancers production
```



This group will immediately launch two instances. If you don't want that because you want to specify your triggers first, you can set `--min-size` to 0 and change it to the required size later. If your group does not have running instances, it is impossible to initiate scaling activities. The first scaling activity has to be manual in that case.

With the configuration done so far, this group will not really scale automatically, but it will make sure you keep the minimum you specified. You can test this by terminating one of the instances.

Sometimes it might happen that the request to start a new instance cannot be met, for example due to capacity problems. If you specified multiple availability zones in the autoscaling group, it will launch an instance in another availability zone. The autoscaling group will later try to rebalance the instances evenly over the availability zones. If your autoscaling group is hooked up with an ELB, instances will always be removed from the load balancer before being terminated. This is to ensure that the load balancer's service is not degraded.

Autoscaling (alarms and policies)

To make this autoscaling group really scale automatically, we need a way to monitor the group and execute a scaling policy. This used to be done with triggers (part of the Auto Scaling API), but it has changed to policies. The actual triggering has moved to CloudWatch, where metric alarms can execute a policy. The alarms determine when to take action, the actions are specified in policies, the autoscaling group defines the boundaries (min, max, availability zones), and the launch configuration defines which instances are created.

In the Kulitzer example, we will create two different policies: one for scaling up and one for scaling down. Each will be triggered by a different alarm, also shown here:

```
$ as-put-scaling-policy app-server-scale-UP-on-CPU \
  --auto-scaling-group app-server-as-group-1 \
  --type ChangeInCapacity \
  --adjustment 2 \
  --cooldown 300

$ mon-put-metric-alarm alarm-app-server-scale-UP \
  --alarm-actions arn:aws:autoscaling:us-east-1:205414005158:scalingPolicy:...
  --metric-name CPUUtilization \
  --unit Percent \
  --namespace AWS/EC2 \
  --statistic Average \
```

```

--dimensions="AutoScalingGroupName=app-server-as-group-1" \
--period 60 \
--evaluation-periods 2 \
--threshold 60 \
--comparison-operator GreaterThanThreshold

$ as-put-scaling-policy app-server-scale-DOWN-on-CPU \
--auto-scaling-group app-server-as-group-1 \
--type ChangeInCapacity \
--adjustment=-2

$ mon-put-metric-alarm alarm-app-server-scale-UP \
--alarm-actions arn:aws:autoscaling:us-east-1:205414005158:scalingPolicy:...
--metric-name CPUUtilization \
--unit Percent \
--namespace AWS/EC2 \
--statistic Average \
--dimensions="AutoScalingGroupName=app-server-as-group-1" \
--period 60 \
--evaluation-periods 5 \
--threshold 20 \
--comparison-operator LessThanThreshold

```



It is important for the number of instances you scale up or down to be a multiple of the number of availability zones your autoscaling group utilizes in case your autoscaling group is connected to an ELB. The load balancer will distribute the traffic evenly over the active zones.

When we create the alarm, we pass the policy identifier returned by the `as-put-scaling-policy` command. There are different metrics we can check; in this case, we scale up when the `CPUUtilization` is more than 60% on average over all the instances of the given image for two consecutive periods of one minute each. Similarly, for scaling down, we check that the `CPUUtilization` on average is less than 20% for five consecutive periods of one minute.

Policies and metric alarms are very important for operating the autoscaling group. It is important to understand your application in order to choose which measure to use. You can tune the alarms based on your needs and requirements.

You can't make assumptions about which instances are terminated when the autoscaling group scales down. But if you scale regularly, the chances are good that the turnover of the physical hardware is quite high. This makes your application much more robust, because the chances of hardware failure increase with the age of deployment.

The steps to configure autoscaling, then, are the following:

1. Create a launch configuration, specifying the AMI and type of instances you want.
2. Create an autoscaling group, passing the above launch configuration, the availability zone(s), the minimum and maximum number of instances, and load balancer(s), if any.

3. Create policies that represent scaling actions to take, like launching new instances or terminating existing ones.
4. Create alarms that trigger the execution of the above policies under certain conditions.

Semiautoscaling

Sometimes you don't want to scale based on the state of the group itself, but on a particular day and time. This might at first seem inefficient, because it's not directly triggered by a change in load. But if you know the door to the shop opens at 09:00 and you're selling the iPhone 7, you do want to preheat your server farm.

Instead of scaling up and down based on CPU utilization, we want to make sure we have 24 instances running between 08:30 and 18:00 until 23 December 2011. On Christmas Eve, everyone is doing something else, so we can terminate this special group. We can do that like this:

```
$ as-put-scheduled-update-group-action christmas-app-server-scale-UP \  
  --auto-scaling-group app-server-as-group-1 \  
  --desired-capacity 24 \  
  --time 2010-12-23T08:30:00Z  
  
$ as-put-scheduled-update-group-action \  
  christmas-app-server-scale-DOWN-on-christmas-eve \  
  --auto-scaling-group app-server-as-group-1 \  
  --desired-capacity 0 \  
  --time 2010-12-24T20:00:00Z
```

Auto Scaling in Production

AWS Auto Scaling is great, and when used in combination with ELB it is a powerful tool in bringing down your costs by optimizing resource usage. In production, however, it is a bit difficult to handle. You can perform most common tasks easily if you know how, and all destructive activities are protected by warnings and dialog-style questions. The change we most often have to apply is changing a launch configuration because the image has changed.

Pausing Auto Scaling

When working on the autoscaling groups or instances in the group, autoscaling activities are unwelcome guests. In these cases, you can instruct AWS to leave your autoscaling group alone. They call this *suspend* and *resume*, and you can do it like this:

```
$ as-suspend-processes app-server-as-group-1  
$ as-resume-processes app-server-as-group-1
```

Replacing the launch configuration

If the image has changed for any reason, you will want to replace the launch configuration. The launch configuration only states what is to be done in the future, so it doesn't change anything that is currently in production. We always rotate instances when changing the launch configuration:

```
# first create a new launch configuration, then change the auto scaling group
$ as-create-launch-config app-server-launch-config-2 \
  --image-id ami-c503e8ac \
  --instance-type c1.medium \
  --group web
$ as-update-auto-scaling-group app-server-as-group-1 \
  --launch-configuration app-server-launch-config-2

# now replace the instances gracefully
$ as-terminate-instance-in-auto-scaling-group i-433e1829 -D

# and if the new instances is 'In Service' terminate the other
$ as-terminate-instance-in-auto-scaling-group i-413e182b -D
```

-D indicates that we don't want to decrement the capacity of the autoscaling group by 1. The result is that a new instance will be launched to compensate for each terminated instance. If we wanted instead to decrement the capacity of the group, we would use the -d option, and a new instance would be launched. This is not, of course, what we want in this example.

Unfortunately, replacing instances this way terminates the old instance before registering the new one. This means you are temporarily running with one fewer instance. Make sure you wait until the new instance is registered, healthy, and In Service before terminating the next.

Changing the alarms

The alarms are potentially more dangerous to your production environment. By changing the alarms, you can force your autoscaling group to breach the thresholds and initiate scaling activities. However, it is not very difficult to change the alarms, though it is slightly different from the launch configuration. Instead of first creating a new alarm, you can just update the existing one:

```
$ mon-put-metric-alarm alarm-app-server-scale-UP \
  --period 300 \
  --evaluation-periods 2 \
  --threshold 30 \
  --comparison-operator LessThanThreshold
```

This command may seem a bit long, but you have to specify everything again.

Changing the autoscaling group

If you are technically breaching, but running at minimum or maximum capacity, the autoscaling group does not launch or terminate instances. In our case, two medium instances is too much at idle time. If we decrease the minimum size of the autoscaling group, the remaining two instances will be terminated:

```
$ as-update-auto-scaling-group app-server-as-group-1 \
  --min-size 0
```

Decommissioning an autoscaling group

If you terminate an instance directly, the autoscaling group will probably just launch another one. If you use the command we showed for rotating instances, the same will happen. Set the minimum size of the autoscaling group to 0 and, if necessary, terminate the instances by hand:

```
$ as-update-auto-scaling-group app-server-as-group-1 \
  --min-size 0
$ as-terminate-instance-in-auto-scaling-group i-870620ed -d
$ as-terminate-instance-in-auto-scaling-group i-810620eb -d
$ as-delete-auto-scaling-group app-server-as-group-1
```

Scaling a Relational Database

It's amazing what the AWS team achieved with RDS in a little over a year since it was introduced. The team added the high-availability feature it calls multi-AZ, or multiavailability zone. And after that, it also added *read replicas*. With these new features, in addition to the existing scalability features, RDS is growing into a serious RDBMS service. You can easily scale up while minimizing downtime, and also scale out without too much hassle.

If your app gets to the point that you need to start scaling either up or out, it is a good idea to switch to multi-AZ if you don't run it already. If you have a simple RDS instance, you will degrade your service significantly while scaling, as you can expect to lose the ability to write and/or read. With multi-AZ RDS instances, your service is almost uninterrupted.

Scaling Up (or Down)

Scaling up is so easy it is almost ridiculous. The only drawback is that you have some downtime during the operation. If you don't have multi-AZ enabled, the downtime of your RDS instance could be several minutes, as you have to wait until a new instance is launched and fully functional. For multi-AZ RDS instances, you will experience some downtime as a failover is initiated after the slave has been scaled up (or down). This failover doesn't take more than a minute most of the time.

If you initiate a scaling activity via the Console, make sure you enable Apply Immediately if you are in a hurry. If you don't, scaling will take place during the scheduled maintenance period (Figure 3-4).

The screenshot shows the 'Modify DB Instance: production' dialog box. It contains the following fields and options:

- DB Instance Class:** db.m2.xlarge
- DB Engine Version:** 5.1.50
- Auto Minor Version Upgrade:** Yes (selected)
- Allocated Storage:** 5 GB
- Multi-AZ Deployment:** Yes (selected)
- Backup Retention Period:** 7 days
- Backup Window:** Start Time: 03:00 UTC, Duration: 2 hours
- Maintenance Window:** Start Time: Sunday 04:00 UTC, Duration: 4 hours
- DB Parameter Group:** production
- DB Security Groups:** default, production
- Master User Password:** (empty field)
- Apply Immediately:** ☒

At the bottom, there is a note: 'Note: If you wish to change your DB Instance Class, and/or Backup Retention Period, along with selecting "Apply Immediately", you will need to schedule a maintenance window to perform these changes.' Below the note are 'Cancel' and 'Yes, Modify' buttons.

Figure 3-4. Modify the RDS instance (scaling up)

Scaling using the command-line tools is a two-step process. First scale, and then reboot:

```
$ rds-modify-db-instance production \
  --db-instance-class db.m1.xlarge --apply-immediately
$ rds-reboot-db-instance production
```

DB instance classes

Of course, every service in AWS uses a slightly different naming convention. The equivalent of EC2 Instance Type for RDS is called DB Instance Class. Luckily, the classes themselves are more or less similar to the types in EC2. The smallest possible RDS instance you can have is comparable to a small EC2 instance, for example, though we experience the performance as a bit more consistent with the RDS instance. Here are all the instance classes with their descriptions as AWS advertises them:

Small DB Instance

1.7 GB memory, 1 EC2 Compute Unit (1 virtual core with 1 ECU), 64-bit platform, moderate I/O capacity

Large DB Instance

7.5 GB memory, 4 ECUs (2 virtual cores with 2 ECUs each), 64-bit platform, high I/O capacity

Extra Large DB Instance

15 GB memory, 8 ECUs (4 virtual cores with 2 ECUs each), 64-bit platform, high I/O capacity

High-Memory Extra Large DB Instance

17.1 GB memory, 6.5 ECUs (2 virtual cores with 3.25 ECUs each), 64-bit platform, high I/O capacity

High-Memory Double Extra Large DB Instance

34 GB memory, 13 ECUs (4 virtual cores with 3.25 ECUs each), 64-bit platform, high I/O capacity

High-Memory Quadruple Extra Large DB Instance

68 GB memory, 26 ECUs (8 virtual cores with 3.25 ECUs each), 64-bit platform, high I/O capacity

Scaling Out

You can scale out a relational database in two different ways:

- Using read-only slaves (*read replicas* in AWS)
- Sharding or partitioning

There are still some hard problems to solve, as sharding/partitioning has not been addressed yet with RDS. Master-slave type scaling is available, though. A slave, or read replica, is easily created from the Console (Figure 3-5). The only requirement on the master RDS instance is that backups are not disabled by setting the backup retention period to 0. Currently, you can have up to five read replicas that you have to launch one by one. Amazon is working on the ability to launch multiple replicas at once, but that is not yet available.

On a multi-AZ RDS instance, launching a read replica goes unnoticed. A snapshot is taken from the standby, the replica is launched, and when it is ready, it starts to catch up with the master. For a normal RDS instance, there is a brief I/O suspension in the order of one minute. AWS advises to use the same instance classes, as differing classes may incur replica lag. With read replicas, you basically introduce eventual consistency in your database (cluster).



The read replica mechanism uses MySQL's native, asynchronous replication. This means replicas might be lagging behind the master as they try to catch up with writes. The interesting thing about this is that multi-AZ RDS instances apparently use another, proprietary type of synchronous replication.

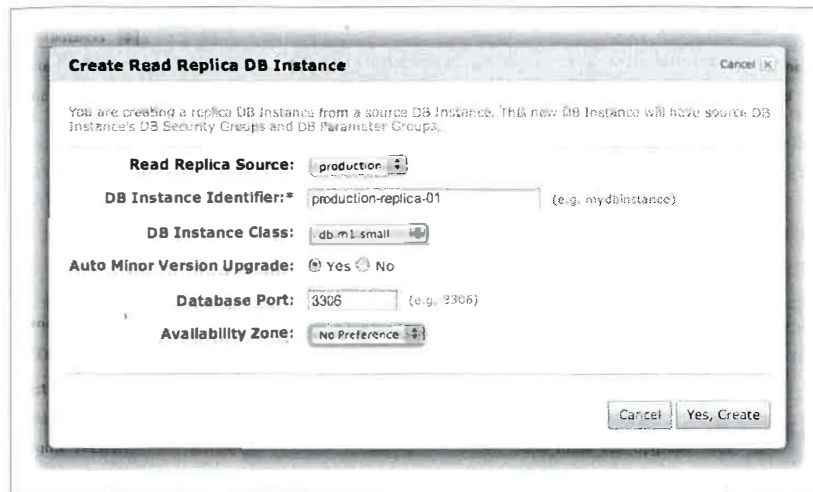


Figure 3-5. Create a read replica

Storage engine

The default storage engine with RDS is InnoDB, but you are free to choose another, like the popular MyISAM. It is important to realize that read replicas on nontransactional storage engines (like MyISAM) require you to freeze your databases, as the consistency cannot be guaranteed when snapshotting. But if you use InnoDB, you are safe, and the only thing you have to do is fire up a new read replica.

Tips and Tricks

RDS is MySQL, but you don't have as much control as you would with MySQL. There are certain peculiarities in RDS' MySQL implementation that can cause a lot of frustration.

Disk is slow

A disk is always slower than memory. If you run your own MySQL using local disks, that's slow as well. But using disk-based operations in RDS is just horrible. Minimizing disk usage means implementing proper indexes, something you always want to do. Other operations that require fast disk access are views. For those experienced with Oracle, this is a huge surprise, but the views in MySQL are not that mature yet.

Slow log

We always enable slow query logging. Sometimes the slow query grows too much. You can access the slow log through the `slow_log` table in the `mysql` database. But you can't just truncate; you have to use the procedure `rds_rotate_slow_log` by executing the following command in your MySQL client:

```
> CALL rds_rotate_slow_log
```

Storage

RDS storage is independent of RDS instance classes. Every class can have from 5 GB to 1 TB of storage associated. Scaling up the storage is easy, and you can do it using the Console. It does require a reboot. On the other hand, scaling down the storage is impossible.

Elastic Beanstalk

As we described in this chapter, AWS offers a number of services that help you scale your application. There are numerous ways to configure the different components, such as Auto Scaling and Elastic Load Balancing. AWS gives you a lot of power, but sometimes you just want a quicker way to get up and running with your app.

And that is exactly what Elastic Beanstalk (EB) can do for you. At this moment EB is Java-only, although this should be just the start. EB works with Tomcat as the Java app server, and the only thing you have to do is give it a WAR (Web application ARchive).

We don't have our own WAR, so we chose to run the provided sample application. After signing up for EB, we were able to launch this application successfully (see Figure 3-6).

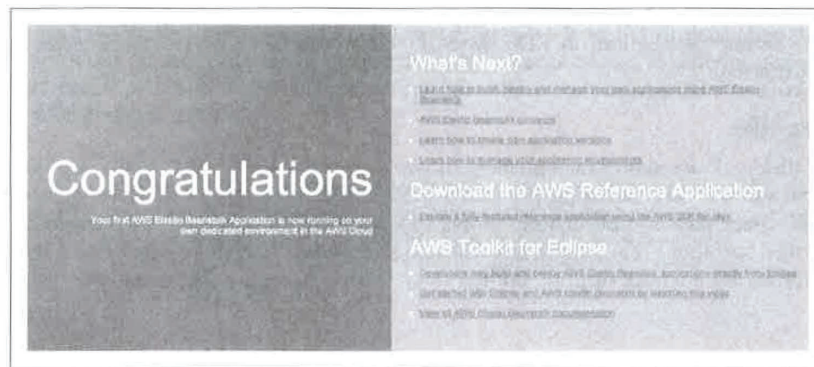


Figure 3-6. Elastic Beanstalk sample application

So what happened here? The default EB setup consists of one micro instance behind an ELB part of an autoscaling group. A security group was created as well. These AWS components are given defaults to operate with. The autoscaling group, for example, has a minimum of one instance and maximum of four.

The supplied AMI when launching a Beanstalk sample application is 32-bit. This means that you can only choose between micro, small, and medium instances. If you create a new application, you can choose the architecture type of the instance you want to use for your environment, either 32-bit or 64-bit.

For production environments with sustained traffic, micro instances are probably not sufficient. If you expect traffic, you will definitely want to choose either small or medium. If you do this, take into account that you will have much more memory, but if you want to utilize this memory, you have to explicitly change the Java Virtual Machine (JVM) settings in the Container tab of the Edit Configuration dialog (Figure 3-7).

Edit Configuration Cancel X

Edit the attributes of your configuration below. When you are finished making edits, click "Apply Changes".

Server **Load Balancer** **Auto Scaling** **Database** **Notifications** **Container**

Container/JVM Options

These settings control container behavior and allow you to pass key/value pairs in as OS environment variables. [Learn more >>](#)

Initial JVM Heap Size (MB) 256m

Maximum JVM Heap Size (MB) 256m

Maximum JVM Permanent Generation Size (MB) 64m

JVM Command Line Options

☐ **Enable log file rotation to Amazon S3**
Note: When enabled Elastic Beanstalk will rotate your Tomcat log files to an S3 bucket every hour.

Environment Properties

These properties are passed into the application as environment variables. [Learn more >>](#)

Cancel Apply Changes

Figure 3-7. JVM settings in Elastic Beanstalk

Most of the settings we have seen in this chapter can be configured in the Elastic Beanstalk section of the AWS Console. This is nice, as we don't have Auto Scaling in the Console yet, for example. Another great feature they integrated with EB is Notifications. This allows you to get emails (using Amazon SNS) about events in your infrastructure, such as when instances are launched or terminated.

There is one more feature worth mentioning, and that is deployment. EB allows you to manage versions of your application. Deployments happen on a running EB cluster (for lack of a better name). If things do not go as planned, you can easily choose another version to deploy, for example when you want to do a rollback. This might just be the killer feature of Elastic Beanstalk.

Even though this service has just been released, it will surely prove useful in helping you get up and running quickly. And if you outgrow the Elastic Beanstalk way of deploying your application, you can start to take over one step at a time.

In Short

In this chapter, we started taking advantage of the elasticity property of AWS infrastructures. On one hand you can scale up manually, changing to bigger instances. However, you can also scale *out* by using more instances, and AWS provides ways of automating this. Before you can set up your infrastructure to scale out, though, you need a way to put your data somewhere so it can be shared. We explored different ways to do this, and showed an example of using S3 for user uploaded data.

The two most important new components we introduced in this chapter are ELB and Auto Scaling. You can use these components together to set up an autoscalable infrastructure. By specifying a few rules, you can determine when your infrastructure will scale out by launching new instances, or scale in by terminating instances when load is reduced again, for example. We also discussed using Auto Scaling without an ELB to make your application more resilient by monitoring an instance and launching a new one when it fails.

We then focused on scaling relational databases. We showed how easy it is to scale an RDS instance up and down by choosing another instance class. For scaling out, we have the option of read replicas. In both cases, using multi-AZ RDS reduces or removes downtime when you make modifications to the RDS machines.

Finally, we gave a brief overview of the Elastic Beanstalk tool, which helps you set up a scalable application very quickly using EC2, Auto Scaling, ELB, S3, and SNS.

Programming Amazon EC2

If you plan to use Amazon Web Services to run applications in the cloud, the end-to-end approach in this book will save you needless trial and error. You'll find practical guidelines for designing and building applications with Amazon Elastic Compute Cloud (EC2) and a host of supporting AWS tools, with a focus on critical issues such as load balancing, monitoring, and automation.

How do you move an existing application to AWS, or design your application so that it scales effectively? How much storage will you require? *Programming Amazon EC2* not only helps you get started, it will also keep you going once you're successfully positioned in the cloud. This book is a must-read for application architects, developers, and administrators.

- Determine your application's lifecycle and identify the AWS tools you need
- Learn how to build and run your application as part of the development process
- Migrate simple web applications to the cloud with EC2, Amazon Simple Storage Service, and CloudFront content delivery
- Meet traffic demand with EC2's Auto Scaling and Elastic Load Balancing
- Decouple your application using Simple Queue Service, Simple Notification Service, and other tools
- Use the right tools to minimize downtime, improve uptime, and manage your decoupled system

Previous programming experience is recommended.

US \$34.99 CAN \$40.99
ISBN: 978-1-449-39368-7



LC COPYRIGHT



"Jurg van Vliet and Werner Vogels have done a great job in this book building a practical guide on how to build real systems using AWS."

—Werner Vogels
VP & CTO at Amazon.com

Jurg van Vliet co-founded 9Apps, an AWS solution provider and silver partner of Eucalyptus. He worked for various web startups and media companies before starting the e-learning firm 2Yellows.

Flavia Paganelli, co-founder of 9Apps, has been developing software over the past 14 years for several industries while working at companies such as TomTom and Layar. She is the creator of the Decaf EC2 smartphone app.

O'REILLY®
oreilly.com

Safari Books Online **Free online edition**
for 45 days with purchase of this book. Details on last page.

