

A Comprehensive Guide to Building a Scalable Web App on Amazon Web Services - Part 1

By Josh Padnick, [Phoenix DevOps](#)

Published on [AirPair.com](#)

It seems like everyone today wants to build the next great scalable web app. And, at least [according to Gartner](#), Amazon Web Services has emerged as the undisputed leading cloud provider where you should build it.

But AWS can look overwhelming. There are now over 40 different AWS services, and many concepts to think about when building your app. AWS provides [excellent documentation](#) on each of their services, [helpful guides](#) and [white papers](#), [reference architectures](#) for common apps, and even a [program specially for startups](#) where startups can receive one month free to speak with AWS Cloud Support Engineers, and in select cases, free AWS credits.

But how do you put it all together? What's the relationship between, say, the Elastic Load Balancer (ELB) that you hear about that "just works" and the seemingly unrelated DevOps concept of Service Discovery? When does it make sense to use a non-AWS solution (e.g. using nginx instead of an ELB)?

Also, how do you balance the need to build an app quickly and inexpensively today, but that's scalable for tomorrow?

My goal in this guide is both to answer these questions, and, more importantly, provide an overall framework you can use to think about *how* to answer these questions.

My target audience is the developer who has a few web or mobile apps under

their belt and understands core programming concepts, but now wants to learn how to build their scalable app in AWS. This guide will also be valuable to developers working in larger organizations who are already running an AWS app, or those who wish to migrate to AWS.

I'll cover both high-level concepts and details. I won't describe the very basics of AWS like "How to Launch an EC2 Instance" but I'll provide best practices and perspective on these areas. Finally, I won't be explicitly discussing Big Data, Event-Driven, or other types of architectures, though there is certainly lots of material here that will apply to those.

Let's get started.

1. What We'll Cover

This guide exists in two parts. You're reading Part 1, where I'll focus on the high-level concepts in AWS and how to put together an AWS architecture. In Part 2, our focus will be on DevOps and maintaining the infrastructure.

1.1 How to Think About AWS & Scalability (Part 1)

We'll start with high-level guidelines about how to approach and think about AWS.

1.2 AWS Concepts (Part 1)

You'll need to learn some -- but not all! -- AWS services in depth. I'll give you brief summaries of the AWS services you'll need specifically to build a scalable web app:

- EC2
- S3
- VPC
- RDS
- IAM
- Route 53

- CloudFront
- CloudWatch
- CloudFormation vs. Elastic Beanstalk vs. OpsWorks

1.3 Architecture Concepts (Part 1)

As you think about what your app will do and how it will scale, you will need to architect a way for your data to flow and be persisted, along with many ancillary concerns. We'll discuss:

- Architecture Paradigms
- Application Layers
- Architecting for Scalability
- Architecting for High Availability
- Docker & Containers
- Security

1.4 DevOps Concepts (Part 2)

In Part 2, I'll cover the operational aspects of running your infrastructure day-to-day with a special emphasis on automation.

Recently, the term "DevOps" has become a fashionable way to refer to these concepts. Technically, DevOps refers to the idea that application developers must actively think about the operational aspects of their code while ops people are using "development" concepts like source control and automated testing to manage infrastructure. But people now use DevOps to generally mean "infrastructure automation".

We'll cover these concepts:

- Configuration Management
- Streamlining Dev Environment Setup
- Orchestration
- Service Discovery
- Log & Error Management

- Monitoring & Alerting
- Dashboards
- Automated Deployment
- Backup & Disaster Recovery
- Security
- Scalability
- Email Service

2. How to Think About AWS & Scalability



I want to start by building your mental model so you know *how to think* about your scalable web app on AWS. I'll share some fundamental ideas that aren't very hard to understand, but they'll make your ramp-up process easier.

2.1 You don't need to learn all 40+ AWS services


The first good news is that, of the 40+ AWS services, you'll only need to dive deep for a small fraction of them. As of December 2014, here's the latest list of all AWS Services:

Amazon Web Services





Compute

-  **EC2**
Virtual Servers in the Cloud
-  **Lambda** PREVIEW
Run Code in Response to Events




Storage & Content Delivery

-  **S3**
Scalable Storage in the Cloud
-  **Storage Gateway**
Integrates On-Premises IT Environments with Cloud Storage
-  **Glacier**
Archive Storage in the Cloud
-  **CloudFront**
Global Content Delivery Network







Database

-  **RDS**
MySQL, Postgres, Oracle, SQL Server, and Amazon Aurora
-  **DynamoDB**
Predictable and Scalable NoSQL Data Store
-  **ElastiCache**
In-Memory Cache
-  **Redshift**
Managed Petabyte-Scale Data Warehouse Service



Networking

-  **VPC**
Isolated Cloud Resources
-  **Direct Connect**
Dedicated Network Connection to AWS
-  **Route 53**
Scalable DNS and Domain Name Registration




Administration & Security

-  **Directory Service**
Managed Directories in the Cloud
-  **Identity & Access Management**
Access Control and Key Management
-  **Trusted Advisor**
AWS Cloud Optimization Expert
-  **CloudTrail**
User Activity and Change Tracking
-  **Config** PREVIEW
Resource Configurations and Inventory
-  **CloudWatch**
Resource and Application Monitoring







Deployment & Management

-  **Elastic Beanstalk**
AWS Application Container
-  **OpsWorks**
DevOps Application Management Service
-  **CloudFormation**
Templated AWS Resource Creation
-  **CodeDeploy**
Automated Deployments




Analytics

-  **EMR**
Managed Hadoop Framework
-  **Kinesis**
Real-time Processing of Streaming Big Data
-  **Data Pipeline**
Orchestration for Data-Driven Workflows

Application Services

-  **SQS**
Message Queue Service
-  **SWF**
Workflow Service for Coordinating Application Components
-  **AppStream**
Low Latency Application Streaming
-  **Elastic Transcoder**
Easy-to-use Scalable Media Transcoding
-  **SES**
Email Sending Service
-  **CloudSearch**
Managed Search Service

Mobile Services

-  **Cognito**
User Identity and App Data Synchronization
-  **Mobile Analytics**
Understand App Usage Data at Scale
-  **SNS**
Push Notification Service


Enterprise Applications

-  **WorkSpaces**
Desktops in the Cloud
-  **Zocalo**
Secure Enterprise Storage and Sharing Service

If you're building a standard scalable web app that does not qualify as "Big Data", here are the AWS services you're likely to use:

Amazon Web Services





Compute

-  **EC2**
Virtual Servers in the Cloud
-  **Lambda** PREVIEW
Run Code in Response to Events


Storage & Content Delivery

-  **S3**
Scalable Storage in the Cloud
-  **Storage Gateway**
Integrates On-Premises IT Environments with Cloud Storage
-  **Glacier**
Archive Storage in the Cloud
-  **CloudFront**
Global Content Delivery Network







Database

-  **RDS**
MySQL, Postgres, Oracle, SQL Server, and Amazon Aurora
-  **DynamoDB**
Predictable and Scalable NoSQL Data Store
-  **ElastiCache**
In-Memory Cache
-  **Redshift**
Managed Petabyte-Scale Data Warehouse Service

Networking

-  **VPC**
Isolated Cloud Resources
-  **Direct Connect**
Dedicated Network Connection to AWS
-  **Route 53**
Scalable DNS and Domain Name Registration

Administration & Security

-  **Directory Service**
Managed Directories in the Cloud
-  **Identity & Access Management**
Access Control and Key Management
-  **Trusted Advisor**
AWS Cloud Optimization Expert
-  **CloudTrail**
User Activity and Change Tracking
-  **Config** PREVIEW
Resource Configurations and Inventory
-  **CloudWatch**
Resource and Application Monitoring







Deployment & Management

-  **Elastic Beanstalk**
AWS Application Container
-  **OpsWorks**
DevOps Application Management Service
-  **CloudFormation**
Templated AWS Resource Creation
-  **CodeDeploy**
Automated Deployments




Analytics

-  **EMR**
Managed Hadoop Framework
-  **Kinesis**
Real-time Processing of Streaming Big Data
-  **Data Pipeline**
Orchestration for Data-Driven Workflows

Application Services

-  **SQS**
Message Queue Service
-  **SWF**
Workflow Service for Coordinating Application Components
-  **AppStream**
Low Latency Application Streaming
-  **Elastic Transcoder**
Easy-to-use Scalable Media Transcoding
-  **SES**
Email Sending Service
-  **CloudSearch**
Managed Search Service

Mobile Services

-  **Cognito**
User Identity and App Data Synchronization
-  **Mobile Analytics**
Understand App Usage Data at Scale
-  **SNS**
Push Notification Service

Enterprise Applications

-  **WorkSpaces**
Desktops in the Cloud
-  **Zocalo**
Secure Enterprise Storage and Sharing Service

Among those services, the bulk of your learning will be in **EC2**, **VPC**, **S3**, and one or more of the persistence services including **RDS** or **DynamoDB**.

A great way to identify the specific set of services you need for your unique app is by reviewing the [AWS Solutions](#) page.

2.2 Making the Right Architecture Decisions

Most of the architecture decisions you make will be a tradeoff of these factors:

- **Time:** How long it takes you to setup
- **Team:** How productive your team will be with this decision
- **Cost:** How much you'll pay to AWS for these services
- **Risk:** How much down time / data loss / security risk you're exposed to
- **Scale:** How many users you can serve / how fast your app is

AWS is a platform, but it's up to you to assemble the pieces. Essentially, AWS allows you to "purchase" the level of availability, redundancy, security or scale that you need.

For example, consider how you might setup a Wordpress site in AWS? Here are two different ways of hosting the same site on AWS. Don't worry about the meaning of every acronym or concept. We'll cover that later; the point now is to show how, for the same app, different needs drive different decisions.

- **Low Power Wordpress:** Everything is hosted on a single virtual machine known as an EC2 instance (Cost). You can use a pre-built and tested Amazon Machine Image (AMI) like [Bitnami Wordpress](#) to be up and running in minutes (Time). Bitnami is well-documented, and we'll setup automated snapshots of the EC2 instance every 24 hours so that if the server goes offline, we can restore to 24 hours ago (Team, Risk). But we'll be exposed to data loss up to 24 hours (Risk). Price will be low (Cost), and we can upgrade the instance type if we get more traffic (Team). If the server goes down, the site will go offline until we restore it (Risk).
- **High Power Wordpress:** We'll use S3 and CloudFront to serve all static files so we can improve site load time (Scale). We'll setup a Wordpress server in each Availability Zone so that if any one server or Availability Zone dies, the site stays up (Risk). We'll use Amazon's managed Relational Database Service (RDS) to host our database with the Multi-Availability-Zone option so that we have automated backups for our database, good performance, and low risk of data loss (Risk, Scale). Because we'll have multiple Wordpress servers, we'll need to setup a special deployment method to ensure that all Wordpress servers are always in sync (Team, Cost). Since we are using multiple AWS technologies like CloudFront, S3, EC2, and RDS plus multiple EC2 instances, we'll pay more (Cost).

I believe the most important factor is "Team." I like how Edmond Lau puts it from one of his [blog posts](#):

The guiding heuristic for evaluating most tradeoffs should ultimately be:
"What course of action will ultimately increase the probability that the team succeeds?"

2.3 Recognize the "spectrums" inherent to AWS services

Many AWS services exist along a spectrum. Recognize these to help guide your choice on which AWS services to use.

For example, AWS Elastic Beanstalk, OpsWorks, and CloudFormation all provide the same core service of orchestrating your infrastructure (i.e. launching your EC2 instances, setting up Security Groups, etc.), but you trade convenience vs. control.

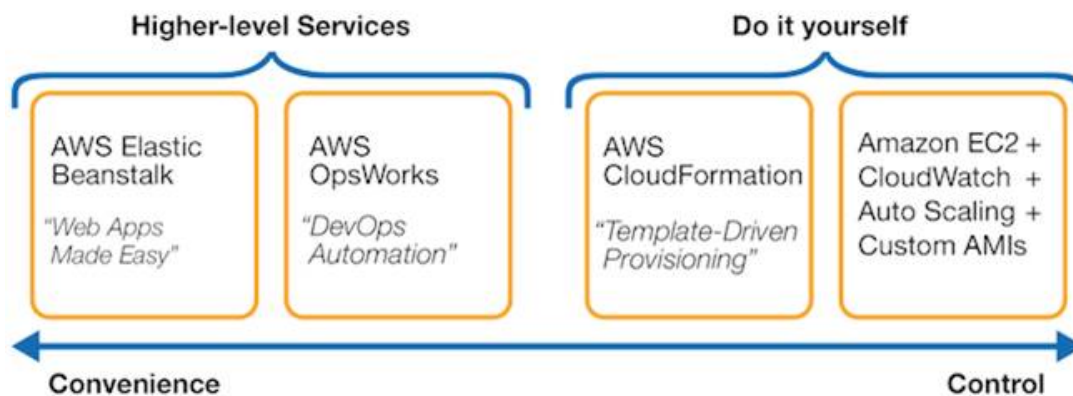


IMAGE SOURCE: [DevOps, PaaS and Everything in Between](#)

Or when dealing with storage data, you can store files in S3 where files can be retrieved within milliseconds of requesting them, or in Glacier where files are available 2 - 6 hours after requesting them but at a cost that is 1/3 the cost of S3. Here the spectrum is "Speed vs. Cost."

2.4 Recognize the third-party alternatives to AWS services

AWS has been quite aggressive about "horizontally" expanding its services. As an example, did you know you can buy domain names through AWS? It's part of **AWS Route53** (AWS's DNS-as-a-Service).

AWS also recently released a competitor to Dropbox (**AWS Zocalo**) and even an alternative to GitHub (**AWS CodeCommit**).

In fact, here's a table of services provided by AWS which are also provided with great success by other third-party companies:

Feature	AWS Service	Third-Party Alternatives
Email	AWS SES	SendGrid, Postmark, Mailgun
DNS	AWS Route53	DynDNS, dnsimple
Domain Names	AWS Route53	GoDaddy, Register.com
Key Management	AWS Key Management Service	AWS CloudHSM, Trend Micro SecureCloud, Porticor
Content Delivery Network	AWS CloudFront	CloudFlare, Akamai
Hosted Active Directory	AWS Directory Service	adapt, Emantra
Log Management	AWS CloudWatch Logs	Loggly, Splunk, SumoLogic
Mobile Backend	AWS Cognito	Parse
End-User File Sync	AWS Zocalo	Dropbox

My default position is to start with the AWS service. First, it's integrated into my AWS environment. Second, if applicable, it will generate service-specific alerts to the right people using email, text message, or other means via AWS Simple Notification Service (SNS). Third, I can probably grant granular permissions on that service to my existing AWS IAM users (i.e. the other members of my team who need to login to the AWS console or access AWS via the API). Also, it's usually the cheapest.

But there are also many cases where third-party services are superior. Log management is a good example. AWS CloudWatch Logs lets you define your own

custom metrics ("metric filters") based on your application log data. You can then view these metrics in chart format or set alerts on them. It also lets you view real-time log data as it streams in.

But tools like Loggly and SumoLogic (both of which have free versions) allow for sophisticated searching, and can summarize your log data in more interesting and useful ways than AWS CloudWatch Logs.

2.5 AWS will let you scale both up and out, but plan to eventually scale out

As your traffic grows, you can either "scale up" by using increasingly powerful EC2 instances, or "scale out" by simply adding more instances to handle your app's load.

2.5.1 Scaling Out

The classic wisdom is that you should architect your app to scale out. Scaling out means you can adjust your system capacity (by adding or removing EC2 instances) as your app gets more or less load. This works especially nicely for apps that experience an occasional surge in activity, like an eCommerce site on Cyber Monday.

Scaling out also implies that if one of your servers dies, this becomes a non-event because you've architected your app to be independent of any single EC2 instance.

One of the best things about scaling out, is you can automate it. AWS offers a set of building blocks to facilitate this. For example, Auto Scaling Groups are an EC2 feature where a metric of your choice -- e.g. an instance's CPU load or a custom app metric -- can trigger launching or terminating additional EC2 instances upon exceeding a threshold you define.

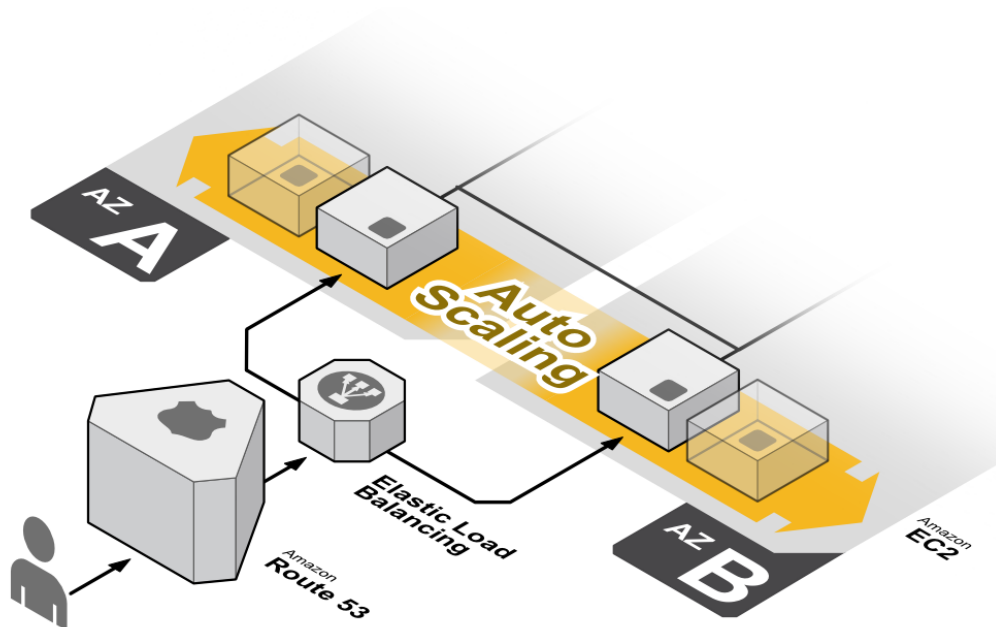


IMAGE SOURCE: [Auto Scaling with the AWS Management Console](#)

But architecting your application to scale out is non-trivial. We'll cover this in more detail later.

2.5.2 Scaling Up

Be sure to leverage scaling up, too. Once you create an EC2 instance, it's possible to stop it, upgrade the instance type, and then start it.

Sometimes, an exclusively "scale up" strategy can work wonders. [StackOverflow is famous for having executed well on this.](#) But an exclusively scale-up strategy usually can't dynamically adjust capacity up and down, and is usually more expensive in public cloud platforms like AWS.

2.5.3 What I Do

When I build my own apps, I architect them from day 1 to support horizontal scaling, primarily by ensuring my web/app tier is completely stateless (more on this later). I then launch them with t2 instances (AWS's entry-level instance type) to start, and if my metrics like CPU Load, Memory Usage, Network I/O, or

Average API Response Time are too slow for my liking, I'll upgrade to more powerful instances. At some point, I stop scaling up and start scaling out.

AWS also recently announced support for Docker as part of its [EC2 Container Service](#). This changes the calculus of when/how to scale out, and we'll cover this later as well.

2.6 Always build Multi-Availability Zone (Multi-AZ) architectures

When you launch a single EC2 instance, you are launching a virtual machine running on a physical server at one of Amazon's data centers. That means any of the following events could take down your instance:

- Your instance itself could fail, or its underlying hard drive volume (Elastic Block Store volume) could become corrupted
- The physical machine on which your EC2 instance resides could fail
- The data center within which the physical machine is located could fail

In AWS-speak, the "data center" is called an **Availability Zone (AZ)**. About 5 - 15 miles away from one AZ is at least one more AZ. A whole cluster of AZ's is known as a **Region**. There are 3 Regions in the USA and 11 worldwide.

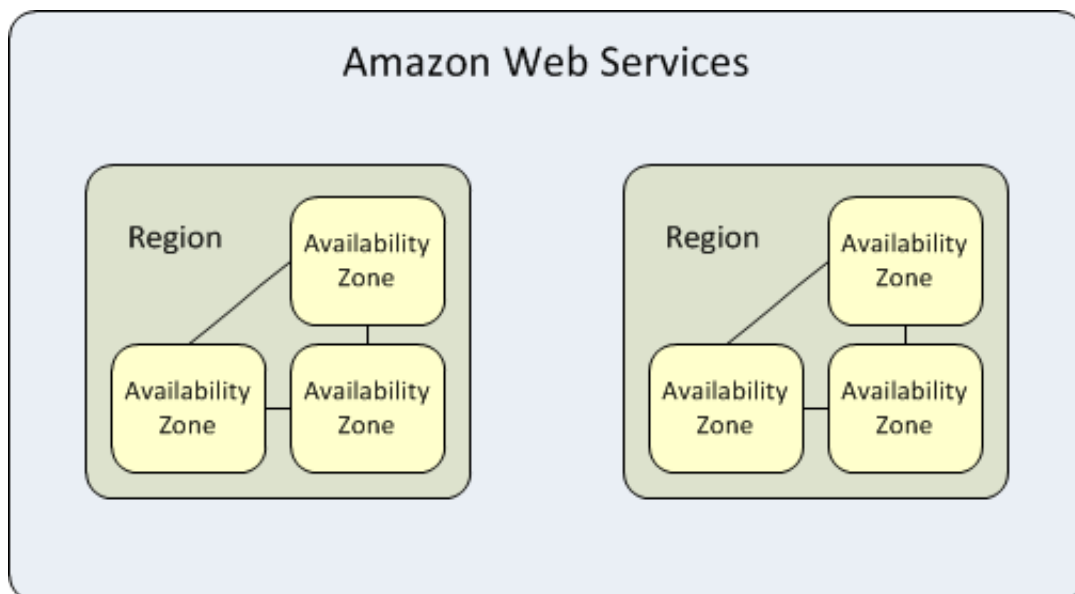


IMAGE SOURCE: [Official AWS Documentation](#)

2.6.1 Leverage MultiAZ for High Availability

You should assume that any of your EC2 instances will fail at any time in any of the ways outlined above. When a single EC2 instance or an entire AZ goes offline, your architecture should detect this and simply stop routing traffic to the affected instances. This is the idea behind a Multi-AZ setup, and we'll cover it in more detail later.

Some AWS services such as AWS RDS (where AWS manages a relational database instance of you) have built-in support for Multi-AZ and enabling it requires simply checking a box in the configuration.

You can go beyond Multi-AZ and use a Multi-Region architecture, where your app can survive an entire region going offline, but a multi-Region setup introduces new complexities and additional cost and is typically only implemented by larger organizations.

2.7 You will hardly ever pay upfront, and pay only for what you need.

AWS repeatedly emphasizes their goal to shift your thinking "from CapEx to OpEx" or "Pay for what you use." In other words, in the past to achieve the kind of infrastructure AWS offers, you would need to invest a significant sum upfront (a "Capital Expense" or CapEx). Today, AWS is philosophically committed to eliminating the CapEx so that you pay only ongoing "Operating Expenses" (OpEx) to run your infrastructure.

This means AWS services are almost always priced by usage. For example, you pay by the hour when you run an EC2 instance. If it's stopped, you pay nothing. S3 is charged per GB of data stored along with some bandwidth costs.

A notable exception to this rule is the concept of a "Reserved Instance." This is a billing option that AWS offers where you can "purchase" a lower hourly rate for EC2 instances by paying upfront. We'll cover this shortly.

2.8 Plan out and monitor your costs and use Reserved Instances

One of the hardest things with AWS is managing your costs. Most of us love the "Pay for what you use" model, but in practice you wind up using a wide variety of services and it's hard to keep track of them of all.

AWS recognized this challenge and, to their credit, has provided several tools to help you manage your costs.

2.8.1 Estimate Your Costs Upfront

AWS provides an [Online Cost Calculator](#) you can use to estimate your monthly (and if applicable upfront) costs based on the services you plan to use.

2.8.2 Use Billing Alerts

Your AWS bill should be inline with what you expect from the calculator, and should remain pretty stable unless you use more services. You can guarantee these assumptions are true by setting up [Billing Alerts and Notifications](#).

Rich Adams has an outstanding article on [AWS Tips I Wish I'd Known Before I Started](#) where he advises setting up a billing alert for his expected usage each week.

So the first week's alert [is] for say \$1,000, the second for \$2,000, third for \$3,000, etc. If the week-2 alarm goes off before the 14th/15th of the month, then I know something is probably going wrong.

2.8.3 Use the Right EC2 Pricing Model

When you launch an EC2 instance, you pay per hour. But the hourly rate you pay can vary depending on which pricing model you use.

2.8.3.1 On Demand Instances

If you just launch an instance you are paying the **On Demand** rate. For example, an m3.large instance costs \$0.140/hour, or about \$100/month, plus a negligible charge for the EBS volumes your instance uses (\$2/month for 20GB) if you run your instance 24/7 for the month. If you stop your instance, you pay only for the cost of your EBS volume, and pay nothing for the EC2 instance.

2.8.3.2 Reserved Instances

If you expect that you will keep your EC2 instance running for an extended period of time (e.g. 24/7 for a year), you can sign up for a **Reserved Instance** and save up to 60%! This is a billing concept only. It will affect how you're billed and it will not affect the technical performance of the EC2 instance in any way.

The basic idea of a Reserved Instance is that you commit to AWS that you will use a particular profile of EC2 instance (e.g. Oregon region, m3 instance type, etc.) for a 1-year or 3-year period. You signify your commitment by either paying:

- 100% upfront, Nothing ongoing (highest savings)
- 0% upfront, but you commit to pay monthly through your term (lowest savings)
- Some upfront in exchange for a lower hourly rate (medium savings)

Savings here are significant, ranging from around 30% - 60% depending on the length of your commitment and your utilization of the services. If you know you're going to use EC2 instances for a year, this is a no-brainer.

The caveat here is that you can change *some* but not all of the conditions of your Reserved Instance and there is no official refund from AWS. See the [AWS Reserved Instances FAQ](#) for more information.

While AWS won't issue any refunds, you can actually sell your remaining contract on the open market for a price that you set through the AWS-provided [Reserved Instance Marketplace](#).

2.8.3.3 Spot Instances

Finally, you can purchase **Spot Instances** which means you bid an hourly rate

you're willing to pay and if your bid is above the current "spot" rate set by AWS, an instance will launch. As of November 28, 2014, that same m3.large instance goes for just \$0.0165/hour (a monthly equivalent of \$11!). But the catch is that once the spot rate goes above your bid, your instance will immediately terminate. This model is suited to large ongoing computations and obviously requires some special architecture.

See the [EC2 Pricing Page](#) for more information.

2.8.4 Consider AWS Trusted Advisor

If you happen to have a Business Support plan or better, you get access to the full version of AWS Trusted Advisor, which will automatically analyze your account for instances which are good candidates for Reserved Instances, or which are sitting idle.

3. Key AWS Services

Now that you have a high-level perspective on AWS, let's dive one level deeper and give a brief summary of the AWS services essential to a scalable web app. Use the [AWS documentation](#) or other resources to learn the "how", my goal is to give you the "what" and "why."

3.1 EC2

Elastic Compute Cloud (EC2) is where you launch servers. In AWS, "servers" are actually virtual machines, and in AWS lingo a single virtual machine is an "EC2 Instance."

Each EC2 instance is backed by one or more "virtual hard drives", which are known as **Elastic Block Store (EBS) Volumes**. You can launch EC2 instances based on a "hard drive snapshot" which is known as an **Amazon Machine Images (AMI)**. For example, you can select an Ubuntu 14.04 AMI, a Windows Server 2008 AMI, etc.

EC2 Instances are protected with "firewalls" known as **Security Groups**. For example, to SSH into your instance, the Security Group associated with your instance must permit port 22 from your IP address.

Every EC2 instance has a private IP address, which is private in the sense that it's only visible within the network in which your EC2 instance resides (known as a **VPC**, see below). Some EC2 instances can also have one more public IP addresses. If you can assign an EC2 instance's public IP address to a different EC2 instance, that IP address is known as an **Elastic IP Address**.

EC2 instances can be stopped, (like turning off the server), started (like turning it on), or terminated (like destroying it).

3.2 VPC

A Virtual Private Cloud (VPC) is a private network where you place your EC2 instances.

Originally in AWS, every EC2 instance was directly exposed to the public Internet, guarded only by the Security Group rules set on it. Many users recognized that in a traditional data center you could setup private networks using direct wiring; VPC attempts to achieve a version of this in the cloud.

Each EC2 instance is placed into a particular **Subnet**. A Subnet specifies a range of IP addresses and exists in a particular **Availability Zone**. A Subnet is either directly exposed to the Internet (a "public subnet") or unreachable from the Internet (a "private subnet"). For example, you probably do not want your database server to be exposed to the public Internet, so you should place it in a private subnet.

When you want to setup high-availability architectures, you can spread EC2 instances across different subnets, and thereby different Availability Zones.

3.3 S3 and Glacier

Simple Storage Service (S3) is for storing files in the cloud.

Glacier is just like S3 but cheaper, with the tradeoff that once you request a file, it may take 2 - 6 hours before it's available for download.

To upload a file to S3, first you select or create an **S3 Bucket**, which is simply a namespace for a group of files. The file you upload in S3 can either be publicly downloadable via a URL, or private and only available in certain cases.

A common pattern for web and mobile apps is to store all user-generated files in S3. Such files should generally not be public, so when a user needs to download a file you can issue a temporary access token which can be used to download a file from S3 from its URL for a defined period of time (e.g. 60 seconds).

You can assign "policies" to an S3 bucket regarding what happens to files in a bucket. For example, files in a bucket can be automatically deleted after a certain period, automatically archived to Glacier, or both.

Files uploaded to S3 can be encrypted using a key you provide, or one transparently managed by AWS.

3.4 RDS and DynamoDB

Relational Database Service (RDS) is a managed relational database service for MySQL, PostgreSQL, SQL Server, or Oracle.

DynamoDB is a managed NoSQL service that uses a proprietary AWS NoSQL engine.

There are many challenges to administering your own database servers. RDS and DynamoDB automatically handle backups, scaling, master-standby replication, read replicas, database version updates, and security patches.

The primary alternative to RDS and DynamoDB is managing your own EC2 instances running a database software of your choice.

AWS recently announced **Aurora**, a massively scalable MySQL-compatible database as a service.

3.5 IAM

Identity and Access Management (IAM) is a set of services for managing permissions among your human team members and AWS resources.

When you first create your AWS account, the email address you use will be the "root" account and will have super-admin access. It is a best practice to never login with this account, and instead create individual accounts for each member of your team using IAM.

You can then assign permissions to each account so that, for example, some team members can see all information but cannot start or stop an EC2 instance.

IAM permissions can also be assigned to AWS resources such as an EC2 instance itself. This enables an EC2 instance with the right permissions to automatically have access to, for example, a particular folder in S3 without the need for separate authentication.

Recently, AWS added **Key Management Service** to IAM so that you can centrally manage symmetric encryption keys.

3.6 Route 53

Route 53 is DNS as a service, and also includes the ability to purchase domain names.

Using Route 53, you can setup public DNS records for any domain name. Route 53 recently introduced the ability to setup private DNS as well (DNS that only your private network can query but which is unavailable to the outside world).

You can use Route 53 in conjunction with many other AWS services. For example, using Route 53 in conjunction with S3 enables you to host a static website on S3.

3.7 CloudFront

CloudFront is a Content Delivery Network managed by AWS.

CloudFront contains **Edge Locations** throughout the world which can cache certain content. As a result, when users access CloudFront-cached content, no load is placed on your app, and fewer physical network hops are required for the user to receive the data.

CloudFront can work with either S3 or an HTTP Server as the **Origin Server** from which it will source the original content.

CloudFront also supports streaming media files.

3.8 CloudWatch

CloudWatch provides monitoring and alerting for your AWS resources. CloudWatch also offers basic features for managing application log data on your EC2 instances.

Each AWS service has a unique set of metrics it exposes to CloudWatch. For example, EC2 instances expose CPU Utilization, Memory Utilization, and much more, while RDS instances expose Database Connections, Read IOPS, and CPU Utilization.

You can set automated alerts that trigger when a metric exceeds a threshold you define. The result of the alert can be that someone gets notified, or that an action within AWS is taken. For example, a classic pattern is to automatically launch additional EC2 instances when average CPU utilization on existing EC2 instances exceeds a certain threshold.

3.9 CloudFormation vs. Elastic Beanstalk vs. OpsWorks

CloudFormation is a way to capture everything you can do in the AWS Web Console as a single JSON file. This means you can "version control" and "code review" your AWS infrastructure settings.

OpsWorks accomplishes the same infrastructure management as CloudFormation, but at one higher level of abstraction, and with a special

focus on application deployment.

Elastic Beanstalk accomplishes the same infrastructure management as CloudFormation and OpsWorks, but at the highest level of abstraction. For example, you can launch a NodeJS stack with just a few clicks.

The most primitive (and most common!) way of setting up your infrastructure on AWS is to use the web console. But when you work on large teams, this can cause problems. Imagine that Engineer A changes a security group setting and then goes home. An hour later, Engineer B receives an alert that a service is down. If Engineer B doesn't have a record of what was changed, discovering the root cause of the issue becomes difficult.

A best practice for managing AWS infrastructure is to run every single change through CloudFormation and to commit your changes to version control. In the example above, Engineer B can now immediately check the CloudFormation template for updates to see if that caused the issue.

But sometimes CloudFormation is overkill, especially for AWS beginners. At the other end of the spectrum is Elastic Beanstalk, which gives you "point and click" access to deploying an entire stack of common technologies, or even of Docker Containers! While you may use "point and click" to provision the infrastructure, you can still see all the associated AWS resources (e.g. EC2 instances) in their usual place.

OpsWorks was an acquisition by AWS formerly known as [Scalarium](#), and is a sort of middle ground. OpsWorks offers more control than Elastic Beanstalk and also introduces "point and click" deployment options, but is also more limiting than using CloudFormation.

3.10 Other AWS Services

To keep things short, I've only summarized the AWS services that are most essential to the scalable web app, though certainly you will use others beyond the ones I've listed.

4. Architecture Concepts

Now it's time to discuss the decisions you will make as you create your system architecture on AWS.

For a complementary perspective on how your architecture might evolve over time from a single EC2 instance to hundreds of auto-scaled instances, watch the excellent presentation from Chris Munns at AWS re:Invent 2014 on [Scaling Up to your First 10 Million Users](#).

4.1 Architecture Paradigms

One of the first decisions to make is whether to build your app as a "single stack" or "many small single stacks."

4.1.1 Single Stack: The Monolithic Architecture

Traditionally, the default architecture for a new software project was the classic n-tier architecture. Wikipedia gives a nice [general overview](#) of this.

As the Wikipedia link describes, this means setting up your persistence tier as a database (e.g. PostgreSQL or MongoDB) and/or cache (e.g. Redis or Memcached), setting up your middle tier to handle business logic, and setting up your frontend tier, typically with a web server like Nginx or Apache.

The defining feature of the single-stack architecture is that all or most of your code is contained within these tiers. Your persistence tier will have *all* your database tables or document types. Your middle tier will contain your *entire* domain model and business logic. Each tier may have a single EC2 instance or multiple EC2 instances spread across multiple Availability Zones.

A few years ago, many software teams began to complain that their N-tier architectures had become so large that enhancing and maintaining it was becoming increasingly painful and expensive. In such situations, they pejoratively labeled the N-Tier architecture a "Monolithic Architecture."

Indeed, there are [many examples online](#) of companies who share their journey from Monolith to a microservices architecture.

4.1.2 Many Small Single Stacks: The Microservices Architecture

So what is a microservices architecture? The microservices architecture is a paradigm where each microservice is a standalone "single stack" that, as Sam Newman concisely summarizes in [Building Microservices](#):

- Is small, and focused on doing one thing well
- Is a separate, independent process
- Communicates via language agnostic APIs
- Is highly decoupled

A microservices architecture can be thought of as an "approach" to a Service-Oriented Architecture (SOA), though there is often no clear distinction between the two terms.

As an example of a microservices architecture, if you were building a grocery shopping app, you might have a microservice just for creating, managing, and authenticating users, one for managing your catalog of SKUs, and one for managing your inventory levels. Each of these microservices would be on their own independent stack of EC2 instances, could be managed by separate teams and even use different technology choices, connecting to each other via RESTful APIs.

There are many benefits to a microservices architecture and it is certainly the hipster way to build a new app today. But, as with everything in software, there are Pro's and Con's. Check out Martin Fowler's [Microservices](#) article for a good overview and comparison. I also like [Armon Dadgar's summary](#).

Here's my take on the issue, albeit with some broad generalizations:

	SINGLE STACK	MICROSERVICES
Technology Selection	Commit upfront to a single tech stack (e.g. .Net or Node.js)	Option to use a different tech stack for each microservice
Team Setup	One or more teams work on one large app	Each microservice is owned by a dedicated team
Ability to Make Changes	Easier to make app-wide changes that modify class service contracts	Because you don't know how your microservice is consumed, API changes are expensive
Agility	Easier to fix suboptimal architectures by refactoring a full vertical slice	If you partition your app into the wrong microservices, re-partitioning is expensive
Scalability	You scale the entire app at once	You can scale each microservice individually

As you can see, there are many benefits to a Microservices architecture, but it has its downsides, too.

4.1.3 Microservices and Infrastructure Automation

A critical point about a microservices architecture is that infrastructure automation (which I discuss in Part 2) becomes extremely important. It's hard enough to manually manage a single stack infrastructure with its N tiers. Imagine managing 5 single stacks manually?

Part of the reason microservices is even a viable architecture today is because of the recent advances in infrastructure automation. Ideally, when setting up your microservices architecture, you would "template" your full stack so that it can be easily reused.

For example, you would use **Configuration Management** technologies like [Chef](#), [Puppet](#), [Ansible](#), or [Saltstack](#) to automate the configuration of servers at each of your tiers, **Orchestration** tools like AWS CloudFormation or [Terraform](#) to automate the deployment of AWS infrastructure like EC2 instances, RDS (Amazon's managed relational database store) instance or an Elastic Load Balancer. You would bake into your server configurations a common standard for handling logging, alerting, monitoring, process management, security procedures, and more. The details of this are the heart of our Part 2 article.

Actually, the requirement to streamline your infrastructure is itself a benefit, but, per our decision making guidelines discussed earlier, if your team isn't prepared to setup this level of infrastructure automation, microservices may be a very expensive architecture.

Personally, I start my greenfield projects today with a microservices architecture, but that's only because I'm willing to invest the heavy overhead upfront to automate as much of my infrastructure as possible.

4.1.4 Microservices and Asynchronous Message Passing

As we've said, each microservice may call other microservices. This will raise an important question in your mind: when someone calls one of your microservice: does this need to be a **synchronous** or **asynchronous** call?

Imagine you're running an eCommerce site. If you are collecting a credit card from a user, when the user presses "submit" we need to let them know immediately whether their credit card was approved. This business rule means that "process credit card for approval" must be a synchronous call. The "submit button" code should pause while it waits for a response from the "process credit card" service (by the way, it hopefully pauses in a "non-blocking" way so that the thread on which it's operating is freed up to do other things). This is a classic example of a synchronous call, and you would typically use an HTTP RESTful web service to make the call.

But once that credit card is approved and the user is ready to submit an order, this can be done asynchronously. Your business will rarely "reject" a newly submitted

order, so in this case we could submit the order not synchronously via a RESTful API call, but asynchronously using a message queue. With a message queue we simply submit our order to the queue. Meanwhile, some other service like the "order processing service" is consuming messages off the queue as fast as it can handle them. In our example, the "order processing service" might use a First-In-First-Out (FIFO) processing order.

In AWS, you can implement such a queue either by using a key-value store like Redis (or AWS's managed Redis, ElastiCache), or you can use **AWS Simple Queue Service (SQS)** which offers the benefit of a fully managed queue and the detriment of some additional lock-in to AWS.

4.1.5 Microservices and AWS

AWS is especially well-suited to the infrastructure automation that microservices require. For example:

- You can represent an entire application stack, including EC2 instances launched, auto-scaling groups, security groups (individual firewalls for servers), and just about everything else all as code using **AWS CloudFormation**.
- One option for streamlining deployment is to use a tool like [packer](#) to automate the creation of an **Amazon Machine Image (AMI)** that's used to launch a new EC2 instance.
- AWS recently announced the [EC2 Container Service](#) to streamline deploying multiple microservice stacks across a cluster of EC2 instances using Docker containers.

4.2 Application Layers

Whether you're dealing with a single stack or a microservices architecture, you're most likely going to have at least one instance of an N-Tier architecture, that is, at least one full stack. The most common tiers in an architecture are as follows, with a discussion on handling SSL toward the end.

4.2.1 Load Balancing Tier

The role of this tier is to evenly distribute incoming traffic to the next tier, which is likely to be either the Web Tier or App Tier, but could be any tier. The official AWS solution (and the most popular solution) for this tier is an **Elastic Load Balancer (ELB)**, but you can also setup your own EC2 instances to handle this function using software like [HAProxy](#) or [Nginx](#).

Your load balancer is often the "entry point" to your app so high availability here is essential. AWS manages high availability for you when using an ELB, whereas managing your own EC2 instance may require setting up things like automatic failover.

If you use AWS Route 53 to handle your DNS, you can take advantage of [Route 53 Health Checks](#), which means that Route 53 checks the health status of the endpoint it's routing to, and if it fails the standard you specify, will route traffic to healthy endpoints.

But be careful here, because DNS responses from Route 53 include a "Time To Live" property that indicates for how long the DNS response should be cached by a local Internet Service Provider's DNS servers. Not all DNS servers respect this value, so you can't guarantee instant fail-over for all your clients. It's still a handy option, though, since it's an automated way of re-routing traffic.

4.2.2 Web Tier

The role of this tier is to serve static files to users, and often to route traffic to the right endpoint in the App Tier.

A popular option for the Web Tier is to use software like [Apache](#) or [Nginx](#) installed on your EC2 instances. These servers are often the final arbiters of the HTTP response that your app sends back to the client, defining properties like HTTP headers, Cross-Origin-Resource Sharing (CORS) preferences, and more.

Sometimes they double as a load balancer, as is the case with Nginx, but you can still use an ELB for load balancing and Nginx as your web server.

In fact, it's a best practice to leverage the High Availability of an ELB and to have a Multi-AZ web tier so that if one EC2 instance in your web tier goes down, no one will notice.

While web server software is designed to serve static files very quickly, it's still best practice to use **S3** or **AWS CloudFront** (AWS's Content Delivery Network) to deliver static files.

Web Servers are also an excellent place to use Auto Scaling Groups. If CPU load, network traffic, or I/O exceeds a certain threshold, you can configure your ASG to simply launch more instances.

4.2.3 App Tier

The role of this tier is to run your app's main process. For example, if you've built a Java app, this tier is running a Java program that's listening for incoming connections. If you've built a PHP app, this tier is running a web server with PHP support and listens for incoming connections.

Basically, your technology choices will dictate how this server is setup. For that reason, AWS is the least "opinionated" about this tier, providing you with nothing more than a server (the EC2 instance plus its underlying EBS volumes) and the supporting tooling like Elastic IP Addresses, Security Groups, and AMIs,

Your App Server is sometimes where "state" is stored in your app. For example, a user logging in may have his session information on a particular EC2 instance in the App Tier. The problem with this is that this user is now "pinned" to one EC2 instances, and if it fails, your user's session dies with it.

The best practice here is to make your App Tier instances stateless and store things like session state in a different tier. We discuss this more in [4.3 Architecting for Scalability](#)

4.2.4 Cache Tier

The role of the Cache Tier is to store ephemeral data like user session

information, or the results of commonly requested queries. When data is requested from the Cache Tier, either the Cache Tier has the requested data and serves it back (known as a "hit") or it does not have the requested data (known as a "miss") and your app must either write new data to the Cache Tier (e.g. a session key) or fetch existing data from the Database Tier (e.g. a commonly run query) and store it in the Cache Tier.

The idea behind a cache is that accessing memory (i.e. RAM) is orders of magnitude faster than accessing data on disk and avoids repeatedly querying the database for the same information. So we shift load away from our Database Tier to the Cache Tier when possible.

Note that nothing prevents you from storing your session data in a relational database, but you're unnecessarily adding extra load to a future bottleneck.

In theory, because a cache server handles ephemeral data, it should be able to be completely reset without affecting your app (other than resetting all active sessions). But in practice, if the Cache Tier fails, suddenly an enormous amount of load is put on the Database Tier, sometimes causing a cascade of problems. For this reason, some cache servers like Redis support persisting their data to disk to support rapid recovery.

4.2.4.1 The Cache Tier in AWS

You can setup any cache server of your choice on an EC2 instance, or AWS provides the **ElastiCache** service as a managed Cache Tier and lets you choose whether it should use [Memcached](#) or [Redis](#) as the underlying software. Redis is generally the newer and more popular solution on newer projects.

When your Cache Tier holds all ephemeral state like session values, your App Tier no longer has to maintain this state itself. This means you can launch additional EC2 instances, and as long as they are configured to look to the Cache Tier for ephemeral data and the Database Tier for persistent data, they will "just work." For this reason, a Cache Tier is usually an essential part of auto scaling.

4.2.5 Database Tier

The Database Tier is where all your persistent data is managed. In general, data is persisted either to a Relational Database Management System (RDBMS) such as PostgreSQL, MySQL, SQL Server, or Oracle, or to a "document store" (more commonly known as NoSQL) database such as MongoDB or AWS's proprietary DynamoDB.

Deciding whether to use an RDBMS (relational) or NoSQL database is a very important architecture decision that is outside the scope of this guide.

4.2.5.1 AWS DynamoDB

If you decide to use AWS DynamoDB you are entering a world that comes as close to zero administration as I have seen. DynamoDB has a single metric -- throughput capacity -- that you modify as your app scales, plus settings around data consistency and other items that are mostly one-time settings.

Otherwise, there isn't much more to manage. Perhaps most telling is that AWS released backup/restore functionality for DynamoDB *two years after DynamoDB itself was released*. Users were concerned that their own app might write bad data or otherwise corrupt the database, but AWS takes primary responsibility for ensuring your data itself is durable.

That being said, you should still consider taking DynamoDB backups and storing them offsite. When AWS events occur, they tend to cause a chain reaction of events, leading to emergent behavior that can be hard for AWS to predict. See the [October 2012 AWS Outage Post-Mortem](#) to get a sense of what I'm talking about.

4.2.5.2 Everything Else

If you're using DynamoDB you should be paranoid about things going wrong in your Database Tier. If you're not using DynamoDB, you should be very, very paranoid about things going wrong. If your application were to go offline, you can eventually restore it, but losing even a small amount of data is usually considered catastrophic. The most common issues to worry about are:

- **You could lose data because...**

- A database server failed and you had no real-time data replication
 - Your real-time replication was setup but did not function properly
 - Your backups failed to occur
 - Your backups occurred but are corrupted or otherwise unusable
 - Your backups occurred but failed to transfer to your offsite storage
 - A rogue employee deleted backup data
- **You could lose availability because...**
 - A database server failed and you had no automatic fail-over setup
 - Automatic fail-over took far longer than expected
 - Automatic fail-over itself failed
 - Automatic fail-over worked, but Service Discovery issues prevented automatic fail-over from working properly
- **You could suffer from slow performance because...**
 - Your database server is receiving too much load for the EC2 instance type it's running on
 - You are not offloading enough load to Read Replicas
 - You are not offloading enough load to the Cache Tier

And I haven't even covered everything. So how do you handle these issues? The main solutions here are:

- **Use real-time data replication for High Availability.** This can enable a hot standby so that if your master database server fails the hot standby immediately becomes the new master.
- **Use real-time data replication to create Read Replicas.** As your database load increases, you can send all writes to one server and put all reads on a "Read Replica" server to reduce load on the "Write" server. This can introduce data consistency issues, though, so make sure you understand the consistency guarantees your architecture makes about reading data from a read replica.
- **Setup automated data backups.** Each database server comes with built-in backup functionality to take a full backup of the database. For example, in

the PostgreSQL world this is the `pg_dump` command. In MongoDB, this is `mongodump` command. Sometimes your data set is so large that only continuous backups are an option. Sometimes using alternative database backup strategies like file system snapshots in combination with log replay is the right strategy.

4.2.5.3 The Database Tier in AWS

As you can see, there is a lot to manage in your Database Tier. AWS recognized this and created the **AWS RDS** service for Relational Databases and **AWS DynamoDB** for Document Store Databases, which means that AWS manages almost all the above issues for you, and your job is merely to configure, minimally administrate, and occasionally validate things.

AWS RDS is available for MySQL, PostgreSQL, SQL Server, and Oracle. RDS will give you "one-click" Multi-AZ replication, simple steps to creating a Read Replica, automated backup, point-in-time recovery, control over the database configuration options, and the ability to scale storage and CPU as needed.

AWS RDS is an excellent choice for your relational database, but it's still not completely hands off. In *Section 4.5.2 High Availability for Your Stateful Tiers*, we'll discuss some of the "gotcha" issues of RDS. Also, remember that you can always set up all the above yourself on self-managed EC2 instances.

If you use a NoSQL database, AWS DynamoDB is the fully managed NoSQL option and requires virtually no administration. Keep in mind that DynamoDB is *not* a managed version of MongoDB or other third-party product. This is AWS's proprietary product. As before, the primary alternative is to use your favorite NoSQL database on self-managed EC2 instances.

4.2.6 Ancillary Tiers

There may be additional instances that are ancillary to your app. For example, you may use a **Bastion Host** to more securely connect to instances, a **Continuous Integration / Continuous Deployment** instance, an instance to run background jobs, an instance for displaying a dashboard, or accumulating data

like ElasticSearch.

I don't discuss these here because these are incidental, not fundamental to our app. But in Part 2 when we cover the DevOps concepts, we'll go over these in more detail.

4.2.7 SSL

Most applications today need to offer their service via SSL. Since we're discussing this in the context of "Application Layers," the obvious question is **on which layer should SSL terminate?** (If you're not familiar with the word "terminate" in relation to SSL, see this [short Wikipedia article](#).)

The answer is "you probably want to terminate SSL on your ELB, but this may not be sufficient."

If you can, I recommend terminating SSL on your ELB. It's relatively simple to setup, and you remove the CPU load of terminating SSL from your self-managed EC2 instances.

There are some scenarios where regulations require that every connection between every one of your servers is encrypted, even within your private network. In such cases, you may still use SSL termination on your ELB, but it may also make sense to just forward HTTPS traffic from your ELB to your next tier.

4.2.8 We Have a Common Set of Considerations for Each Tier

In addition to the inherent functionality we want from a tier, we also care about:

- **High Availability:** If one instance in the tier fails, or if an entire AZ fails, the tier should keep running.
- **Load Balancing:** Load should be distributed evenly across the instances we do have in a given tier.
- **Scaling:** Each tier should be capable of scaling up and down to meet demand
- **Backup & Disaster Recovery:** In the cloud, failure shouldn't be a crisis, just

a state we plan for. For any tier, we have to prepare for how we recover from a single instance failing.

- **Logging:** Log data that generated may be stored in a file by default, but it's best if we view it as a stream of events we can aggregate, search, and proactively monitor
- **Monitoring & Alerting:** If one or more of our instances run into issues, we want to be alerted so we can take action.

We'll cover the first 3 concepts below. We'll discuss the remaining concepts in Part 2 when we cover DevOps.

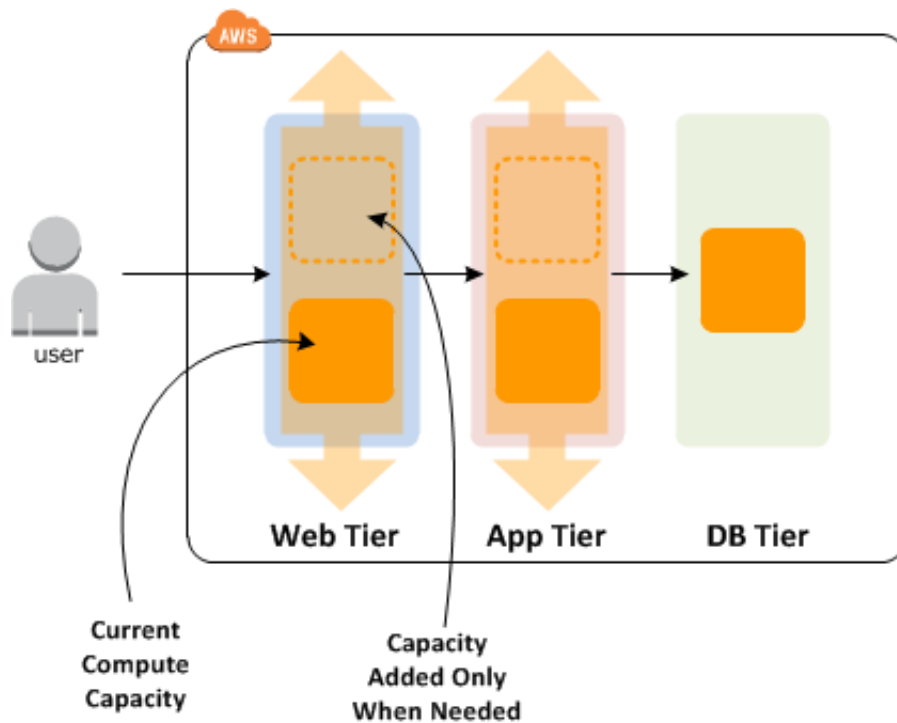
4.3 Architecting for Scalability

As we discussed earlier, you can always upgrade your EC2 instance type to "scale up", but eventually you will want to "scale out" by adding more EC2 instances to your infrastructure.

When it comes to scaling out, high-level presentations about AWS always trumpet the ability to dynamically scale your cluster of EC2 instances up or down depending on load. To actually achieve this, the most common solution is [Auto Scaling Groups](#).

Auto Scaling Groups can launch (or terminate) EC2 instances either according to a pre-set schedule (e.g. every day at 9am, launch 4 new instances), or based on metrics from AWS CloudWatch, such as when CPU utilization exceeds a certain threshold.

You create a unique **Auto Scaling Group (ASG)** for each tier of your app that you want to scale. This graphic from the [official AWS documentation](#) illustrates this nicely, showing one ASG for the Web Tier and one for the App Tier:



Automatically launching or terminating EC2 instances is one piece of the puzzle. The other is that your architecture must support dynamically adding/removing instances. There are two key considerations (and sometimes more depending on your architecture) to take into account here:

4.3.1 Service Discovery

In the illustration above, imagine that an ASG has just added a new EC2 instance to your App Tier. This means your Web Tier needs to know that this server exists before it can send any traffic to it! This is a key DevOps concept known as **Service Discovery**.

Hard-coding an IP address in your Web Tier configuration is the most basic form of Service Discovery and suffers from the problems of not being dynamically updated when new servers get added, and not auto-removing unhealthy instances. The classic way to handle Service Discovery for ASGs is using an **Elastic Load Balancer** as described in the [official documentation](#).

I'll talk in detail about your options for achieving Service Discovery in Part 2. For

now, note that you'll either have to leverage AWS built-in features for handling this, or implement it yourself.

4.3.2 Managing State

One of the most significant considerations for architecting in the cloud is state. Some parts of your app will be inherently stateful. For example, your persistence tier is designed to be stateful, to permanently store your data. Your caching tier is also stateful, but is meant to hold ephemeral data such as information about a user's session.

But everything else should be stateless. Any user request that comes in to your Web Tier should be agnostic about which instance in the Web Tier handles it, and agnostic about which instance in the App Tier handles it. Since these tiers are stateless, we can launch or terminate instances, and as soon as our Service Discovery mechanism routes traffic their way, they should "just work."

But how do you handle user sessions if the app tier is stateless? The most common solution here is to use a Cache Tier, as described earlier in the section "Application Layers." Caches store their data in memory (and in some cases can be it to disk for backup or scaling purposes) so they are very fast. Now all instances in your app tier just query the Cache or Database for state.

4.3.3 Scaling Your Database Tier

Of course, your Cache Tier and Database Tier need to scale, too, but since these are both stateful, we handle scaling these in a different way. We can't just "add servers."

First, let's discuss NoSQL databases. If you use AWS DynamoDB, scaling is as simple as specifying the provisioned throughput your app needs. More throughput costs more money, but AWS claims no limit as to what you can request.

Now let's discuss relational databases. Scaling relational databases is a deep and rich topic, the details of which are outside the scope of this guide. But I'll cover some highlights.

Basically, strategies for scaling the database amount to:

1. Using more powerful hardware, and
2. Reducing load on an individual database server in myriad ways.

4.3.3.1 Using More Powerful Hardware

Using more powerful hardware equates to using more powerful EC2 instances and faster I/O (i.e. more IOPS on EBS Volumes). Currently, the most powerful database instance -- the db.r3.8xlarge -- can give you 32 cores and 244 GB of RAM! It will also cost you about \$3,000/month per instance.

On the I/O side, you can pay more to use **EBS Provisioned IOPS**, which increases throughput and decreases latency to achieve higher database performance.. If you're managing your own EC2 instances for the Database Tier, you can even implement RAID across multiple EBS volumes to increase throughput.

AWS RDS makes it easy to upgrade the instance types using the API or web console. Recently, AWS introduced AWS RDS for Aurora, which an AWS-proprietary MySQL-compatible database (notably, not MySQL itself). Aurora is designed to eek more performance than MySQL out of the same hardware. This means you can get even more leverage out of "use more powerful hardware."

4.3.3.2 Reducing Load on an Individual Database Server

You can't just "reduce traffic" on your app, but you can certainly distribute and manage the load caused by that traffic in different ways. Here are some common options:

- **Read Replicas:** Instead of using one server for all database traffic, send all writes to one server and all reads to one or more "Read Replicas."
- **Shard:** Instead of using one server for all database traffic, partition your data set across multiple database servers. Each partition has its own master and may also leverage Read Replicas.
- **Use a Task & Queue Model:** By default, every call your code makes to the

database is synchronous, meaning it waits -- occupying a valuable database connection and the according resources -- until it gets a response. Instead of making 100% of database calls synchronous, your team could implement some calls to the database asynchronously by creating a "database processing task" and then adding it to a queue. This makes your overall database load more predictable, instead of having peaks and troughs of load.

- **Optimize the Data Itself:** You can de-normalize data to avoid expensive `JOIN`s, review existing indexes, implement partial indexes, etc.

4.4 Architecting for High Availability

If an individual EC2 instance or an entire AWS Availability Zone fails, your app should stay up. This is the essence of architecting for High Availability (HA).

As we've said, the key idea for achieving HA is eliminating a single point of failure by placing multiple EC2 instances in multiple Availability Zones. If you can afford it (in time and money) you can also consider a Multi-Region or even Multi-Cloud strategy. I'll limit this discussion to Multi-AZ.

When dealing with availability, an essential question is "do we just need to restore function, or do we also need to restore state?". Let's take a look at both.

4.4.1 High Availability for Your Stateless Tiers

For your *stateless* tiers, HA is straightforward:

- Each tier must have more than one EC2 instance
- Each tier must have EC2 instances in at least 2 Availability Zones (ideally more)
- When an EC2 instance fails, your architecture must be able to dynamically discover only healthy, active EC2 instances

To elaborate on the third point, this, again, is a discussion about Service Discovery.

To achieve effective Service Discovery, I recommend using an Elastic Load Balancer (ELB) in front of every stateless tier. This way, you direct all traffic to the ELB (which is itself designed to be High Availability) and the ELB then takes responsibility for Service Discovery, offering you flexibility on what it means for an EC2 instance to be unhealthy plus other configuration options.

The primary alternatives are to use a third-party Service Discovery tool like [consul](#), or to use a third-party load balancer like HAProxy. I'll discuss the Pro's and Con's of these approaches in Part 2 when we cover Service Discovery.

As a final side tip, you can use Auto Scaling Groups to monitor EC2 instance health and automatically terminate and re-launch unhealthy instances

4.4.2 High Availability for Your Stateful Tiers

For your *stateful* tiers, HA depends on the technology you've chosen and the AWS services you're using.

For the Cache Tier, if you use AWS ElastiCache, HA is managed for you. When configuring your cache cluster, you can select the number of nodes (for Memcached) or the number of Read Replicas (Redis). You can place these in multiple Availability Zones.

For the Database Tier, if you use DynamoDB, AWS automatically replicates DynamoDB data across three Availability Zones, all behind the scenes. Essentially, you get HA for free with this service.

If you use AWS RDS, RDS offers you the option of doing a Multi-AZ deployment. This means (quoting from AWS documentation) that:

RDS will maintain a synchronous standby replica in a different Availability Zone than the DB instance. Amazon RDS will automatically fail over to the standby in the case of a planned or unplanned outage of the primary.

RDS with Multi-AZ is a great option, and it's what I recommend to my clients, but it has some caveats:

- Amazon's Service-Level Agreement guarantees 99.95% availability, or about 20 minutes of downtime per month.
- It typically takes about 2 - 3 minutes for a failover to complete.
- Your app will use a DNS query to discover the correct database IP address, so once the failover completes and updates the private DNS record, your app's DNS cache has to expire (typically 3 minutes as well, but ultimately dependent on your app)
- There are (increasingly rare) [edge cases](#) where automatic failover itself failed

So, to be clear RDS with Multi-AZ does not mean zero-downtime if your primary database instance has an issue. But at least failover should all be automated.

If this is too much exposure to availability risk for your liking, your options are to use AWS RDS for Aurora for improved availability, use DynamoDB for improved availability, or to manage your own database and implement more responsive mechanisms for avoiding downtime.

4.6 Docker & Containers

Five years ago, there was an industry-wide paradigm shift as we went from physical servers to virtual machines (VMs). The idea that a single physical server could run many "virtual" servers made it possible to offer "Infrastructure as a Service" (IaaS), the ability to provision a new server by an API call instead of a phone call. You could say that AWS is a major vendor in the IaaS space.

Now in December 2014, another paradigm shift is already underway: Containers.

What if I told you that the Infrastructure-as-a-Service industry had a dirty secret that, on average, only 8 - 15% of a data center's deployed and active resources are being used ([Source](#))? That means you pay for 100% of the resources (i.e. CPU, memory, disk space), even if you use only 15% of them.

In addition, developers have realized that they no longer want to build applications for single servers. We want to build highly available, load balanced apps that are distributed across many VMs, each located in different Availability Zones. But packaging and distributing apps to do this can be hard. If you have a new app

how, do you easily deploy it to new servers?

These are some of the underlying reasons why Docker and the "containers" paradigm it represents have taken off so rapidly. Indeed, when I was at the AWS re:Invent conference in November 2014, most sessions on Docker were so crowded that the conference rooms reached their fire code limits and people (including me) were turned away.

4.6.1 Learning Docker

One of the consequences of Docker's meteoric rise has been an explosion in solution offerings for the "containerized world." Frankly, it's a confusing jungle out there and just understanding the ecosystem alone could warrant its own article. But here are some brief guidelines about where to get started.

First, you should learn about [Docker](#) itself. Docker represents the container paradigm and learning all about Docker will help you understand how to work with containers in general. Note that, recently, a Docker competitor has arisen called [Rocket](#). Rocket is very early (currently in 0.1), but it highlights that it's the containers concept that is profoundly important, more so than the particular implementation of it.

There are many great resources available for learning Docker. My favorites are this [3-Hour Video Overview](#) and the forthcoming [Docker in Action](#) book by Jeff Nickoloff.

4.6.2 Learning the Docker Ecosystem

Once you start working with Docker, you'll quickly discover that there are management tasks associated with containers. You want to deploy your containers across a cluster or "fleet" of servers, and you need tooling to help with that. As containers get deployed across a cluster, you have to figure out how to handle:

- Securing containers (especially in multitenant setups)
- Linking containers dynamically (i.e. Docker's version of Service Discovery)

- Monitoring containers and the cluster as a whole
- Handling container logging
- Auto scaling containers
- Auto healing when a VM/host dies
- Hosting a private container registry where your containers are stored

AWS recognized these challenges and released [EC2 Container Service \(ECS\)](#) to address some of them. ECS is designed to help you provision a cluster of EC2 instances and then deploy docker containers across the cluster. Watch the [official demo from AWS re:Invent](#). It's an exciting technology, but it's still a first-generation product and best practices here are not yet established.

But the more I've gotten into the container ecosystem, the more I see ECS as a "side story" in this exciting paradigm shift. One alternative to ECS is [CoreOS](#), which is a stripped-down version of Linux (so stripped down you can't even install new packages on it!) that comes with the `docker` software pre-installed. It includes a built-in service discovery tool (`etcd`) that runs across your cluster, and a built-in tool for deploying containers across your cluster (`fleet`). At least at this point, it seems more "vertically integrated" than ECS does.

Docker themselves has recently [thrown their hat in the ring](#) with Docker Machine, Docker Swarm, and Docker Compose. These also look like promising technologies, but Docker claims they won't be generally available until Q2-2015.

There are yet more offerings. [Apache Mesos](#) is an open source "distributed systems kernel" which aims to expose an API to allow developers to work at the abstraction level of a cluster rather than a single server. On Dec 7, 2014, the Mesosphere Datcenter Operating System was [announced](#). Interestingly, this tool can run on top of any Linux distribution, including CoreOS.

And just in case that's not enough for you, there's also [Dokku](#), a very lightweight program that enables you to push Heroku-compatible applications to your own private containers, [Flynn](#), an attempt offer fleet management as a Platform-as-a-Service offering, somewhat like a more "private Heroku" for your Docker containers, and many others.

4.6.3 Docker & AWS

As I mentioned, AWS offers its **EC2 Container Service**. Beyond that, AWS also supports deploying an application from a docker container using **AWS Elastic Beanstalk**, or bootstrapping the Docker software installation using **AWS CloudFormation**.

You can also of course just launch EC2 instances with an AMI that is ready to support Docker containers. CoreOS provides such an AMI. Or you can leverage a feature like [CloudInit](#), which is available for most Linux distributions and can run arbitrary scripts at loadup, including a script to install Docker immediately after the first load.

Best practices for running Docker in EC2 using, for example, CoreOS or Ubuntu, is outside the scope of this article, but it's absolutely something you should consider as you build your AWS app.

4.7 Security

If JP Morgan, Home Depot and Target can get hacked, so can you. I encourage you to take security very seriously and to think about it from Day 1. A rogue party with unauthorized access to your AWS account could [single-handedly destroy your company](#).

I'll list a **subset** of best practices, but this is not a complete list; it is merely a starting point.

4.7.1 Helpful Resources

Security is very difficult to get right. I'll describe mostly infrastructure-level security relevant to AWS, but of course your application itself needs to be secured, too. For app-level security guidelines, check out the very useful [OWASP Top 10 Security Vulnerabilities report](#).

For a good reference point on end-to-end security, check out the [PCI DSS 3.0 standard](#). Anyone who stores credit card data is subject to these requirements,

but most of them represent good security practices anyway.

And of course, AWS themselves publishes a White Paper on [Security Best Practices](#) that is a must read.

4.7.2 Never Login with Your Master Account

When you create your AWS account, you have created your master AWS account that has super-admin privileges. Like `root` on Linux, this account is so powerful, we want to avoid using it unless absolutely necessary.

For that reason, immediately after creating your master account, create a set of **IAM Groups** that represent the different permission levels you want to assign on your team (e.g. Admin, Dev, Test, Read-Only) and then create **IAM Users** for each member of your team. You can assign IAM Users to IAM Groups to give them the right permissions.

You should get ready to lock your master account username & password in a safe and distribute the keys only to the most trusted parties, but before you do...

4.7.3 Setup Multi-Factor Authentication & Require Strong Passwords

If your AWS login password is compromised, you want a fail-safe. For that reason, I recommend setting up Multi-Factor Authentication (MFA) for your master account and mandating it for all IAM Users.

In addition, IAM lets you set password policies for IAM Users. You should enforce long and difficult passwords that would make a brute-force attack computationally infeasible (even though Amazon protects against this already).

But even MFA won't protect you if you fail to address the next point.

4.7.4 Never Commit Your AWS API Access Keys to Source Code

[This person's experience](#) shows that publishing AWS credentials to a GitHub repo is the equivalent of publishing your username and password to the repo.

Actually, credentials are a special case of "configuration information" and in general, you want to keep configuration information out of code. [Adam Wiggins explains this in more detail on 12factor.net](#). Instead, store passwords in Environment Variables. Some people argue in favor of passing config values through command-line parameters, but I worry that this then includes the password in your `bash` command history.

Also, you should never generate API Access Keys for your master account. Instead, only generate API keys for your IAM accounts. You may wish to create one IAM user for each third-party service that needs API access so you can explicitly keep track of how API keys are used and assign minimal permissions.

Think about it, if you use a third-party vendor to send data to S3, when you provide them with your API keys, you have no control over how they manage your secrets. If your vendor screws up, you want to limit the damage that can be done.

4.7.5 Use IAM Roles instead of API Keys for EC2 Instances

Many AWS users will generate an API key and then pass it directly to the EC2 instance via environment variables, or worse, hard coding it.

AWS has a great tool to enable these permissions in a much cleaner way. You can create an **IAM Role** and assign it to a specific EC2 instance. You can then give that IAM Role permissions to access a particular S3 folder, or virtually any other AWS resource. When you do this, AWS will automatically populate Environment Variables on your EC2 instance with temporary API credentials.

This is the most secure way to enable access to AWS resources because there's simply no key to manage! Note that you can only assign an IAM Role to an EC2 instance when launching it.

4.7.6 Centrally Manage Employee Access

Ideally, all employee identity is managed in a single central place. For example, you might have a company-wide Active Directory server, or a centrally hosted LDAP server, or even use a third-party service like [OneLogin](#).

In a Linux environment, you can configure your servers to only permit login from certain groups of users as listed in the LDAP server. This has the benefit of enabling audit logs for every single login by any employee. It also means that there are no root SSH keys for servers floating around.

But central identity servers are sometimes not priority #1 when building a new app, especially with a smaller team of trusted people. So this may be a "Phase 2" security measure.

4.7.7 Have a Policy for Employees Who Leave

If you have a trusted employee who leaves, you should know exactly which resources that person had access to so that you can reset any passwords as needed. Ideally, an employee who leaves would amount to nothing more than revoking his SSH Key as stored on the LDAP Server and deactivating his IAM User Account. Of course, many third-party services that dev teams use don't always offer individual accounts, so you may need to reset multiple third-party accounts as well.

4.7.8 Use a Bastion Host or Connect to Your VPC with a VPN

Ideally, your infrastructure is automated enough that you rarely need to directly login to a server. But when you must login, it's too risky to open login ports (22 for Linux SSH, 3389 for Windows Remote Desktop) directly to the public Internet.

One option is to lock down each server's ports to the specific IP address logging in, but since you will manage many servers, this quickly becomes cumbersome to maintain.

A better option is to either lock out these ports completely from the Public Internet and connect with via a VPN, or to use a **Bastion Host**. A Bastion Host (sometimes called "Jump Box") is a single server that does permit login from the public Internet, but it is the *only* such server in your network that permits direct login.

This allows you to harden a single server, versus every server. Some examples of

extra hardening on this server include permitting access only from specified IP addresses and even setting up two-factor authentication with tools like [Duo](#).

There is a good discussion on Hacker News about the [relative merits of the Bastion Host vs the VPN](#), as well as some ideas for further hardening the Bastion Host.

4.7.9 Pay Attention to Key Management

In my experience, most developers consider encryption and key management a nuisance and an afterthought. In fact, there is a whole body of knowledge on proper key management. One of my favorite resources is the [OWASP Cryptographic Storage Key Sheet](#).

You'll notice there's a great deal of discussion on where and how to store keys. Until recently, there were few good options for this in AWS. Essentially, you could store keys in S3 (hopefully encrypted, hopefully locked down by IAM permissions), or you could spend thousands of dollars (and man-hours) using [AWS CloudHSM](#), which is impractical for many companies.

So in November 2014, AWS released [AWS Key Management Service](#). This service is a perfect complement to the OWASP recommendations and provides a centralized place to manage all your keys. Of course, you'll still need to implement best practices in your app itself.

4.7.10 Storing Passwords and Secrets & Encrypting Data at Rest

As you build your app, you will accumulate secrets. This includes your database password, symmetric encryption keys, asymmetric private keys, credentials used to access third-party services, and generally any other piece of information that, if known, grants access to sensitive data.

So where do we store all these secrets? Your first thought might have been to encrypt the secrets themselves, but then where do you store the master encryption key? Ultimately, important secrets have to be stored somewhere.

It turns out that where you store the keys and how you secure them is the essential part of encrypting data at rest. Once the keys themselves are secured, you have many options of where to store the actual data and how to secure it, whether on S3, in your own EC2 instance, etc.

This is again, a much bigger topic than I can cover here, and I encourage you to review the AWS White Paper [Securing Data at Rest with Encryption](#).

4.7.11 Encrypting Data in Transit

When users use your app, you want to ensure that (a) no one can see their traffic en route, (b) their message to you has not been altered in transit, even by a party who can't read it, and (c) no one is doing a man-in-the-middle attack pretending to be you to the user (and pretending to be the user to you!). In short, we care about **privacy**, **integrity**, and **authenticity**.

The solution to these problems is simple: Use SSL/TLS at all times unless you can specifically guarantee you are within a trusted network (e.g. from your Web Tier to App Tier). But, of course, there are nuances and edge cases to consider here, especially when dealing with peer-to-peer connections.

I will defer again to the [AWS Security Best Practices](#) White Paper.

4.7.12 Additional Security Measures

I can't possibly cover all of security in an article not even dedicated to it. In addition to what I've listed above, you should consider setting egress rules on your Security Groups (after all there's no reason ever that your database should be connecting to IP addresses in Russia), setting up vulnerability scanning, using Host-Based Intrusion Detection (HIDS) systems, and much more.

5. Summary

Congratulations on making it to the end of this guide!

I want to emphasize that there are many, many important AWS technologies and

techniques we did not cover. AWS is a vast terrain that took years to build and consequently takes time to learn.

But we covered a lot of ground and you now have a mental framework within which to incorporate new AWS services and techniques you discover. More importantly, you have the foundation to build and scale your app on AWS.