Implementing Cloud Design Patterns for AWS

Whether you are just getting your feet wet in cloud infrastructure or already creating complex systems, this book aims at describing patterns that can be used to fit your system needs.

The initial patterns will cover some basic processes such as maintaining and storing backups as well as handling redundancy. The book will then take you through patterns of high availability. Following this, the book will discuss patterns for processing static and dynamic data and patterns for uploading data. The book will then dive into patterns for databases and data processing. In the final leg of your journey, you will get to grips with advanced patterns on Operations and Networking and also get acquainted with Throw-away Environments.

Who this book is written for

This book is aimed at architects, solution providers, and those of the DevOps community who are looking to implement repeatable patterns for deploying and maintaining services in the Amazon cloud infrastructure. Prior experience using AWS is required as the book focuses more on the patterns and not on the basics of using AWS.

What you will learn from this book

- Create and maintain server backups
- Implement scaling policies on schedules, influx in traffic, and deep health checks
- Provision servers and data that persist through termination
- Make complete use of high availability storage and redundancy storage
- Design content delivery networks to improve user experience
- Optimize databases through caching and sharding
- Monitor and queue data for processing

Create highly efficient design patterns for scalability, redundancy, and high availability in the AWS Cloud
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Introduction'
- A synopsis of the book’s content
- More information on Implementing Cloud Design Patterns for AWS

About the Author

Marcus Young recently graduated with a degree in computer science and mathematics before getting involved in system administration and DevOps. He currently works in software automation using open source tools and technologies. His hobbies include playing ice hockey and brewing homebrew beer. He also enjoys hardware projects based on microcontrollers and single board computers.
Implementing Cloud Design Patterns for AWS

Amazon Web Services (AWS) is arguably the most cutting-edge Cloud provider currently available. In the past, data centers were massive entities that often required days to provide resources for applications. With AWS, this barrier is nonexistent. Applications can be scaled almost instantly. Metrics can be gathered with little or no configuration. Moving into the Cloud, however, might not be easy.

This book will act as a small reference guide, with detailed implementation examples, to show how (and how not) to design your applications in a way that makes them tolerant of underlying hardware failures, resilient against an unexpected influx of data, and easy to manage and replicate. You will be able to see both the benefits and limitations of the current services available to you from the AWS infrastructure.

What This Book Covers

Chapter 1, Introduction, introduces you to AWS and the problems encountered when deploying and maintaining applications in the Cloud. Problems include upgrading databases, data replication, cache issues, and zero downtime SLAs.

Chapter 2, Basic Patterns, demonstrates some examples of basic patterns such as scaling instances, dynamic disk allocation, and more.

Chapter 3, Patterns for High Availability, demonstrates some examples of patterns for highly available services such as data center replication, floating IP address allocation, health checking, and more.

Chapter 4, Patterns for Processing Static Data, demonstrates some examples of patterns for static data such as cache distribution, direct hosting, web storage hosting, and more.

Chapter 5, Patterns for Processing Dynamic Data, demonstrates some examples of patterns for dynamic data such as state sharing, URL rewriting, rewrite/cache proxying, and more.

Chapter 6, Patterns for Uploading Data, provides some examples of patterns and solutions for object uploading, storage indexing, and write proxying.

Chapter 7, Patterns for Databases, provides some examples of patterns for data replication, in-memory caching, and sharding.

Chapter 8, Patterns for Data Processing, provides some examples of patterns for batch processing issues such as queuing chains, priority queues, and job observers.
Chapter 9, *Patterns for Operation and Maintenance*, provides some examples of patterns for server swapping, startup settings, backup patterns, and others.

Chapter 10, *Patterns for Networking*, provides some examples of patterns for multiload balancers, operational and functional firewalls, and on-demand NAT networking.

Chapter 11, *Throw-away Environments*, is the closing chapter and provides some examples of third-party tools such as CloudFormation, Terraform, and so on, which aid in infrastructure design.
The paradigm for development of applications has shifted in many ways over the years. Instead of just developing pure applications, aimed at specific system configurations, the trend has moved towards web applications. These applications present a very different set of challenges not just for the developers, but also for the people who manage the systems that host them. The reaction to this need to build, test, and manage such web applications has been to develop an abstraction on top of the hardware that allows for the ability to bring up entire virtualized environments quickly and consistently.

Throughout these chapters, you will learn basic design principles for applications and known issues. These may not be completely compatible with all application types but should serve as a basic toolkit for bigger design patterns. It is also very important to note that AWS adds new services all the time, some of which by default solve these design patterns at the time of writing. If your software or services handle sensitive data and have in-flight or at-rest requirements, be very careful to read how each individual AWS-provided service handles data.

The topics that are covered in this chapter are:

- Introduction to AWS
- Cloud computing service models
- Benefits of moving to the Cloud
- Problems encountered with AWS
Introduction to AWS

Amazon Web Services (AWS) is a very large suite of Cloud services provided by Amazon. AWS provides, at a base level, virtual machines and the services surrounding them. Many Cloud-based virtual machine services such as Google Compute Engine, DigitalOcean, Rackspace, Windows Azure, and so on provide the ability to bring up a machine from a supported base operating system image or snapshot, and it’s up to the user to customize it further.

Amazon has made itself one of the leaders for Cloud-hosting by providing not just virtual machines, but configurable services and software implementations of hardware found in data centers. For most large-scale systems, the move to Cloud infrastructure brings to the table a huge set of questions on how to handle issues such as load balancing, content delivery networks, failover, and replication. The AWS suite can handle the same issues that a physical data center can, usually for a fraction of the cost. They can get rid of some of the red tape that comes with a data center such as requesting provisioning, repairs, and scheduling downtime.

Amazon is constantly offering new services to tackle new and unique problems encountered with Cloud infrastructure. However, this book may not cover every service offered by Amazon. The services that this book will cover include:

- Computing and networking
  - Elastic Cloud Compute (EC2) virtual machines
  - Route 53 DNS provides local and global DNS look-ups
  - Virtual Private Cloud (VPC) isolated Cloud networks provide internal networks
  - Elastic Load Balancers (ELB) automatically distribute traffic across EC2 instances
  - Auto Scaling Groups (ASG) provide a way to scale instances up and down based on schedules or metrics gathered via CloudWatch from the EC2 instances attached to them

- Database
  - SimpleDB is a highly scalable NoSQL database
  - Relational Database Service (RDS) is a scalable SQL database apart from MySQL, Oracle, PostgreSQL, or SQL Server
  - ElastiCache is an in-memory cache on top of Redis or MemCached
• Storage and content delivery
  ° **Simple Storage Service** (S3) is a distributed storage network that crosses data center boundaries with built-in redundancy
  ° CloudFront is a CDN that distributes content based on latency or location

• Application services
  ° **Simple Queue Service** (SQS) is a fast, reliable, scalable, and fully managed message queuing service

• Deployment and management
  ° CloudFormation is a service that allows the provisioning and updating of AWS resources through templates, usually JSON

• Logging
  ° CloudWatch can monitor, display, and alert on instance metrics and logs

For information on other services provided by AWS that are not relevant to the information in this book visit http://aws.amazon.com/products/.

### Cloud computing service models
AWS falls under the category of Cloud computing called Infrastructure as a Service. In Cloud computing there are three service models:

• **Infrastructure as a Service (IaaS)**
• **Platform as a Service (PaaS)**
• **Software as a Service (SaaS)**

### Infrastructure as a Service
IaaS can be described as a service that provides virtual abstractions for hardware, servers, and networking components. The service provider owns all the equipment and is responsible for its housing, running, and maintenance. In this case, AWS provides APIs, SDKs, and a UI for creating and modifying virtual machines, their network components, routers, gateways, subnets, load balancers, and much more. Where a user with a physical data center would incur charges for the hardware, shelving, and access, this is removed by IaaS with a payment model that is per-hour (or per-use) type.
Platform as a Service
While AWS itself is an IaaS provider, it contains a product named ElasticBeanstalk, which falls under the PaaS category for Cloud models. PaaS is described as the delivery of a computing platform, typically an operating system, programming language execution environment, database, or web server. With ElasticBeanstalk, a user can easily turn a code into a running environment without having to worry about any of the pieces underneath such as setting up and maintaining the database, web server, or code runtime versions. It also allows it to be scaled without having to do anything other than define scale policies through the configuration.

Software as a Service
AWS also provides a marketplace where a user can purchase official and third-party operating system images that provide configurable services such as databases, web applications, and more. This type of service falls under the SaaS model. The best interpretation for the SaaS model is on-demand software, meaning that the user need only configure the software to use and interact with it. The draw to SaaS is that there is no need to learn how to configure and deploy the software to get it working in a larger stack and generally the charges are per usage-hour.

The AWS suite is both impressive and unique in that it doesn't fall under any one of the Cloud service models as described previously. Until AWS made its name, the need to virtualize an entire environment or stack was usually not an easy task and consisted of a collection of different providers, each solving a specific part of the deployment puzzle. The cost of using many different providers to create a virtual stack might not be cheaper than the initial hardware cost for moving equipment into a data center. Besides the cost of the providers themselves, having multiple providers also created the problem of scaling in one area and notifying another of the changes. While making applications more resilient and scalable, this Frankenstein method usually did not simplify the problem as a whole.

Benefits of moving to the Cloud
There are many different answers to why moving to a Cloud-hosted environment might be beneficial but it depends on the end user. The shift may suit small teams but for mid-sized teams the effort saved begins to outweigh the cost. I start at mid-sized because this is the size that usually includes the two teams that benefit the most:

- The developers and testers
- Operations
For a developer, the biggest benefit of Cloud providers is the ability to throw away entire environments. In a traditional developer setting, the developers usually develop their code locally, have access to a shared physical server, or have access to a virtual server of some type. Issues that usually arise out of these setups are that it’s hard to enforce consistency and the servers can become stale over time. If each developer works locally, inconsistency can arise very quickly. Different versions of the core language or software could be used and might behave differently from machine to machine. One developer might use Windows and prefer registry look-ups while another developer may use Mac and prefer environment variables.

If the developers share a core server for development, other issues may arise such as permissions or possibly trying to modify services independent of each other causing race conditions. No matter what problems exist, known or unknown, they could be solved by always starting from the same base operating system state.

The leading software for solving this issue is Vagrant. Vagrant provides the ability to spin up and destroy a virtual machine from a configuration file along with a configuration management suite such as Puppet, Chef, Docker, or Ansible. Vagrant itself is agnostic to the Cloud hosting tool in the sense that it does not require AWS. It can spin up instances at AWS given the credentials, but it can also spin up virtual machines locally from VirtualBox and VMWare.

Vagrant gives back consistency to the developers in the sense that it takes a base box (in AWS terms this is an Amazon Machine Image or AMI) and configures it via one of the configuration suites or shell to create a running virtual machine every time it is needed. If all the developers share the same configuration file then all of them are mostly guaranteed to work against the same environment. That environment can be destroyed just as easily as it was created, giving the resources back and incurring no charges until needed again.

The bringing up and destroying of the instances becomes a small invisible piece of their workflow. By virtue of enforcing a strategy like this on a team, a lot of issues can be found and resolved before they make their way up the chain to the testing or production environments.

More information on Vagrant can be found at http://www.vagrantup.com.

The other team I mentioned that benefits from moving to the Cloud is the operations team. This team differs greatly in responsibility from company to company but it is safe to assume that the team is heavily involved with monitoring the applications and systems for issues and possible optimizations. AWS provides enough infrastructure for monitoring and acting on metrics and an entire book could be dedicated to the topic. However, I’ll focus only on auto scaling groups and CloudWatch.
For AWS, an auto scaling group defines scaling policies for instances based on schedules, custom metrics, or base metrics such as disk usage, CPU utilization, memory usage, and so on. An auto scaling group can act on these thresholds and scale up or down depending on how they are configured. In a non-Cloud environment this same setup takes quite a bit of effort and depends on the software whereas, it's a core concept to AWS.

Auto scaling groups also automatically register instances with a load balancer and shift them into a round robin distribution. For an operations team, the benefit of moving to Amazon might justify itself only to alleviate all the work involved in duplicating this functionality elsewhere, allowing the team to focus on creating deeper and more meaningful system health checks.

Throw-away environments can also be beneficial to the operations teams. A sibling product to Vagrant, very recently released, is Terraform. Terraform, like Vagrant, is agnostic to the hosting environment but does not solely spin up virtual instances. Terraform is similar to CloudFormation in the sense that its goal is to take a central configuration file, which describes all the resources it needs. It then maps them into a dependency graph, optimizes, and creates an entire stack. A common example for Terraform would be to create a production environment from a few virtual machines, load balancers, Route53 DNS entries, and set auto scaling policies. This flexibility would be nearly impossible in traditional hardware settings and provides an on-demand mentality not just for the base application, but also for the entire infrastructure, leading to a more agile core.

More information on Terraform can be found at http://www.terraform.io.

Common problems encountered at AWS
The previous sections have tried to make light of issues found in traditional settings, which might make moving to a Cloud infrastructure seem like a logical choice with no ramifications. But this is not true. While Cloud infrastructure aims to resolve many problems, it does bring up new issues to the user.

Underlying hardware failures
Some issues can be avoided while others may not. Some examples of issues that may not be avoided, other than user error, are underlying hardware issues that propagate themselves to the user. Hardware has a fail rate and can be guaranteed to fail at some point while the benefit of IaaS is that, even though the hardware is abstracted away, it is still a relevant topic to anyone using it.
AWS has a **Service Level Agreement (SLA)** policy for each service, which guarantees that at their end you will meet a certain percentage of uptime. This implies a certain amount of downtime for scheduled maintenance and repairs of the hardware underneath.

As an AWS user this means you can expect an e-mail at some point during usage warning about instances being stopped and the need to restart manually. While this is no different from a physical environment where the user schedules their own downtime, it does mean that instances can misbehave when the hardware is starting to fail. Most of the replication and failover is handled underneath but if the application is real-time and is stopped, it does mean that you, as a user, should have policies in place to handle this situation.

**Over-provisioning**

Another issue with having virtual machines in the Cloud is over-provisioning. An instance type is selected when an instance is launched that corresponds to the virtualized hardware required for it. Without taking measures to ensure that replication or scaling happens on multiple data centers, there is a very real risk that when a new instance is needed, the hardware will not be immediately available. If scaling policies are in effect that specify your application should scale out to a certain number of instances, but all of those instances are in a data center nearing its maximum capacity, the scaling policy will fail. This failure negates having a scaling policy in place if it cannot always scale to the size required.

**Under-provisioning**

A topic that is rarely talked about but is very common is under-provisioning and it is the opposite of over-provisioning. We will start with an example: say we purchase a server for hosting a database and purchase the smallest instance type possible. Let's assume that for the next few days this is the only machine running in a specific rack in the AWS data center. We are promised the resources of the instance we purchased but as the hardware is free, it gives us a boost in performance that we get accustomed to unknowingly.

After a few days, the hardware that has been provisioned for other customers, now gives us the resources we were promised and not the extra boost we got for free in the background. While monitoring we now see a performance degradation! While this database was originally able to perform so many transactions per second it now does much less. The problem here is that we grew accustomed to the processing power that technically was not ours and now our database does not perform the way we expected it to.
Introduction

Perhaps the promised amount is not suitable but it is live and has customer data within it. To resolve this, we must terminate the instance and change the instance type to something more powerful, which could have downstream effects or even full downtime to the customer. This is the danger of under-provisioning and it is extremely hard to trace. Not knowing what kind of a performance we should actually get (as promised in the SLA) causes us to possibly affect the customer, which is never ideal.

Replication

The previous examples are extreme and rarely encountered. For example, in a traditional hosting environment where there are multiple applications behind a load balancer, replication is not trivial. Replication of this application server requires registration with the load balancer and is usually done manually or requires configuration each time. AWS-provided ELBs are a first-class entity just like the virtual machines themselves. The registration between this is abstracted and can be done with the click of a button or automatically through auto scaling groups and start-up scripts.

Redundancy

Apart from replication, redundancy is another hot topic. Most database clustering takes redundancy into effect but requires special configuration and initial setup. The RDS allows replication to be specified at the time of setup and guarantees redundancy and uptime as per its SLA. Their Multi-AZ specification guarantees that the replication crosses availability zones and provides automatic failover. Besides replication, special software or configuration is needed to store offsite backups. With S3, an instance may synchronize with an S3 bucket. S3 is itself a redundant storage that crosses data center sites and can be done via an AWS CLI or their provided API. S3 is also a first-class entity so permissions can be provided transparently to virtual machines.

The previous database example hints towards a set of issues deemed high availability. The purpose of high availability is to mitigate redundancy through a load balancer, proxy, or crossing availability zones. This is a part of risk management and disaster recovery. The last thing an operations team would want is to have their database go down and be replicated to New Orleans during Hurricane Katrina. This is an extreme and incredibly rare example but the risk exists. The reason that disaster recovery exists and will always exist is the simple fact that accidents happen and have happened when ill-prepared.
Improving the end user experience

Another set of problems to be solved is optimization to the end user. Optimization in this case is proxying through cache servers so that high workloads can be handled without spinning up more instances. In a scaling policy, high bandwidth would lead to more instances, which incur cost and startup time. Caching static content, where possible, can help mitigate high bandwidth peaks. Other ways to optimize without caching might be to use Content Delivery Networks (CDNs) such as the AWS-provided CloudFront service, which automatically choose servers geographically close to the user.

Monitoring and log-gathering

The last set of problems to discuss in small detail is operational in nature. Most applications generate logs and large software stacks with many disparate logs. Third-party software such as Loggly and Splunk exist to aggregate and search log collections but AWS has services dedicated to this as well. The preferred way is to upload logs to CloudWatch. CloudWatch allows you to directly search and create alerts on the data within logs. Since CloudWatch is a first-class AWS service, they provide an SLA similar to the instance itself and the storage is scalable.

These are only some of the issues that someone shifting into AWS might encounter or need to fortify their infrastructure against. Reading through the chapters of this book will serve as a beginner’s guide of sorts to help create a resilient infrastructure against these issues and others.

Summary

Throughout this brief introduction to AWS, we learned not only the background and industry shift into virtualized infrastructure, but also where AWS fits in with some competitors. We not only discussed the kinds of problems AWS solves, but also the kinds of problems that can be encountered in Cloud infrastructure. There are countless unique processes to be solved with this dynamic paravirtual environment. Picking up consistent patterns throughout this book will help to strengthen applications of many forms against these issues. In the next chapter, we will go over some basic design patterns. It is one of the easier topics and covers data backups through instance snapshots, replication through machine imaging, scaling instance types, dynamic scaling through CloudWatch, and increasing the disk size when needed. These patterns help solve common provisioning issues for single instances.
Where to buy this book

You can buy Implementing Cloud Design Patterns for AWS from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.