Author Picks

FREE

Exploring Cloud Computing

Chapters edited by
Michael Wittig and Andreas Wittig

manning

Exploring Cloud Computing

Selected by Michael Wittig and Andreas Wittig

Manning Author Picks

For online information and ordering of these and other Manning books, please visit www.manning.com. The publisher offers discounts on these books when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

Manning Publications Co.
20 Baldwin Road Technical
PO Box 761
Shelter Island, NY 11964

Cover designer: Leslie Haimes

ISBN 9781617294877
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 - EBM - 21 20 19 18 17 16

contents

introduction

Cloud Computing is enabling many trends in IT today: microservices, pay-as-you-go pricing, and serverless architectures, to name three. The biggest player in the market's Amazon, who offers the most mature cloud systems through their Amazon Web Services (AWS). Other companies are gaining traction as well: Google invests into their Google Cloud Platform, and Microsoft entered the game with Azure.

One key tenet of microservices is the ability to replace them easily with other implementations, but that ability also requires that your infrastructure isn't fixed. If you add and remove microservices every few weeks, you need a dynamic infrastructure, and that dynamic infrastructure's exactly what you get in the cloud.

Using the cloud is like consuming power. At the end of the month you get a bill for the resources you've consumed. You don't need to buy resources upfront, and you can stop using them whenever you want. There's no capacity planning and no procurement process anymore, which enables you to create SaaS offerings for your customers with the same pricing model.

The latest trend created by AWS is serverless computing. You can now run code in the cloud without managing the underlying operating systems and execution platforms. You upload your code and it's executed in the cloud. You pay only for the time the function executes. No more idle resources!

The Cloud at Your Service

Cloud Computing is a business changer. Today you can start a new business that's able to grow with minimal capital. The cloud provides shared and virtualized resources to use the hardware better than you could personally. Therefore, prices go down. Cloud environments can be fully automated because you can programmatically talk to your infrastructure using APIs. Chapter 1 of *The Cloud at Your Services* explains what Cloud Computing is about.

Chapter 1 from *The Cloud at Your Service* by Jothy Rosenberg and Arthur Mateos

What is cloud computing?

This chapter covers

- Defining the five main principles of cloud computing
- Benefiting from moving to the cloud
- How evolving IT led to cloud computing
- Discussing the different layers (types) of clouds

Cloud computing is the hottest buzzword in the IT world right now. Let's understand why this is and what this cloud computing hype is all about. A growing consensus among cloud vendors, analysts, and users defines cloud computing at the highest level as computing services offered by a third party, available for use when needed, that can be scaled dynamically in response to changing needs. Cloud computing represents a departure from the norm of developing, operating, and managing IT systems. From the economic perspective, not only does adoption of cloud computing have the potential of providing enormous economic benefit, but it also provides much greater flexibility and agility. We'll continue to refine and expand our

definition of cloud computing as well as your understanding of its costs and benefits throughout this book.

Not only are IT journals and IT conferences writing and talking about cloud computing, but even mainstream business magazines and the mass media are caught up in its storm. It may win the prize for the most over-hyped concept IT has ever had. Other terms in this over-hyped category include Service-Oriented Architectures (SOA), application service providers, and artificial intelligence, to name a few. Because this book is about cloud computing, we need to define it at a much more detailed level. You need to fully understand its pros and cons, and when it makes sense to adopt it, all of which we'll explain in this chapter. We hope to cut through the hype; and to do that we won't merely repeat what you've been hearing but will instead give you a framework to understand what the concept is all about and why it really is important.

You may wonder what is driving this cloud hype. And it would be easy to blame analysts and other prognosticators trying to promote their services, or vendors trying to play up their capabilities to demonstrate their thought leadership in the market, or authors trying to sell new books. But that would ignore a good deal of what is legitimately fueling the cloud mania. All of the great expectations for it are based on the facts on the ground.

Software developers around the world are beginning to use cloud services. In the first 18 months that it was open for use, the first public cloud offering from Amazon attracted over 500,000 customers. This isn't hype; these are facts. As figure 1.1 from Amazon's website shows, the bandwidth consumed by the company's cloud has quickly eclipsed that used by their online store. As the old adage goes, "where there's smoke, there must be a fire," and clearly something is driving the rapid uptake in usage from a cold start in mid-2006.

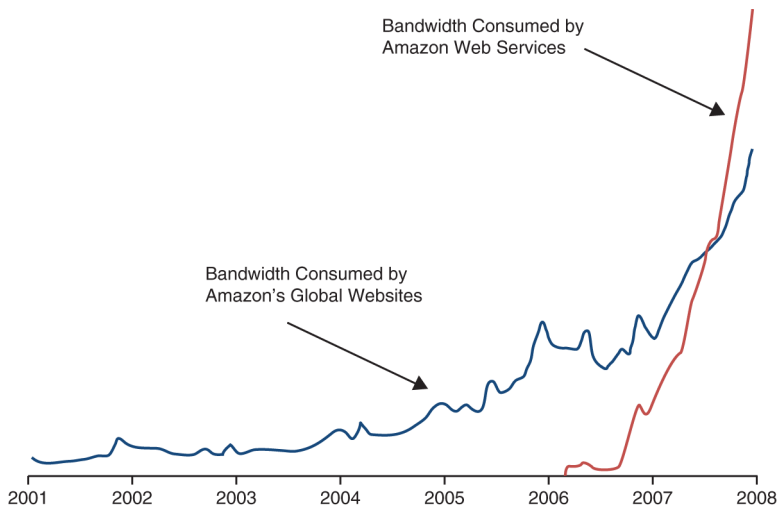


Figure 1.1 Amazon originally deployed a large IT infrastructure to support its global e-commerce platform. In less than 18 months after making the platform available as a cloud service to external users, its usage, as measured by amount of bandwidth consumed, outstripped bandwidth used internally.

Similar to the previous technology shifts—such as the move from mainframes to client-server, and then from client-server to the internet—cloud computing will have major implications on the business of IT. We hope to provide you with the background and perspective to understand how it can be effectively used as a component of your overall IT portfolio.

We'll begin by expanding on our earlier definition of cloud computing in terms of its five main principles.

1.1 Five main principles that define cloud computing

We can summarize the five main principles of cloud computing as follows:

- Pooled computing resources available to any subscribing users
- Virtualized computing resources to maximize hardware utilization
- Elastic scaling up or down according to need
- Automated creation of new virtual machines or deletion of existing ones
- Resource usage billed only as used

We assert, with very few notable exceptions called out later, that these five main principles are necessary components to call something *cloud computing*. They're summarized in table 1.1 with a brief explanation of each one for quick reference.

Table 1.1 The five main principles of cloud computing

Resource	Explanation
Pooled resources	Available to any subscribing users
Virtualization	High utilization of hardware assets
Elasticity	Dynamic scale without CAPEX
Automation	Build, deploy, configure, provision, and move, all without manual intervention
Metered billing	Per-usage business model; pay only for what you use

We'll now discuss these principles in concrete terms, making sure you understand what each one means and why it's a pillar of cloud computing.

1.1.1 Pooled computing resources

The first characteristic of cloud computing is that it utilizes pooled computing assets that may be externally purchased and controlled or may instead be internal resources that are pooled and not dedicated. We further qualify these pooled computing resources as contributing to a cloud if these resources are available to any subscribing users. This means that *anyone* with a credit card can subscribe.

If we consider a corporate website example, three basic operational deployment options are commonly employed today. The first option is the self-hosting option. Here,

companies choose not to run their own data center and instead have a third party lease them a server that the third party manages. Usually, managed hosting services lease corporate clients a dedicated server that isn't shared (but shared hosting is common as well). On this single principle, cloud computing acts like a *shared managed hosting service* because the cloud provider is a third party that owns and manages the physical computing resources which are shared with other users, but there the similarity ends.

Independent of cloud computing, a shift from self-hosted IT to outsourced IT resources has been underway for years. This has important economic implications. The two primary implications are a shift of capital expenses (CAPEX) to operational expenses (OPEX), and the potential reduction in OPEX associated with operating the infrastructure. The shift from CAPEX to OPEX means a lowering of the financial barrier for the initiation of a new project. (See the definition in section 3.1.)

In the self-hosted model, companies have to allocate a budget to be spent up front for the purchase of hardware and software licenses. This is a fixed cost regardless of whether the project is successful. In an outsourced model (managed hosting), the startup fees are typically equivalent to one month's operational cost, and you must commit to one year of costs up front. Typically, the one-year cost is roughly the same or slightly lower than the CAPEX cost for an equivalent project, but this is offset by the reduced OPEX required to operate the infrastructure. In sharp contrast, in a cloud model, there are typically no initial startup fees. In fact, you can sign up, authorize a credit card, and start using cloud services literally in less time than it would take to read this chapter. Figure 1.2 showcases side by side the various application deployment models with their respective CAPEX and OPEX sizes.

The drastic difference in economics that you see between the hosting models and the cloud is due to the fact that the cost structures for cloud infrastructures are vastly better than those found in other models. The reasons for the economies of scale are severalfold, but the primary drivers are related to the simple economics of volume. Walmart and Costco can buy consumer goods at a price point much lower than you or I could because of their bulk purchases. In the world of computing, the "goods" are computing, storage, power, and network capacity.

1.1.2 Virtualization of compute resources

The second of the five main principles of cloud computing has to do with virtualization of compute resources. Virtualization is nothing new. Most enterprises have been shifting much of their physical compute infrastructure to virtualized for the past 5 to 10 years. Virtualization is vital to the cloud because the

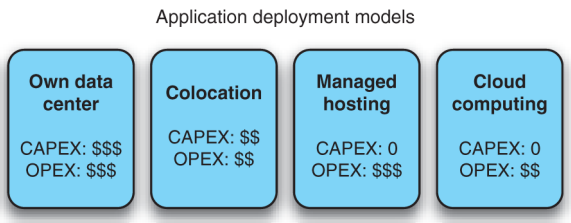


Figure 1.2 IT organizations have several alternatives for hosting applications. The choice of deployment model has different implications for the amount of CAPEX (up-front capital expenditure) and OPEX (ongoing operational costs). The number of $ signs represent the relative level of CAPEX and OPEX involved with the choice of deployment model.

scale of cloud infrastructures has to be enormous, based on thousands of servers. Each server takes up physical space and uses significant power and cooling. Getting high utilization out of each and every server is vital to be cost effective.

The recent technological breakthrough that enabled high utilization on commodity hardware—and which is the single biggest factor behind the cloud being a recent IT phenomenon—is virtualization where each physical server is partitioned into many virtual servers. Each one acts like a real server that can run an operating system and a full complement of applications.[1] Virtualized servers are the primary units that can be consumed as needed in the cloud. These virtualized servers constitute a large pool of resources available when required. But having such a large pool will work only if applications can use more or less of the pool as demands placed on the applications grow and shrink. As you'll see in chapter 4, the notion of a private cloud softens this first principal but keeps all the others.

1.1.3 Elasticity as resource demands grow and shrink

The fact that this large pool of resources exists enables a concept known as *elasticity*—the third of our five main principles. Elasticity is such a key concept in cloud computing that Amazon decided to name its cloud Amazon Elastic Compute Cloud.

Elasticity—a synonym for *dynamic scaling*—refers to the ability to dynamically change how much resource is consumed in response to how much is needed. Typical applications require a base level of resources under normal, steady-state conditions, but need more resource under peak load conditions.

In a non-cloud world, you would have to build sufficient capacity to not only perform adequately under baseline load conditions, but also handle peak load scenarios with sufficiently good performance. In the case of a self-hosted model, this means over-provisioning the amount of hardware for a given allocation. In the case of a managed hosting deployment, you can start with a small set of resources and grow as the requirements of the application grow. But provisioning for a new set of dedicated hardware resources takes weeks or, in many larger organizations, months. Having thousands of virtualized resources that can be harnessed and released in correlation to application demand would be useless if such allocation and freeing required manual intervention.

1.1.4 Automation of new resource deployment

The ability to automatically (via an API) provision and deploy a new virtual instance of a machine, and, equivalently, to be able to free or de-provision an instance, is our fourth principle of cloud computing. A cloud-deployed application can provision new instances on an as-needed basis, and these resources are brought online within minutes. After the peak demand ebbs, and you don't need the additional resources, these

[1] The rapid shift to multicore servers only strengthens the impact of virtualization. Each virtual machine with its operating system and full complement of applications can run on its own core simultaneously with all other virtual machines on the same physical server.

virtual instances can be taken offline and de-provisioned, and you will no longer be billed. Your incremental cost is only for the hours that those additional instances were in use and active.

1.1.5 Metered billing that charges only for what you use

The fifth distinguishing characteristic of cloud computing is a metered billing model. In the case of managed hosting, as we mentioned before, there typically is an initial startup fee and an annual contract fee. The cloud model breaks that economic barrier because it's a pay-as-you-go model. There is no annual contract and no commitment for a specific level of consumption.

Typically, you can allocate resources as needed and pay for them on an hourly basis. This economic advantage benefits not only projects being run by IT organizations, but also innumerable entrepreneurs starting new businesses. Instead of needing to raise capital as they might have in the past, they can utilize vast quantities of compute resources for pennies per hour. For them, the cloud has drastically changed the playing field and allowed the little guy to be on equal footing with the largest corporations.

1.2 Benefits that can be garnered from moving to the cloud

"I'll never buy another server again," said the Director of IT for a medium-sized Software-as-a-Service (SaaS) company, only partially in jest, after recently completing the deployment of a new corporate website for his organization. This website (a PHP-based application with a MySQL backend) showcased the corporate brand and the primary online lead-generation capability for the company's business.

Before the overhaul, it was run from a redundant pair of web servers hosted by one of the leading managed-hosting service providers at a total cost of roughly $2,200/month. The company replaced the infrastructure for the original website with a cloud implementation consisting of a pair of virtual server instances running for roughly $250/month—almost a 90 percent savings! Its quality of service (QoS) team monitored the performance and availability of the website before and after the change and saw no measureable difference in the service quality delivered to end users. Buoyed by the success with this initial project, this organization is looking at all future initiatives for the possibility of deployment within the cloud, including a software-build system and offsite backup.

1.2.1 Economic benefits of the change from capital to operational expenses

As we said when discussing the five main principles of cloud computing, the fundamental economic benefit that cloud computing brings to the table is related to the magical conversion of CAPEX to OPEX. A pay-as-you-go model for resource use reshapes the fundamental cost structure of building and operating applications. The initial barrier to starting a project is drastically reduced; and until there is dramatic uptake in the use of an application that has been developed, the costs for running it remain low.

The good news is that this isn't the only cost advantage. By harnessing the cloud, you can also take advantage of cloud providers' economic leverage because of the volume at which they can purchase hardware, power, and bandwidth resources.

In many cases, the economic benefits discussed here will pan out—but as you'll see later, there are always exceptions. For some situations and applications, it makes better economic sense not to use cloud computing. It isn't a panacea.

1.2.2 Agility benefits from not having to procure and provision servers

In addition to lowering the financial barrier to initiating new projects, the cloud approach improves an organization's agility. It comprehensively reduces the months of planning, purchasing, provisioning, and configuring.

Let's take as an example a performance-testing project launching a new consumer-facing website. In the old world, there were two ways to solve this problem, depending on your timeframes and budget. The first involved purchasing a software license for a load-testing tool like HP Mercury LoadRunner and purchasing the requisite servers to run the load-testing software. At that point, you were ready to script your tests and run your test plan. Alternatively, you could hire an outside consulting company that specialized in performance testing and have it run the tests for you. Both were time-consuming exercises, depending on how long it took to negotiate either the licensing agreement for the software or the consulting agreement with the outside firm.

Fast-forward to the new world of cloud computing. You have two new faster and more flexible ways of accomplishing the same task: use an open-source load-testing application installed on cloud instances, and use the cloud's virtual machines to perform the load test (on as many servers as you need). The time required to set up and begin applying load to a system is under half an hour. This includes signing up for an account, as the Python open source load-testing tool called Pylot demonstrates (see http://coreygoldberg.blogspot.com/2009/02/pylot-web-load-testing-from-amazon.html).

If you're looking for a more packaged approach, you can use one of the SaaS offerings that uses the cloud to generate traffic. They can automatically run tests in a coordinated fashion across multiple instances running from multiple cloud operators, all in an on-demand fashion. In either of these scenarios, the time to result is a matter of hours or days, generating time, not to mention cost efficiencies. We'll explore more about cloud-based testing in chapter 7.

1.2.3 Efficiency benefits that may lead to competitive advantages

Adopting cloud technologies presents many opportunities to those who are able to capitalize on them. As we have discussed, there are potential economic as well as time-to-market advantages in using the technology. As organizations adopt cloud computing, they will realize efficiencies that organizations that are slower to move won't realize, putting them at an advantage competitively.

1.2.4 Security stronger and better in the cloud

Surprised by the heading? Don't be: it's true. As you're aware, corporate buildings no longer have electrical generators (which they used to) because we leave electricity generation to the experts. If corporations have their own data centers, they have to develop standard security operating procedures. But it's not their core business to run a secure data center. They can and will make mistakes. A lot of mistakes. The total annual fraud and security breach tab is $1 trillion, according to cybersecurity research firm Poneman (www.nationalcybersecurity.com).

But first, as always, you must weigh the potential benefits against the potential costs. You must take into account other factors, such as reliability and performance, before making the leap into the clouds. In future chapters, we'll address these issues; but suffice it to say we believe that after you understand them and take the proper measures, they can be managed. This done, you'll be able to realize the full benefits of moving to the cloud.

In the next section, we'll look at the evolution of technology that enabled cloud computing. This short detour into history is important because you can learn from previous platform shifts to understand what is similar and what is different this time. That in turn can help you make informed decisions about your shift to this new evolution of IT—the cloud.

1.3 Evolution of IT leading to cloud computing

Cloud computing didn't sprout fully formed from the technology ether in 2005. Its technological underpinnings developed over the course of the last 40 or so years. The technological process was evolutionary, across several disparate areas. But these advances, aggregated into a bundle, represent a revolutionary change in the way IT will be conducted in the future.

Gillett and Kapor made the first known reference to cloud computing in 1996 in an MIT paper (http://ccs.mit.edu/papers/CCSWP197/CCSWP197.html). Today's common understanding of cloud computing retains the original intent. It was a mere decade later when a real-world instantiation of the cloud came into existence as Amazon repurposed its latent e-commerce resources and went into the business of providing cloud services. From there, it was only a matter of a few months until the term became commonplace in our collective consciousness and, as figure 1.3 shows, in our Google search requests (they're the same thing in today's world, right?).

1.3.1 Origin of the "cloud" metaphor

One common question people ask is, "Where did the term *cloud* come from?" The answer is that for over a decade, whenever people drew pictures of application architectures that involved the internet, they inevitably represented the internet with a cloud, as shown in figure 1.4.

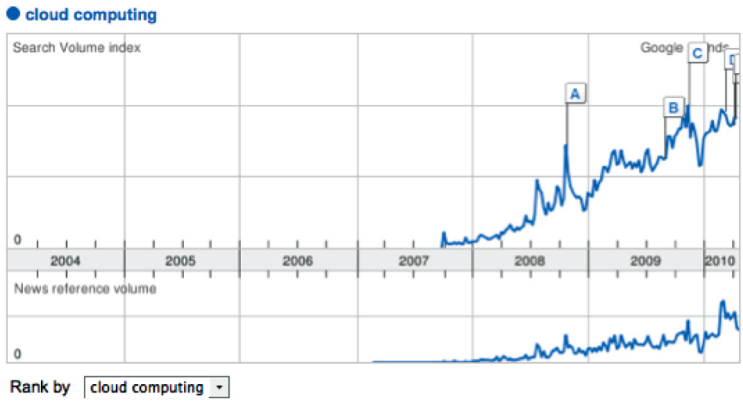


Figure 1.3 Cloud computing as a concept entered our collective consciousness in mid-2007. This figure shows the rapid rise in popularity of the search term *cloud computing* as measured by Google. The labels correspond to major cloud announcements. A: Microsoft announces it will rent cloud computing space; B: *Philadelphia Inquirer* reports, "Microsoft's cloud computing system grow is growing up"; C: *Winnipeg Free Press* reports, "Google looks to be cloud-computing rainmaker." Source: Google Trends (www.google.com/trends), on the term *cloud computing*.

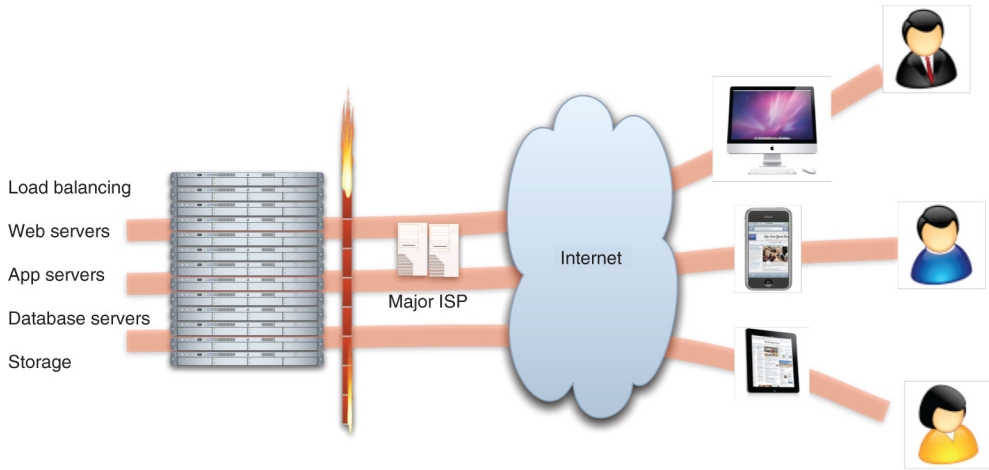


Figure 1.4 A picture of a cloud is a ubiquitous representation of the internet and is used almost universally in discussions or drawings of computer architecture.

The cloud in the diagram is meant to convey that anonymous people are sitting at browsers accessing the internet, and somehow their browser visits a site and begins to access its infrastructure and applications. From "somewhere out there" you get visitors who can become users who may buy products or services from you. Unlike internal customers to whom you may provide IT applications and services, this constituency exists "somewhere else," outside of your firewall, and hence outside of your domain of

control. The image of a cloud is merely a way to represent this vast potential base of anonymous users coming from the internet.

Those users must log in from a PC to access the internet. Technically, each one needs an Internet Service Provider (ISP) that may be a telecom company, their employer, or a dedicated internet access company (such as AOL). Each ISP needs a bank of machines that people can access and that in turn has access to the internet.

Simply put, the earliest concept of the cloud consisted of large aggregations of computers with access to the internet, accessed by people through their browsers. The concept has remained surprisingly true to that early vision but has evolved and matured in important ways. We'll explore those ways in detail in this book.

1.3.2 Major computing paradigm shifts: mainframes to client-server to web

In the 1960s, we saw the development of the first commercial mainframes. In the beginning, these were single-user systems, but they evolved in the 1970s to systems that were time-shared. In this model, the large computing resource was *virtualized*, and a virtual machine was allocated to individual users who were sharing the system (but to each, it seemed that they had an entire dedicated machine).

Virtual instances were accessed in a thin-client model by green-screen terminals. This mode of access can be seen as a direct analog of the concept of virtualized instances in the cloud, although then a single machine was divided among users. In the cloud, it's potentially many thousands of machines. The scarcity of the computing resource in the past drove the virtualization of that resource so that it could be shared, whereas now, the desire to fully utilize physical compute resources is driving cloud virtualization.

As we evolved and entered the client-server era, the primacy of the mainframe as the computing center of the universe dissolved. As computing power increased, work gradually shifted away from centralized computing resources toward increasingly powerful distributed systems. In the era of the PC-based desktop applications, this shift was nearly complete: computing resources for many everyday computing tasks moved to the desktop and became thick client applications (such as Microsoft Office). The mainframe retained its primacy only for corporate or department-wide applications, relegating it to this role alone.

The standardization of networking technology simplified the ability to connect systems as TCP/IP became the protocol of the burgeoning internet in the 1980s. The ascendancy of the web and HTTP in the late 1990s swung the pendulum back to a world where the thin-client model reigned supreme. The world was now positioned to move into the era of *cloud computing*. The biggest stages of the evolution of IT are diagrammed vertically in a timeline in figure 1.5.

The computing evolution we are still in the midst of has had many stages. Platform shifts like mainframe to client-server and then client-server to web were one dimension of the evolution. One that may be less apparent but that is having as profound an

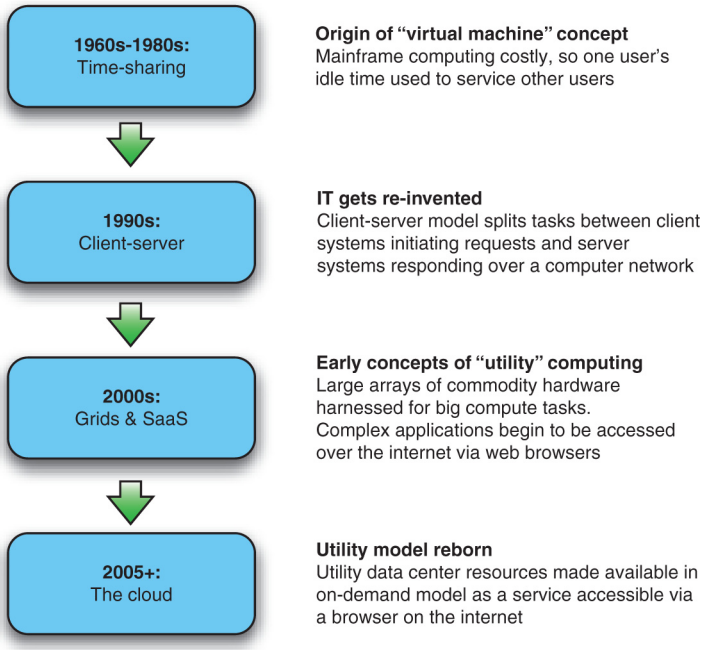


Figure 1.5 Cloud computing is best understood as an evolutionary change. The key elements and concepts of cloud computing emerged gradually over several decades through the various predominant computing paradigms.

impact is the evolution of the data center and how physical computing resources are housed, powered, maintained, and upgraded.

1.3.3 Housing of physical computing resources: data center evolution

Over the past four decades, there have been tremendous changes in hardware capabilities, specifically in computing power and storage. The ability to quickly process prodigious amounts of data on inexpensive and mass-produced commodity servers means that a few inexpensive racks of servers can handle problems that were tackled on NSA-sized budgets as recently as the early 1990s.

One measure of the progress in computational power is the cost in Floating Point Operations Per Second, or FLOPS. FLOPS are simple mathematical operations (such as addition, multiplication, and division) that can be performed in a single operation by a computer. Comparing the number of operations that two computers can perform in one second allows for a rough measure of their computational strength. In 1976, the state-of-the-art Cray-1 was capable of delivering roughly 150 million FLOPS (megaFLOPS) at the price point of $5 million, or over $33,000/MegaFLOPS. A typical quad-core-processor-based PC today can be purchased for under $1,000 and can perform 50 GigaFLOPS (billion FLOPS), which comes out to about $0.02/MegaFLOPS.

Similarly, the cost of storage has decreased dramatically over the last few decades as the capacity to store data has kept pace with the ability to produce terabytes of digital content in the form of high-definition HD video and high-resolution imagery. In the early 1980s, disk space costs exceeded $200/MB; today, this cost has come down to under $0.01/MB.

Network technologies have advanced as well, with modern bandwidth rates in the 100–1000 Gbps range commonplace in data centers today. As for WAN, the turn of the millennium saw a massive build-out of dark fiber, bringing high-speed broadband to most urban areas. More rural areas have satellite coverage, and on-the-go, high-speed wireless networks mean almost ubiquitous broadband connectivity to the grid.

To support the cloud, a huge data-center build-out is now underway. Google, Microsoft, Yahoo!, Expedia, Amazon, and others are deploying massive data centers. These are the engine rooms that power the cloud, and they now account for more than 1.2 percent of the U.S.'s total electricity usage (including cooling and auxiliaries),[1] which doubled over the period from 2000 to 2005. We'll present the economies of scale and much more detail about how these mega data centers are shaping up in chapter 2.

1.3.4 Software componentization and remote access: SOA, virtualization, and SaaS

On the software side of the cloud evolution are three important threads of development: virtualization, SOA, and SaaS. Two of these are technological, and the third relates to the business model.

The first important thread is virtualization. As discussed previously, virtualization isn't a new concept, and it existed in mainframe environments. The new innovation that took place in the late 1990s was the extension of this idea to commodity hardware. Virtualization as pioneered by VMware and others took advantage of the capacity of modern multicore CPUs and made it possible to partition and time-slice the operation of commodity servers. Large server farms based on these commodity servers were partitioned for use across large populations of users.

SOA is the second software concept necessary for cloud computing. We see SOA as the logical extension of browser-based standardization applied to machine-to-machine communication. Things that humans did through browsers that interacted with a web server are now done machine-to-machine using the same web-based standard protocols and are called *SOA*. SOA makes practical the componentization and composition of services into applications, and hence it can serve as the architectural model for building composite applications running on multiple virtualized instances.

The final software evolution we consider most pertinent to the cloud is SaaS. Instead of being a technological innovation, this is a business model innovation. Historically, enterprise software was sold predominantly in a perpetual license model. In this model, a customer purchased the right to use a certain software application in

[1] Jonathan G. Koomey, Ph.D. (www.koomey.com), Lawrence Berkeley National Laboratory & Stanford University.

perpetuity for a fixed, and in many cases high, price. In subsequent years, they paid for support and maintenance at typically around 18 percent of the original price. This entitled the customer to upgrades of the software and help when they ran into difficulty. In the SaaS model, you don't purchase the software—you rent it. Typically, the fee scales with the amount of use, so the value derived from the software is proportional to the amount spent on it. The customer buys access to the software for a specified term, which may be days, weeks, months, or years, and can elect to stop paying when they no longer need the SaaS offering. Cloud computing service providers have adopted this *pay-as-you-go* or *on-demand* model.

This brings up an important point we need to consider next. SaaS is one flavor or layer in a stack of cloud types. A common mistake people make in these early days of the cloud is to make an apples-to-oranges comparison of one type of cloud to another. To avoid that, the next section will classify the different layers in the cloud stack and how they compare and contrast.

1.4 Classifying cloud layers: different types for different uses

First, let's learn a little more about how SaaS evolved and established itself, to set the context for discussing the other classes of clouds.

In the earliest days of commercially practicable computing, computer resources were scarce, and the primary model for their use was much like a utility. But this was different from the sense of utility that cloud computing offers today; it was more akin to the community well in a village during a drought. Members of the community had access to and were allocated a fixed amount of water. In the case of cloud computing today, we've returned to the notion of computing being available as a utility, but without the scarcity.

The cloud movement was presaged by the shift in business model toward SaaS that took over the software industry at the turn of the century. Before it was called SaaS, it was an application rented from an Application Service Provider (ASP); here, the traditional enterprise license model was turned on its head, and you purchased in a pay-as-you-go manner, with costs scaling with usage instead of having a large up-front capital investment. You didn't need to provision hardware and software; instead, the services were turned on when needed. After this approach was renamed SaaS, it evolved into several new kinds of offerings that we'll explore next.

We can classify cloud computing several ways. In this book, we present a taxonomy where cloud services are described generically as "X as a Service," where X can take on values such as Hardware, Infrastructure, Platform, Framework, Application, and even Datacenter. Vendors aren't in agreement about what these designations mean, nor are they consistent in describing themselves as belonging to these categories. Despite this, we'll reproduce one interesting hierarchy that illustrates the use of these terms, with representative vendors (some at this point only historical) populating the diagram in figure 1.6.

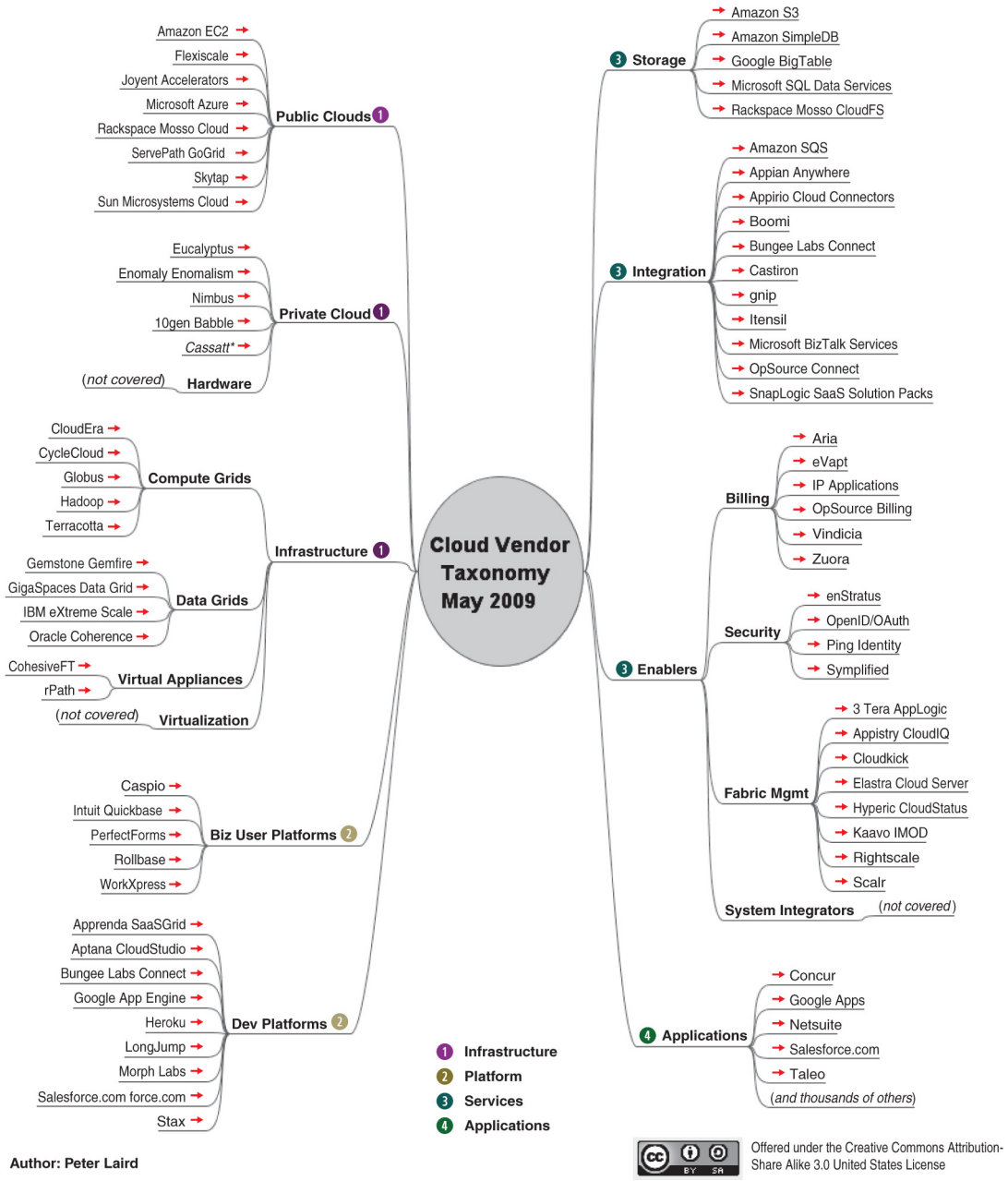


Figure 1.6 Cloud technologies are evolving as various vendors attempt to provide services populating the cloud ecosystem. These services run the gamut from the hardware systems used to build cloud infrastructure to integration services and cloud-based applications. Source: Peter Laird, http://peterlaird.blogspot.com.

A more simplified representation of the cloud types shown in figure 1.7 highlights important aspects and key characteristics of different kinds of cloud offerings.

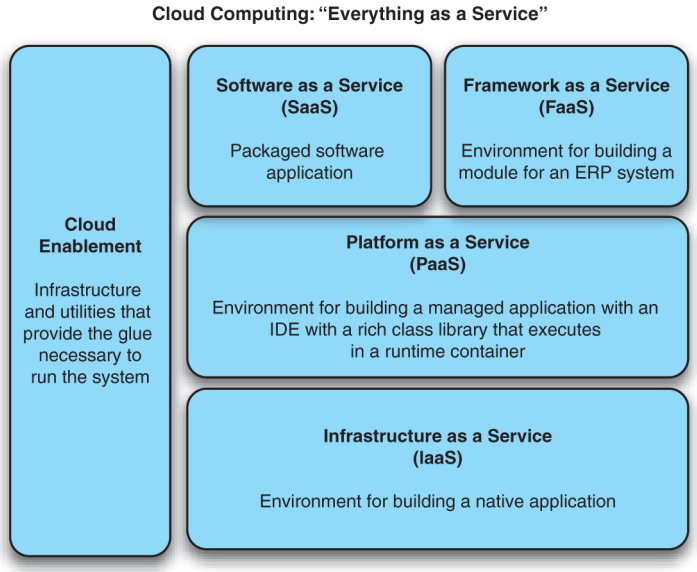


Figure 1.7 In the X-as-a-Service taxonomy, cloud services are classified by the level of prepackaging offered to the consumer of the specific service. An IaaS provides computing capabilities in the rawest form and hence offers the greatest flexibility. At the highest layers, there is less flexibility but also less complexity to be managed.

What does XaaS mean generically? It means on demand, requiring little or no capital expenditure. It means consumable remotely and across any mode of access over the internet, and in a metered billing model. Let's now go through the boxes representing the different classes of clouds in figure 1.7. First up is IaaS.

1.4.1 Infrastructure as a Service (IaaS)

The lowest level of XaaS is known as IaaS, or sometimes as Hardware as a Service (HaaS). A good example of IaaS is the Amazon Elastic Compute Cloud (EC2).

A user of IaaS is operating at the lowest level of granularity available and with the least amount of prepackaged functionality. An IaaS provider supplies virtual machine images of different operating system flavors. These images can be tailored by the developer to run any custom or packaged application. These applications can run natively on the chosen OS and can be saved for a particular purpose. The user can bring online and use instances of these virtual machine images when needed. Use of these images is typically metered and charged in hour-long increments.

Storage and bandwidth are also consumable commodities in an IaaS environment, with storage typically charged per gigabyte per month and bandwidth charged for transit into and out of the system.

IaaS provides great flexibility and control over the cloud resources being consumed, but typically more work is required of the developer to operate effectively in the environment. In chapter 2, we'll delve into IaaS and see how it works in greater detail.

1.4.2 Platform as a Service (PaaS)

PaaS's fundamental billing quantities are somewhat similar to those of IaaS: consumption of CPU, bandwidth, and storage operates under similar models. Examples of PaaS include Google AppEngine and Microsoft Azure. The main difference is that PaaS requires less interaction with the bare metal of the system. You don't need to directly interact with or administer the virtual OSs. Instead, you can let the platform abstract away that interaction and concentrate specifically on writing the application. This simplification generally comes at the cost of less flexibility and the requirement to code in the specific languages supported by the particular PaaS provider.

1.4.3 Software as a Service (SaaS) and Framework as a Service (FaaS)

SaaS, as described earlier in the chapter, refers to services and applications that are available on an on-demand basis. Salesforce.com is an example. FaaS is an environment adjunct to a SaaS offering and allows developers to extend the prebuilt functionality of the SaaS applications. Force.com is an example of a FaaS that extends the Salesforce.com SaaS offering.

FaaS offerings are useful specifically for augmenting and enhancing the capabilities of the base SaaS system. You can use FaaS for creating either custom, specialized applications for a specific organization, or general-purpose applications that can be made available to any customer of the SaaS offering. Like a PaaS environment, a developer in a FaaS environment can only use the specific languages and APIs provided by the FaaS.

1.4.4 Private clouds as precursors of public clouds

In addition to the classifications we discussed earlier, we should introduce some important concepts relative to the different classifications of clouds. *Private clouds* are a variant of generic cloud computing where internal data-center resources of an enterprise or organization aren't made available to the general public—that is, these pooled computing resources are actually not available to *any* subscribing users but are instead controlled by an organization for the benefit of other members of that organization. The public clouds of providers such as Amazon and Google were originally used as private clouds by those companies for other lines of business (book retailing and internet search, respectively).

If an organization has sufficient users and enough overall capacity, a private cloud implementation can behave much like a public cloud, albeit on a reduced scale. There has been a tremendous amount of capital investment in data-center resources over the past decade, and one of the important movements is the reorienting of these assets toward cloud-usage models.

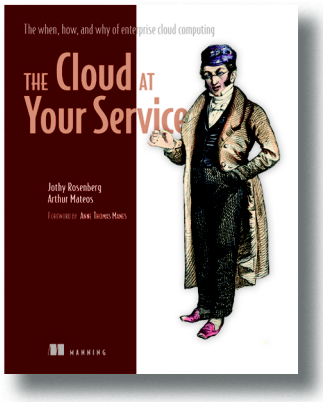
Hybrid clouds combine private and public clouds. You can use them in cases where the capacity of a private cloud is exhausted and excess capacity needs to be provisioned elsewhere.

1.5 Summary

The cloud offers the illusion of infinite resources, available on demand. You no longer need to play the guessing game of how many users need to be supported and how scalable the application is. The cloud takes care of the peaks and troughs of utilization times. In the world of the cloud, you pay for only the resources you use, when you use them. This is the revolutionary change: the ability to handle scale without paying a premium. In this realm of true utility computing, resource utilization mirrors the way we consume electricity or water.

In this chapter, we defined the cloud as computing services that are offered by a third party, are available for use when needed, and can be scaled dynamically in response to changing need. We then touched briefly on the evolution of computing and the developments that led to where we are today. Finally, we looked at a simple cloud classification that should help you understand the various flavors of cloud offerings that are available in the market today and should prevent you from making apples-and-oranges comparisons between incompatible classes of clouds.

As we delve deeper in the next chapter and look at how the cloud works, you'll gain a better understanding of these types of clouds and when it makes sense to use each kind.



Practically unlimited storage, instant scalability, zero-downtime upgrades, low start-up costs, plus pay-only-for-what-you-use without sacrificing security or performance are all benefits of cloud computing. How do you make it work in your enterprise? What should you move to the cloud? How? And when?

The Cloud at Your Service answers these questions and more. Written for IT pros at all levels, this book finds the sweet spot between rapidly changing details and hand-waving hype. It shows you practical ways to work with current services like Amazon's EC2 and S3. You'll also learn the pros and cons of private clouds, the truth about cloud data security, and how to use the cloud for high scale applications.

What's inside

- How to build scalable and reliable applications
- The state of the art in technology, vendors, practices
- What to keep in-house and what to offload
- How to migrate existing IT to the cloud
- How to build secure applications and data centers

Amazon Web Services in Action

What's the biggest advantage of using Amazon Web Services (AWS)? For us it's being able to automate every part of your cloud infrastructure. AWS offers an API, and lots of tools to launch, configure, modify, and delete computing, storage, and networking infrastructure. Our book, *Amazon Web Services in Action*, provides a deep introduction into the most important services and architecture principles. Chapter 1 answers the question: What is Amazon Web Services? You'll learn about the concepts behind AWS and gain a brief overview of what you can do with AWS.

Chapter 1 from *Amazon Web Services in Action* by Michael Wittig and Andreas Wittig.

What is Amazon Web Services?

This chapter covers

- Overview of Amazon Web Services
- Benefits of using Amazon Web Services
- Examples of what you can do with Amazon Web Services
- Creating and setting up an Amazon Web Services account

Amazon Web Services (AWS) is a platform of web services offering solutions for computing, storing, and networking, at different layers of abstraction. You can use these services to host web sites, run enterprise applications, and mine tremendous amounts of data. The term *web service* means services can be controlled via a web interface. The web interface can be used by machines or by humans via a graphical user interface. The most prominent services are EC2, which offers virtual servers, and S3, which offers storage capacity. Services on AWS work well together; you can

use them to replicate your existing on-premises setup or design a new setup from scratch. Services are charged for on a pay-per-use pricing model.

As an AWS customer, you can choose among different data centers. AWS data centers are distributed in the United States, Europe, Asia, and South America. For example, you can start a virtual server in Japan in the same way you can start a virtual server in Ireland. This enables you to serve customers worldwide with a global infrastructure.

The map in figure 1.1 shows the data centers available to all customers.

Which hardware powers AWS?

AWS keeps secret the hardware used in its data centers. The scale at which AWS operates computing, networking, and storage hardware is tremendous. It probably uses commodity components to save money compared to hardware that charges extra for a brand name. Handling of hardware failure is built into real-world processes and software.[1]

AWS also uses hardware especially developed for its use cases. A good example is the Xeon E5-2666 v3 CPU from Intel. This CPU is optimized to power virtual servers from the c4 family.

In more general terms, AWS is known as a *cloud computing platform*.

1.1 What is cloud computing?

Almost every IT solution is labeled with the term *cloud computing* or just *cloud* nowadays. A buzzword may help to sell, but it's hard to work with in a book.

Cloud computing, or the cloud, is a metaphor for supply and consumption of IT resources. The IT resources in the cloud aren't directly visible to the user; there are layers of abstraction in between. The level of abstraction offered by the cloud may vary

Figure 1.1 AWS data center locations

[1] Bernard Golden, "Amazon Web Services (AWS) Hardware," *For Dummies*, http://mng.bz/k6lT.

from virtual hardware to complex distributed systems. Resources are available on demand in enormous quantities and paid for per use.

Here's a more official definition from the National Institute of Standards and Technology:

> *Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.*
>
> —The NIST Definition of Cloud Computing, National Institute of Standards and Technology

Clouds are often divided into the following types:

- *Public*—A cloud managed by an organization and open to use by the general public
- *Private*—A cloud that virtualizes and shares the IT infrastructure within a single organization
- *Hybrid*—A mixture of a public and a private cloud

AWS is a public cloud. Cloud computing services also have several classifications:

- *Infrastructure as a service (IaaS)*—Offers fundamental resources like computing, storage, and networking capabilities, using virtual servers such as Amazon EC2, Google Compute Engine, and Microsoft Azure virtual machines
- *Platform as a service (PaaS)*—Provides platforms to deploy custom applications to the cloud, such as AWS Elastic Beanstalk, Google App Engine, and Heroku
- *Software as a service (SaaS)*—Combines infrastructure and software running in the cloud, including office applications like Amazon WorkSpaces, Google Apps for Work, and Microsoft Office 365

The AWS product portfolio contains IaaS, PaaS, and SaaS. Let's take a more concrete look at what you can do with AWS.

1.2 What can you do with AWS?

You can run any application on AWS by using one or a combination of services. The examples in this section will give you an idea of what you can do with AWS.

1.2.1 Hosting a web shop

John is CIO of a medium-sized e-commerce business. His goal is to provide his customers with a fast and reliable web shop. He decided to host the web shop on-premises, and three years ago he rented servers in a data center. A web server handles requests from customers, and a database stores product information and orders. John is evaluating how his company can take advantage of AWS by running the same setup on AWS, as shown in figure 1.2.

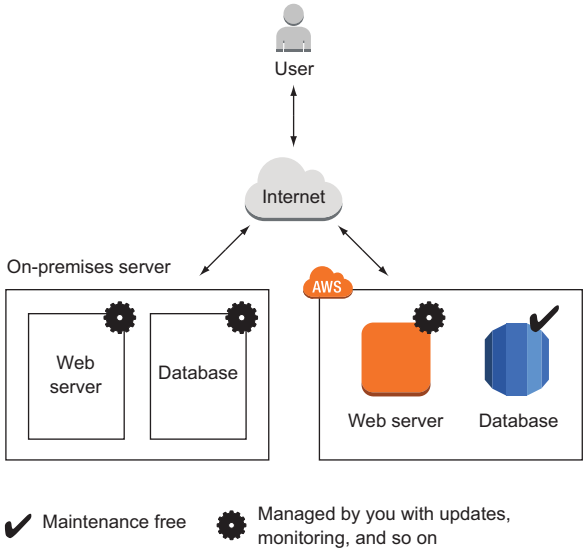


Figure 1.2 Running a web shop on-premises vs. on AWS

John realized that other options are available to improve his setup on AWS with additional services:

- The web shop consists of dynamic content (such as products and their prices) and static content (such as the company logo). By splitting dynamic and static content, John reduced the load for his web servers and improved performance by delivering the static content over a content delivery network (CDN).
- John uses maintenance-free services including a database, an object store, and a DNS system on AWS. This frees him from managing these parts of the system, decreases operational costs, and improves quality.
- The application running the web shop can be installed on virtual servers. John split the capacity of the old on-premises server into multiple smaller virtual servers at no extra cost. If one of these virtual servers fails, the load balancer will send customer requests to the other virtual servers. This setup improves the web shop's reliability.

Figure 1.3 shows how John enhanced the web shop setup with AWS.

John started a proof-of-concept project and found that his web application can be transferred to AWS and that services are available to help improve his setup.

1.2.2 Running a Java EE application in your private network

Maureen is a senior system architect in a global corporation. She wants to move parts of the business applications to AWS when the company's data-center contract expires in a few months, to reduce costs and gain flexibility. She found that it's possible to run enterprise applications on AWS.

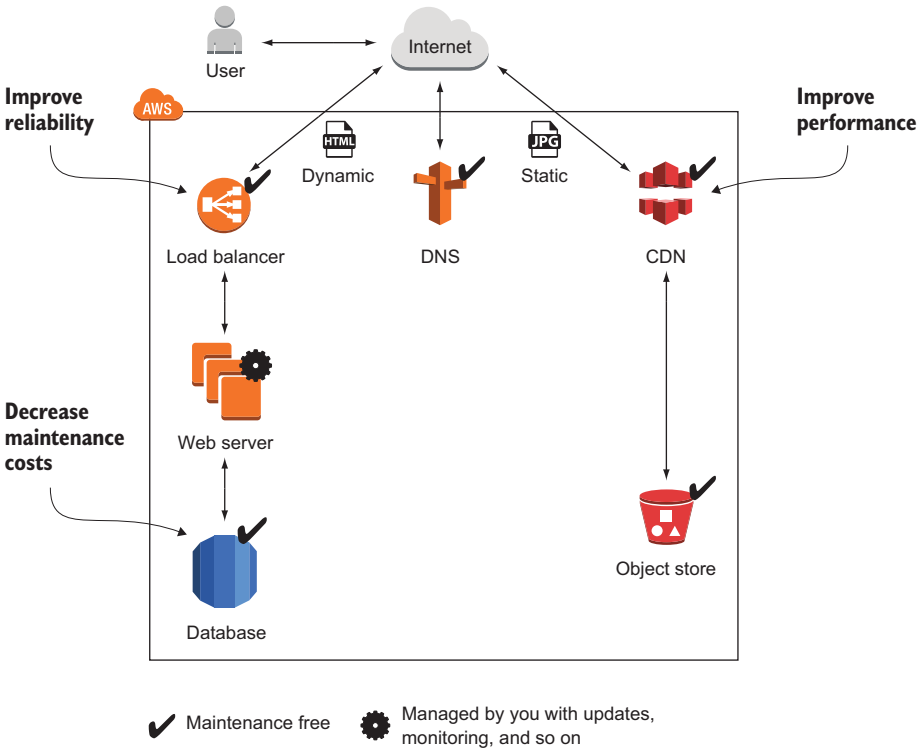


Figure 1.3 Running a web shop on AWS with CDN for better performance, a load balancer for high availability, and a managed database to decrease maintenance costs

To do so, she defines a virtual network in the cloud and connects it to the corporate network through a virtual private network (VPN) connection. The company can control access and protect mission-critical data by using subnets and control traffic between them with access-control lists. Maureen controls traffic to the internet using Network Address Translation (NAT) and firewalls. She installs application servers on virtual machines (VMs) to run the Java EE application. Maureen is also thinking about storing data in a SQL database service (such as Oracle Database Enterprise Edition or Microsoft SQL Server EE). Figure 1.4 illustrates Maureen's architecture.

Maureen has managed to connect the on-premises data center with a private network on AWS. Her team has already started to move the first enterprise application to the cloud.

1.2.3 Meeting legal and business data archival requirements

Greg is responsible for the IT infrastructure of a small law office. His primary goal is to store and archive all data in a reliable and durable way. He operates a file server to

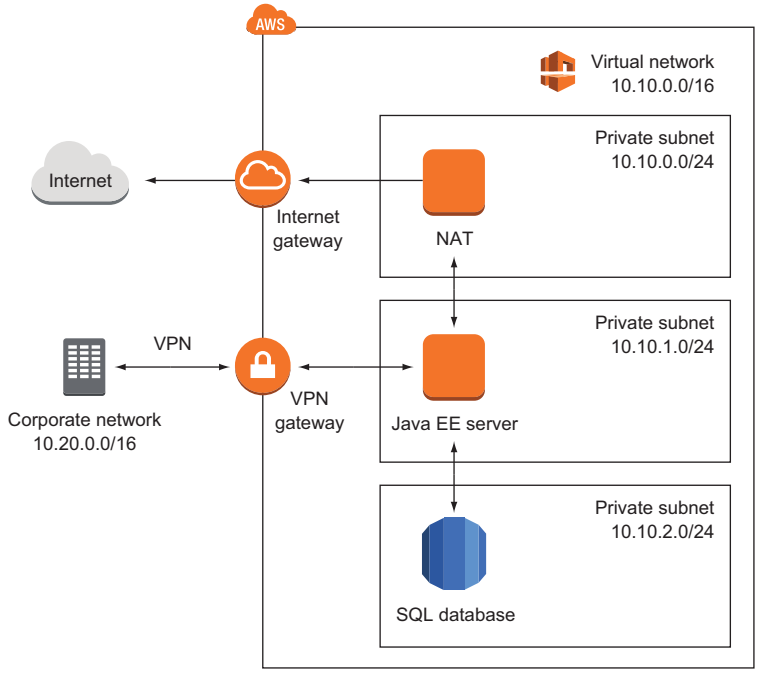


Figure 1.4 Running a Java EE application with enterprise networking on AWS

offer the possibility of sharing documents within the office. Storing all the data is a challenge for him:

- He needs to back up all files to prevent the loss of critical data. To do so, Greg copies the data from the file server to another network-attached storage, so he had to buy the hardware for the file server twice. The file server and the backup server are located close together, so he is failing to meet disaster-recovery requirements to recover from a fire or a break-in.
- To meet legal and business data archival requirements, Greg needs to store data for a long time. Storing data for 10 years or longer is tricky. Greg uses an expensive archive solution to do so.

To save money and increase data security, Greg decided to use AWS. He transferred data to a highly available object store. A storage gateway makes it unnecessary to buy and operate network-attached storage and a backup on-premises. A virtual tape deck takes over the task of archiving data for the required length of time. Figure 1.5 shows how Greg implemented this use case on AWS and compares it to the on-premises solution.

Greg is fine with the new solution to store and archive data on AWS because he was able to improve quality and he gained the possibility of scaling storage size.

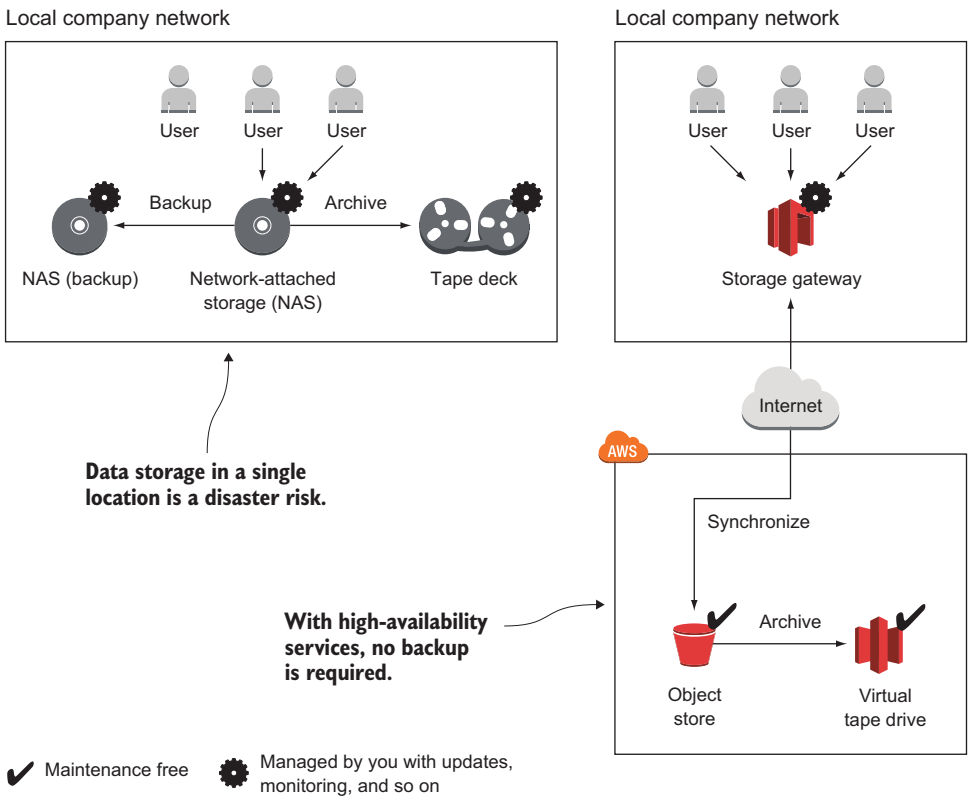


Figure 1.5 Backing up and archiving data on-premises and on AWS

1.2.4 Implementing a fault-tolerant system architecture

Alexa is a software engineer working for a fast-growing startup. She knows that Murphy's Law applies to IT infrastructure: anything that can go wrong, will go wrong. Alexa is working hard to build a fault-tolerant system to prevent outages from ruining the business. She knows that there are two type of services on AWS: fault-tolerant services and services that can be used in a fault-tolerant way. Alexa builds a system like the one shown in figure 1.6 with a fault-tolerant architecture. The database service is offered with replication and failover handling. Alexa uses virtual servers acting as web servers. These virtual servers aren't fault tolerant by default. But Alexa uses a load balancer and can launch multiple servers in different data centers to achieve fault tolerance.

So far, Alexa has protected the startup from major outages. Nevertheless, she and her team are always planning for failure.

You now have a broad idea of what you can do with AWS. Generally speaking, you can host any application on AWS. The next section explains the nine most important benefits AWS has to offer.

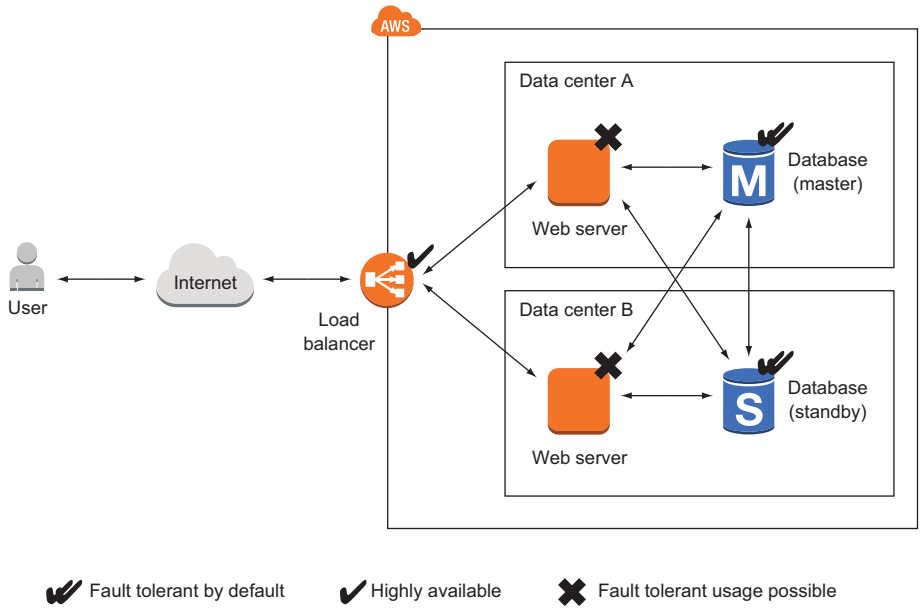


Figure 1.6 Building a fault-tolerant system on AWS

1.3 How you can benefit from using AWS

What's the most important advantage of using AWS? Cost savings, you might say. But saving money isn't the only advantage. Let's look at other ways you can benefit from using AWS.

1.3.1 Innovative and fast-growing platform

In 2014, AWS announced more than 500 new services and features during its yearly conference, re:Invent at Las Vegas. On top of that, new features and improvements are released every week. You can transform these new services and features into innovative solutions for your customers and thus achieve a competitive advantage.

The number of attendees to the re:Invent conference grew from 9,000 in 2013 to 13,500 in 2014.[1] AWS counts more than 1 million businesses and government agencies among its customers, and in its Q1 2014 results discussion, the company said it will continue to hire more talent to grow even further.[2] You can expect even more new features and services in the coming years.

---

[1] Greg Bensinger, "Amazon Conference Showcases Another Side of the Retailer's Business," *Digits*, Nov. 12, 2014, http://mng.bz/hTBo.

[2] "Amazon.com's Management Discusses Q1 2014 Results - Earnings Call Transcript," *Seeking Alpha*, April 24, 2014, http://mng.bz/60qX.

1.3.2 Services solve common problems

As you've learned, AWS is a platform of services. Common problems such as load balancing, queuing, sending email, and storing files are solved for you by services. You don't need to reinvent the wheel. It's your job to pick the right services to build complex systems. Then you can let AWS manage those services while you focus on your customers.

1.3.3 Enabling automation

Because AWS has an API, you can automate everything: you can write code to create networks, start virtual server clusters, or deploy a relational database. Automation increases reliability and improves efficiency.

The more dependencies your system has, the more complex it gets. A human can quickly lose perspective, whereas a computer can cope with graphs of any size. You should concentrate on tasks a human is good at—describing a system—while the computer figures out how to resolve all those dependencies to create the system. Setting up an environment in the cloud based on your blueprints can be automated with the help of infrastructure as code, covered in chapter 4.

1.3.4 Flexible capacity (scalability)

Flexible capacity frees you from planning. You can scale from one server to thousands of servers. Your storage can grow from gigabytes to petabytes. You no longer need to predict your future capacity needs for the coming months and years.

If you run a web shop, you have seasonal traffic patterns, as shown in figure 1.7. Think about day versus night, and weekday versus weekend or holiday. Wouldn't it be nice if you could add capacity when traffic grows and remove capacity when traffic shrinks? That's exactly what flexible capacity is about. You can start new servers within minutes and throw them away a few hours after that.

The cloud has almost no capacity constraints. You no longer need to think about rack space, switches, and power supplies—you can add as many servers as you like. If your data volume grows, you can always add new storage capacity.

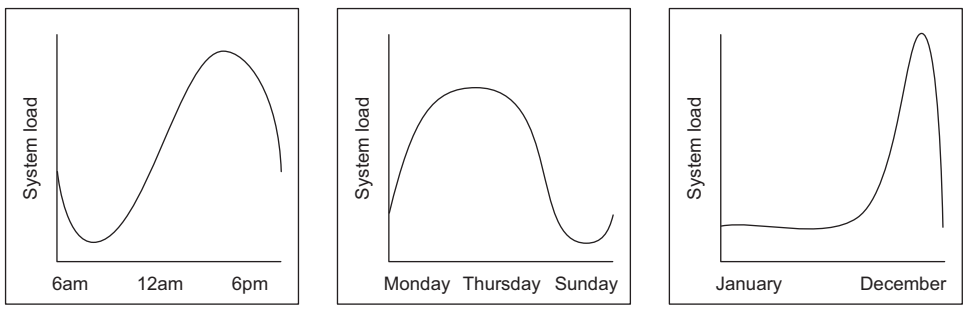


Figure 1.7 Seasonal traffic patterns for a web shop

Flexible capacity also means you can shut down unused systems. In one of our last projects, the test environment only ran from 7:00 a.m. to 8:00 p.m. on weekdays, allowing us to save 60%.

1.3.5 Built for failure (reliability)

Most AWS services are fault-tolerant or highly available. If you use those services, you get reliability for free. AWS supports you as you build systems in a reliable way. It provides everything you need to create your own fault-tolerant systems.

1.3.6 Reducing time to market

In AWS, you request a new virtual server, and a few minutes later that virtual server is booted and ready to use. The same is true with any other AWS service available. You can use them all on demand. This allows you to adapt your infrastructure to new requirements very quickly.

Your development process will be faster because of the shorter feedback loops. You can eliminate constraints such as the number of test environments available; if you need one more test environment, you can create it for a few hours.

1.3.7 Benefiting from economies of scale

At the time of writing, the charges for using AWS have been reduced 42 times since 2008:

- In December 2014, charges for outbound data transfer were lowered by up to 43%.
- In November 2014, charges for using the search service were lowered by 50%.
- In March 2014, charges for using a virtual server were lowered by up to 40%.

As of December 2014, AWS operated 1.4 million servers. All processes related to operations must be optimized to operate at that scale. The bigger AWS gets, the lower the prices will be.

1.3.8 Worldwide

You can deploy your applications as close to your customers as possible. AWS has data centers in the following locations:

- United States (northern Virginia, northern California, Oregon)
- Europe (Germany, Ireland)
- Asia (Japan, Singapore)
- Australia
- South America (Brazil)

With AWS, you can run your business all over the world.

1.3.9 Professional partner

AWS is compliant with the following:

- *ISO 27001*—A worldwide information security standard certified by an independent and accredited certification body

- *FedRAMP & DoD CSM*—Ensures secure cloud computing for the U.S. Federal Government and the U.S. Department of Defense
- *PCI DSS Level 1*—A data security standard (DSS) for the payment card industry (*PCI*) to protect cardholders data
- *ISO 9001*—A standardized quality management approach used worldwide and certified by an independent and accredited certification body

If you're still not convinced that AWS is a professional partner, you should know that Airbnb, Amazon, Intuit, NASA, Nasdaq, Netflix, SoundCloud, and many more are running serious workloads on AWS.

The cost benefit is elaborated in more detail in the next section.

1.4 How much does it cost?

A bill from AWS is similar to an electric bill. Services are billed based on usage. You pay for the hours a virtual server was running, the used storage from the object store (in gigabytes), or the number of running load balancers. Services are invoiced on a monthly basis. The pricing for each service is publicly available; if you want to calculate the monthly cost of a planned setup, you can use the AWS Simple Monthly Calculator (http://aws.amazon.com/calculator).

1.4.1 Free Tier

You can use some AWS services for free during the first 12 months after you sign up. The idea behind the Free Tier is to enable you to experiment with AWS and get some experience. Here is what's included in the Free Tier:

- 750 hours (roughly a month) of a small virtual server running Linux or Windows. This means you can run one virtual server the whole month or you can run 750 virtual servers for one hour.
- 750 hours (or roughly a month) of a load balancer.
- Object store with 5 GB of storage.
- Small database with 20 GB of storage, including backup.

If you exceed the limits of the Free Tier, you start paying for the resources you consume without further notice. You'll receive a bill at the end of the month. We'll show you how to monitor your costs before you begin using AWS. If your Free Tier ends after one year, you pay for all resources you use.

You get some additional benefits, as detailed at http://aws.amazon.com/free. This book will use the Free Tier as much as possible and will clearly state when additional resources are required that aren't covered by the Free Tier.

1.4.2 Billing example

As mentioned earlier, you can be billed in several ways:

- *Based on hours of usage*—If you use a server for 61 minutes, that's usually counted as 2 hours.

- *Based on traffic*—Traffic can be measured in gigabytes or in number of requests.
- *Based on storage usage*—Usage can be either provisioned capacity (for example, 50 GB volume no matter how much you use) or real usage (such as 2.3 GB used).

Remember the web shop example from section 1.2? Figure 1.8 shows the web shop and adds information about how each part is billed.

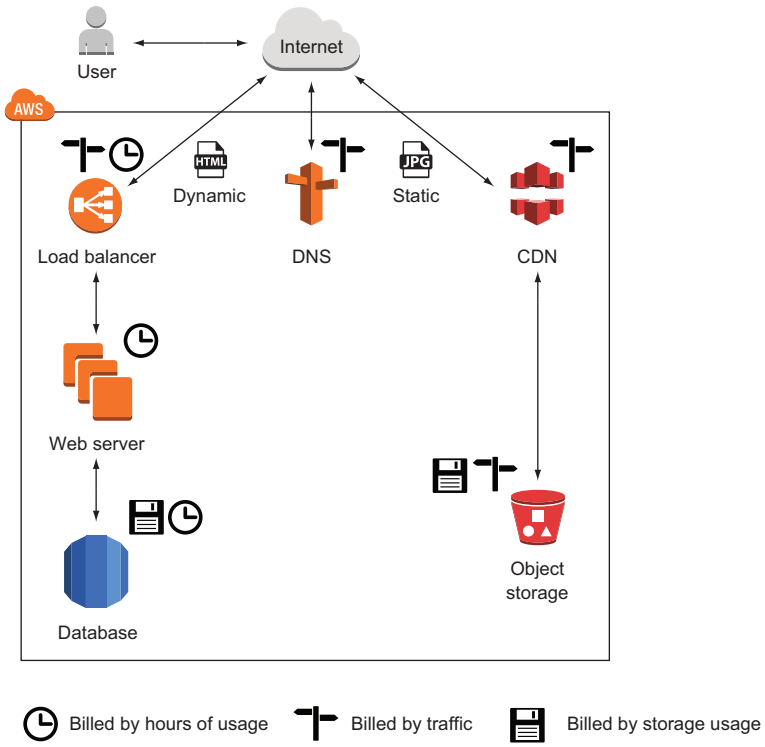


Figure 1.8 Web shop billing example

Let's assume your web shop started successfully in January, and you decided to run a marketing campaign to increase sales for the next month. Lucky you: you were able to increase the number of visitors of your web shop fivefold in February. As you already know, you have to pay for AWS based on usage. Table 1.1 shows your bills for January and February. The number of visitors increased from 100,000 to 500,000, and your monthly bill increased from 142.37 USD to 538.09 USD, which is a 3.7-fold increase. Because your web shop had to handle more traffic, you had to pay more for services, such as the CDN, the web servers, and the database. Other services, like the storage of static files, didn't experience more usage, so the price stayed the same.

With AWS, you can achieve a linear relationship between traffic and costs. And other opportunities await you with this pricing model.

Table 1.1 How an AWS bill changes if the number of web shop visitors increases

Service	January usage	February usage	February charge	Increase
Visits to website	100,000	500,000		
CDN	26 M requests + 25 GB traffic	131 M requests + 125 GB traffic	113.31 USD	90.64 USD
Static files	50 GB used storage	50 GB used storage	1.50 USD	0.00 USD
Load balancer	748 hours + 50 GB traffic	748 hours + 250 GB traffic	20.30 USD	1.60 USD
Web servers	1 server = 748 hours	4 servers = 2,992 hours	204.96 USD	153.72 USD
Database (748 hours)	Small server + 20 GB storage	Large server + 20 GB storage	170.66 USD	128.10 USD
Traffic (outgoing traffic to internet)	51 GB	255 GB	22.86 USD	18.46 USD
DNS	2 M requests	10 M requests	4.50 USD	3.20 USD
Total cost			538.09 USD	395.72 USD

1.4.3 Pay-per-use opportunities

The AWS pay-per-use pricing model creates new opportunities. You no longer need to make upfront investments in infrastructure. You can start servers on demand and only pay per hour of usage; and you can stop using those servers whenever you like and no longer have to pay for them. You don't need to make an upfront commitment regarding how much storage you'll use.

A big server costs exactly as much as two smaller ones with the same capacity. Thus you can divide your systems into smaller parts, because the cost is the same. This makes fault tolerance affordable not only for big companies but also for smaller budgets.

1.5 Comparing alternatives

AWS isn't the only cloud computing provider. Microsoft and Google have cloud offerings as well.

OpenStack is different because it's open source and developed by more than 200 companies including IBM, HP, and Rackspace. Each of these companies uses OpenStack to operate its own cloud offerings, sometimes with closed source add-ons. You could run your own cloud based on OpenStack, but you would lose most of the benefits outlined in section 1.3.

Comparing cloud providers isn't easy, because open standards are mostly missing. Functionality like virtual networks and message queuing are realized differently. If you know what features you need, you can compare the details and make your decision.

Otherwise, AWS is your best bet because the chances are highest that you'll find a solution for your problem.

Following are some common features of cloud providers:

- Virtual servers (Linux and Windows)
- Object store
- Load balancer
- Message queuing
- Graphical user interface
- Command-line interface

The more interesting question is, how do cloud providers differ? Table 1.2 compares AWS, Azure, Google Cloud Platform, and OpenStack.

Table 1.2 Differences between AWS, Microsoft Azure, Google Cloud Platform, and OpenStack

	AWS	Azure	Google Cloud Platform	OpenStack
Number of services	Most	Many	Enough	Few
Number of locations (multiple data centers per location)	9	13	3	Yes (depends on the OpenStack provider)
Compliance	Common standards (ISO 27001, HIPAA, FedRAMP, SOC), IT Grundschutz (Germany), G-Cloud (UK)	Common standards (ISO 27001, HIPAA, FedRAMP, SOC), ISO 27018 (cloud privacy), G-Cloud (UK)	Common standards (ISO 27001, HIPAA, FedRAMP, SOC)	Yes (depends on the OpenStack provider)
SDK languages	Android, Browsers (JavaScript), iOS, Java, .NET, Node.js (JavaScript), PHP, Python, Ruby, Go	Android, iOS, Java, .NET, Node.js (JavaScript), PHP, Python, Ruby	Java, Browsers (JavaScript), .NET, PHP, Python	-
Integration into development process	Medium, not linked to specific ecosystems	High, linked to the Microsoft ecosystem (for example, .NET development)	High, linked to the Google ecosystem (for example, Android)	-
Block-level storage (attached via network)	Yes	Yes (can be used by multiple virtual servers simultaneously)	No	Yes (can be used by multiple virtual servers simultaneously)
Relational database	Yes (MySQL, PostgreSQL, Oracle Database, Microsoft SQL Server)	Yes (Azure SQL Database, Microsoft SQL Server)	Yes (MySQL)	Yes (depends on the OpenStack provider)
NoSQL database	Yes (proprietary)	Yes (proprietary)	Yes (proprietary)	No
DNS	Yes	No	Yes	No

Table 1.2 Differences between AWS, Microsoft Azure, Google Cloud Platform, and OpenStack *(continued)*

	AWS	Azure	Google Cloud Platform	OpenStack
Virtual network	Yes	Yes	No	Yes
Pub/sub messaging	Yes (proprietary, JMS library available)	Yes (proprietary)	Yes (proprietary)	No
Machine-learning tools	Yes	Yes	Yes	No
Deployment tools	Yes	Yes	Yes	No
On-premises datacenter integration	Yes	Yes	Yes	No

In our opinion, AWS is the most mature cloud platform available at the moment.

1.6 Exploring AWS services

Hardware for computing, storing, and networking is the foundation of the AWS cloud. AWS runs software services on top of the hardware to provide the cloud, as shown in figure 1.9. A web interface, the API, acts as an interface between AWS services and your applications.

You can manage services by sending requests to the API manually via a GUI or programmatically via a SDK. To do so, you can use a tool like the Management Console, a web-based user interface, or a command-line tool. Virtual servers have a peculiarity: you can connect to virtual servers through SSH, for example, and gain administrator

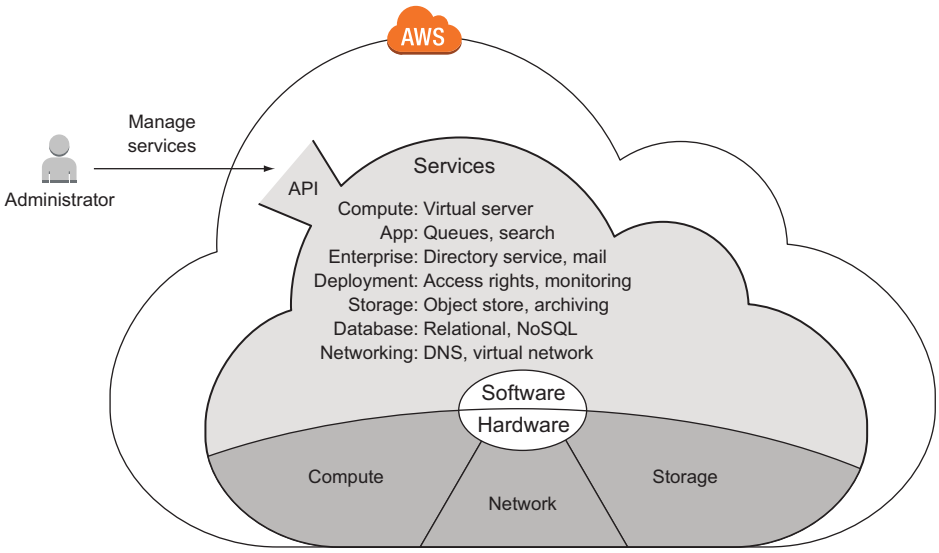


Figure 1.9 The AWS cloud is composed of hardware and software services accessible via an API.

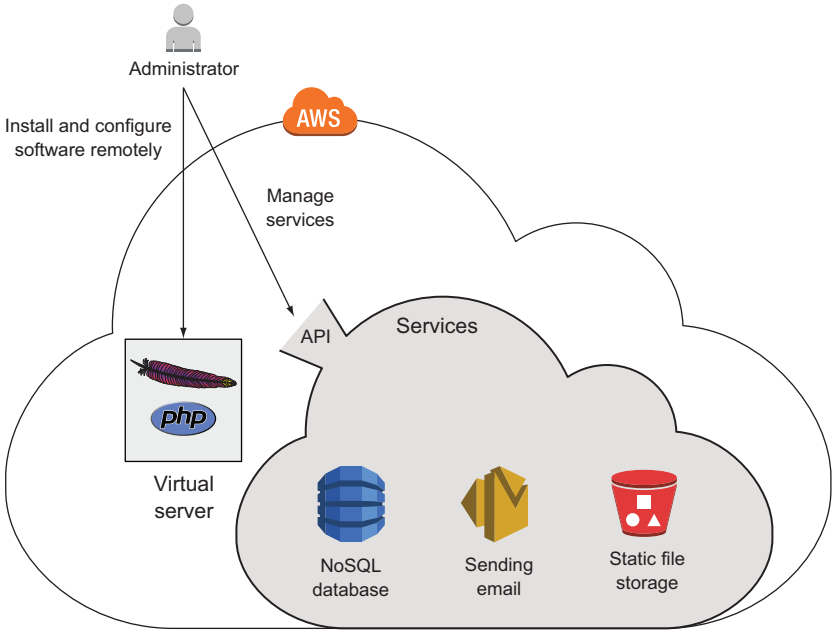


Figure 1.10 Managing a custom application running on a virtual server and dependent services

access. This means you can install any software you like on a virtual server. Other services, like the NoSQL database service, offer their features through an API and hide everything that's going on behind the scenes. Figure 1.10 shows an administrator installing a custom PHP web application on a virtual server and managing dependent services such as a NoSQL database used by the PHP web application.

Users send HTTP requests to a virtual server. A web server is installed on this virtual server along with a custom PHP web application. The web application needs to talk to AWS services in order to answer HTTP requests from users. For example, the web application needs to query data from a NoSQL database, store static files, and send email. Communication between the web application and AWS services is handled by the API, as figure 1.11 shows.

The number of different services available can be scary at the outset. The following categorization of AWS services will help you to find your way through the jungle:

- *Compute services* offer computing power and memory. You can start virtual servers and use them to run your applications.
- *App services* offer solutions for common use cases like message queues, topics, and searching large amounts of data to integrate into your applications.

- *Enterprise services* offer independent solutions such as mail servers and directory services.
- *Deployment and administration services* work on top of the services mentioned so far. They help you grant and revoke access to cloud resources, monitor your virtual servers, and deploy applications.
- *Storage* is needed to collect, persist, and archive data. AWS offers different storage options: an object store or a network-attached storage solution for use with virtual servers.
- *Database storage* has some advantages over simple storage solutions when you need to manage structured data. AWS offers solutions for relational and NoSQL databases.
- *Networking services* are an elementary part of AWS. You can define private networks and use a well-integrated DNS.

Be aware that we cover only the most important categories and services here. Other services are available, and you can also run your own applications on AWS.

Now that we've looked at AWS services in detail, it's time for you to learn how to interact with those services.

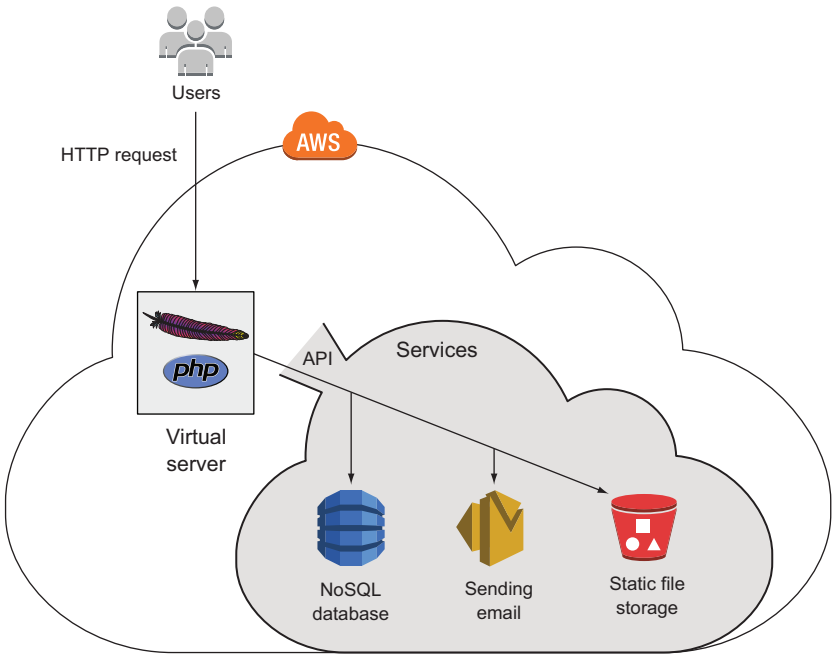


Figure 1.11 Handling an HTTP request with a custom web application using additional AWS services

1.7 Interacting with AWS

When you interact with AWS to configure or use services, you make calls to the API. The API is the entry point to AWS, as figure 1.12 demonstrates.

Next, we'll give you an overview of the tools available to make calls to the AWS API. You can compare the ability of these tools to automate your daily tasks.

1.7.1 Management Console

You can use the web-based Management Console to interact with AWS. You can manually control AWS with this convenient GUI, which runs in every modern web browser (Chrome, Firefox, Safari ≥ 5, IE ≥ 9); see figure 1.13.

If you're experimenting with AWS, the Management Console is the best place to start. It helps you to gain an overview of the different services and achieve success quickly. The Management Console is also a good way to set up a cloud infrastructure for development and testing.

1.7.2 Command-line interface

You can start a virtual server, create storage, and send email from the command line. With the command-line interface (CLI), you can control everything on AWS; see figure 1.14.

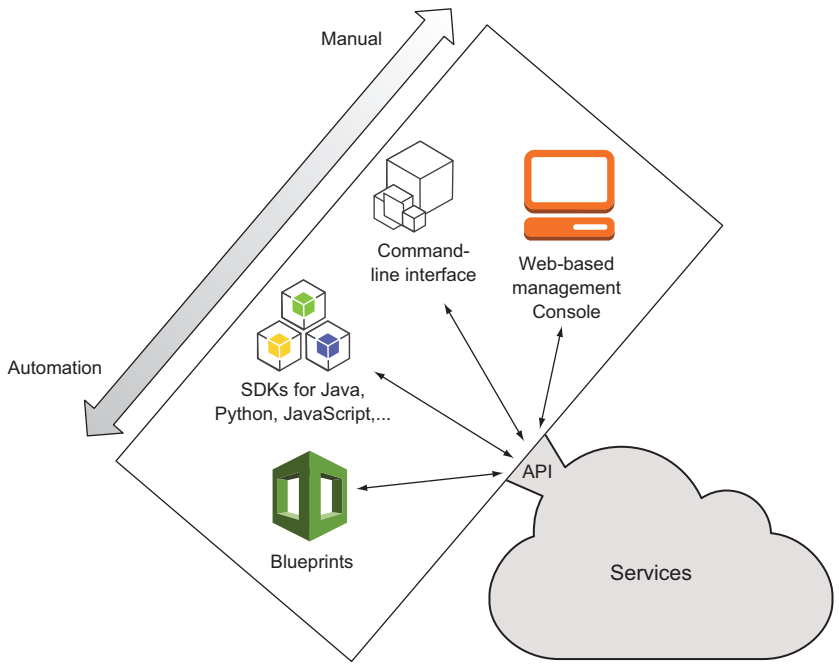


Figure 1.12 Tools to interact with the AWS API

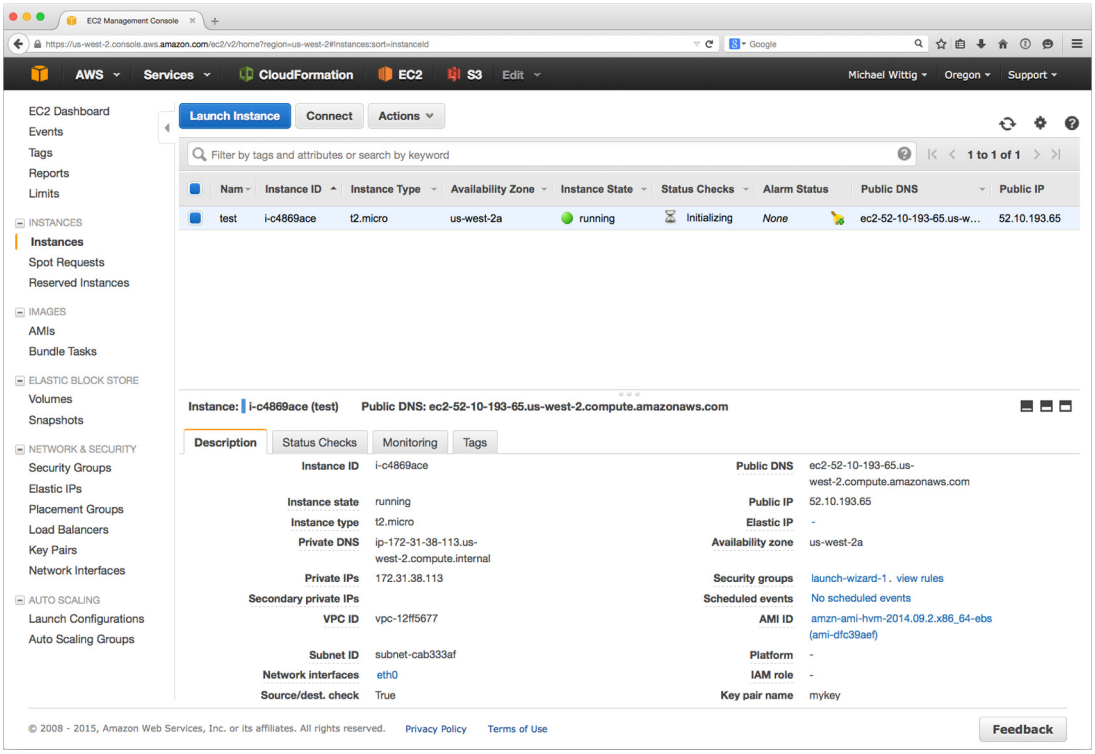


Figure 1.13 Management Console

Figure 1.14 Command-line interface

The CLI is typically used to automate tasks on AWS. If you want to automate parts of your infrastructure with the help of a continuous integration server like Jenkins, the CLI is the right tool for the job. The CLI offers a convenient way to access the API and combine multiple calls into a script.

You can even begin to automate your infrastructure with scripts by chaining multiple CLI calls together. The CLI is available for Windows, Mac, and Linux, and there's also a PowerShell version available.

1.7.3 SDKs

Sometimes you need to call AWS from within your application. With SDKs, you can use your favorite programming language to integrate AWS into your application logic. AWS provides SDKs for the following:

- Android
- Browsers (JavaScript)
- iOS
- Java
- .NET
- Node.js (JavaScript)
- PHP
- Python
- Ruby
- Go

SDKs are typically used to integrate AWS services into applications. If you're doing software development and want to integrate an AWS service like a NoSQL database or a push-notification service, an SDK is the right choice for the job. Some services, such as queues and topics, must be used with an SDK in your application.

1.7.4 Blueprints

A *blueprint* is a description of your system containing all services and dependencies. The blueprint doesn't say anything about the necessary steps or the order to achieve the described system. Figure 1.15 shows how a blueprint is transferred into a running system.

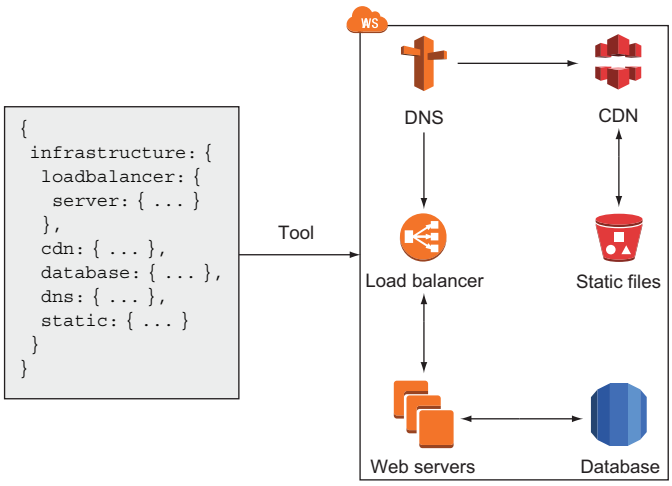


Figure 1.15 Infrastructure automation with blueprints

Consider using blueprints if you have to control many or complex environments. Blueprints will help you to automate the configuration of your infrastructure in the cloud. You can use blueprints to set up virtual networks and launch different servers into that network, for example.

A blueprint removes much of the burden from you because you no longer need to worry about dependencies during system creation—the blueprint automates the entire process. You'll learn more about automating your infrastructure in chapter 4.

It's time to get started creating your AWS account and exploring AWS practice after all that theory.

1.8 Creating an AWS account

Before you can start using AWS, you need to create an account. An AWS account is a basket for all the resources you own. You can attach multiple users to an account if multiple humans need access to the account; by default, your account will have one root user. To create an account, you need the following:

- A telephone number to validate your identity
- A credit card to pay your bills

> **Using an old account?**
>
> You can use your existing AWS account while working on the examples in this book. In this case, your usage may not be covered by the Free Tier, and you may have to pay for your usage.
>
> Also, if you created your existing AWS account before December 4, 2013, you should create a new one: there are legacy issues that may cause trouble when you try our examples.

1.8.1 Signing up

The sign-up process consists of five steps:

1. Provide your login credentials.
2. Provide your contact information.
3. Provide your payment details.
4. Verify your identity.
5. Choose your support plan.

Point your favorite modern web browser to https://aws.amazon.com, and click the Create a Free Account / Create an AWS Account button.

1. Providing your login credentials

The Sign Up page, shown in figure 1.16, gives you two choices. You can either create an account using your Amazon.com account or create an account from scratch. If you create the account from scratch, follow along. Otherwise, skip to step 5.

Fill in your email address, and select I Am a New User. Go on to the next step to create your login credentials. We advise you to choose a strong password to prevent misuse

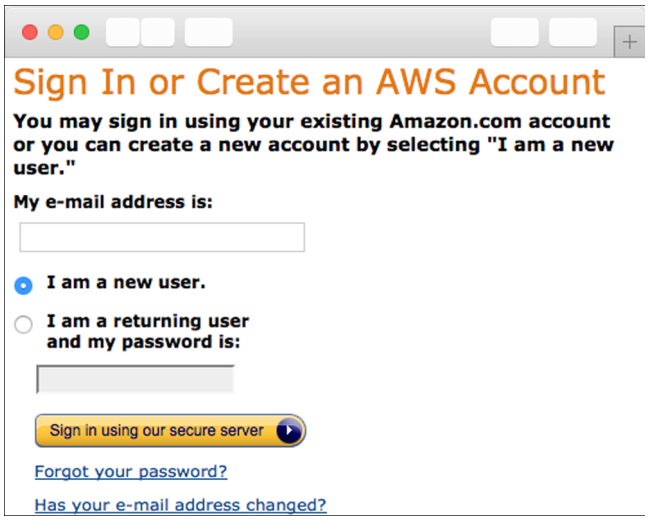


Figure 1.16 Creating an AWS account: Sign Up page

of your account. We suggest a password with 16 characters, numbers, and symbols. If someone gets access to your account, they can destroy your systems or steal your data.

2. PROVIDING YOUR CONTACT INFORMATION

The next step, as shown in figure 1.17, is to provide your contact information. Fill in all the required fields, and continue.

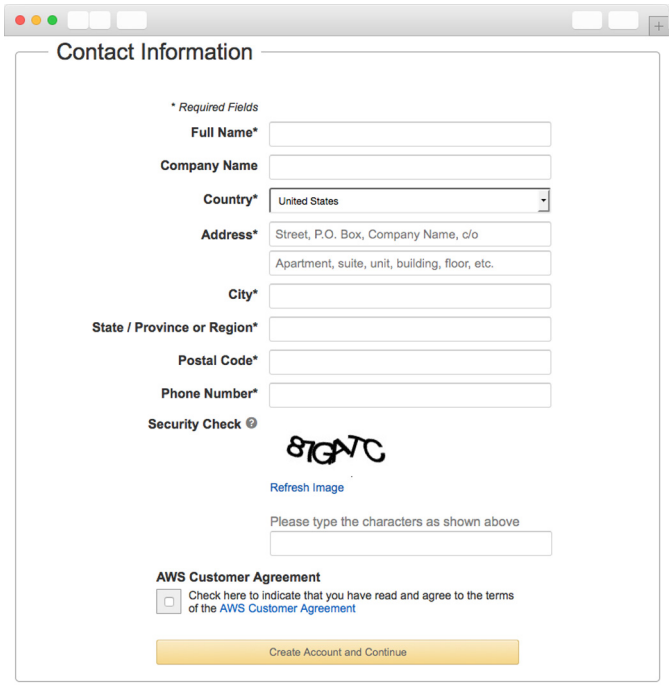


Figure 1.17 Creating an AWS account: providing your contact information

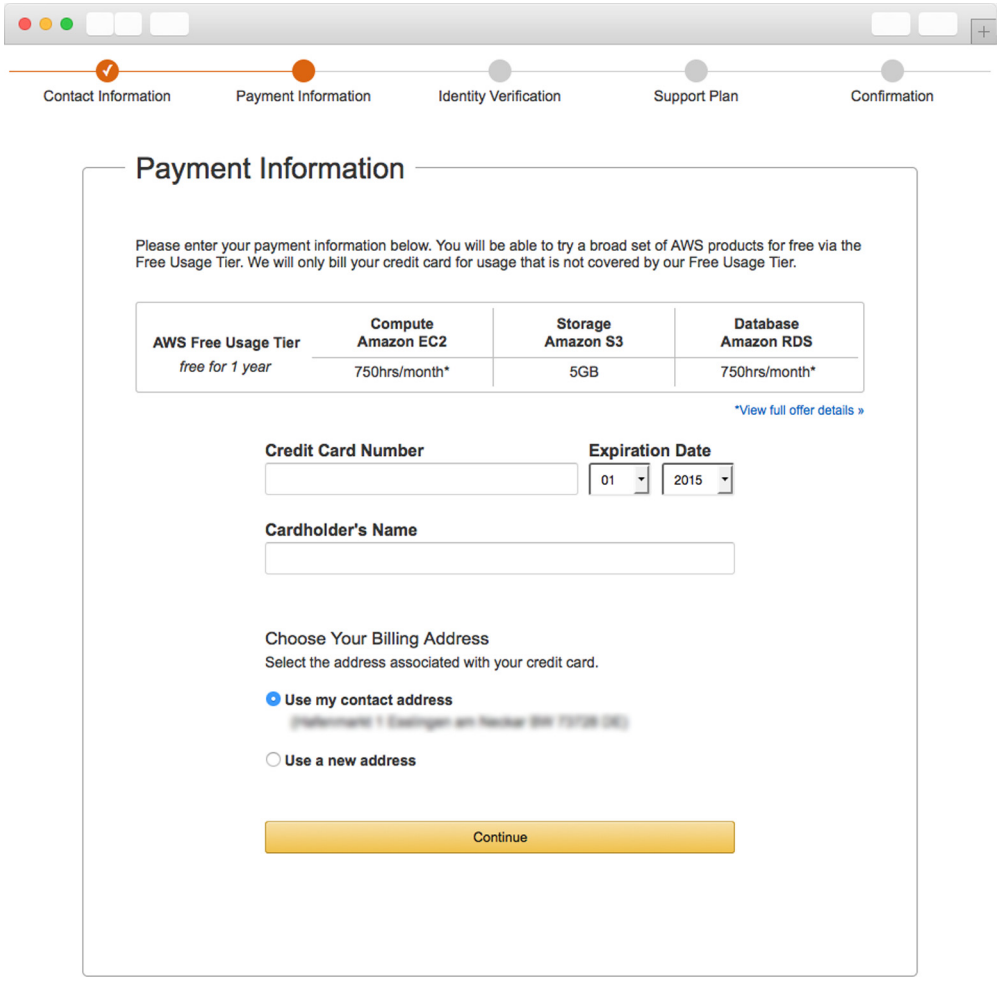


Figure 1.18 Creating an AWS account: providing your payment details

3. Provide your payment details

Now the screen shown in figure 1.18 asks for your payment information. AWS supports MasterCard and Visa. You can set your preferred payment currency later, if you don't want to pay your bills in USD; supported currencies are EUR, GBP, CHF, AUD, and some others.

4. Verifying your identity

The next step is to verify your identity. Figure 1.19 shows the first step of the process.

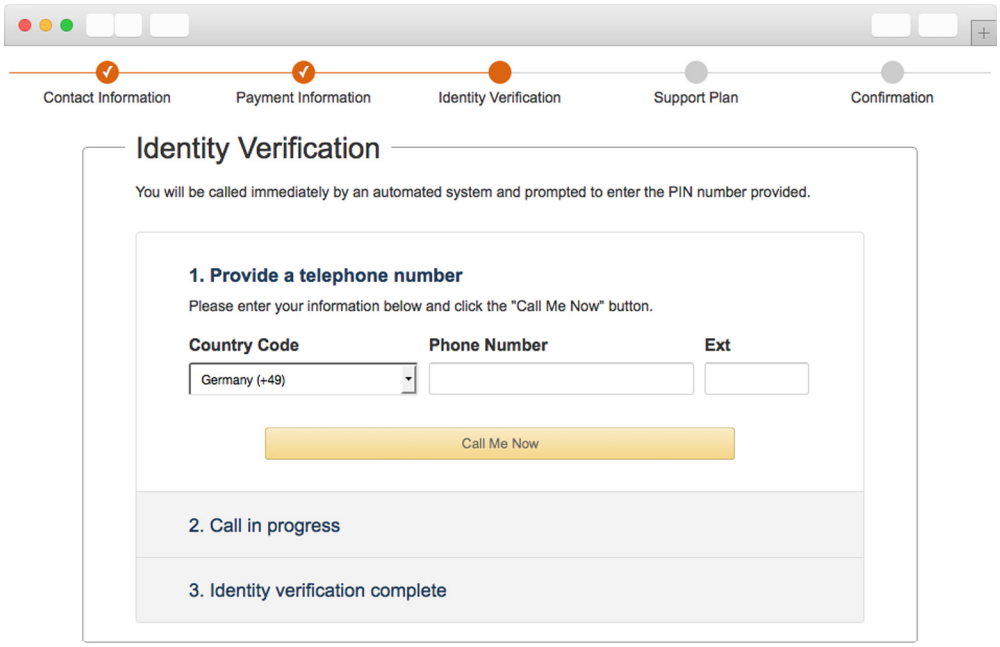


Figure 1.19 Creating an AWS account: verifying your identity (1 of 2)

After you complete the first part, you'll receive a call from AWS. A robot voice will ask you for your PIN, which will be like the one shown in figure 1.20. Your identity will be verified, and you can continue with the last step.

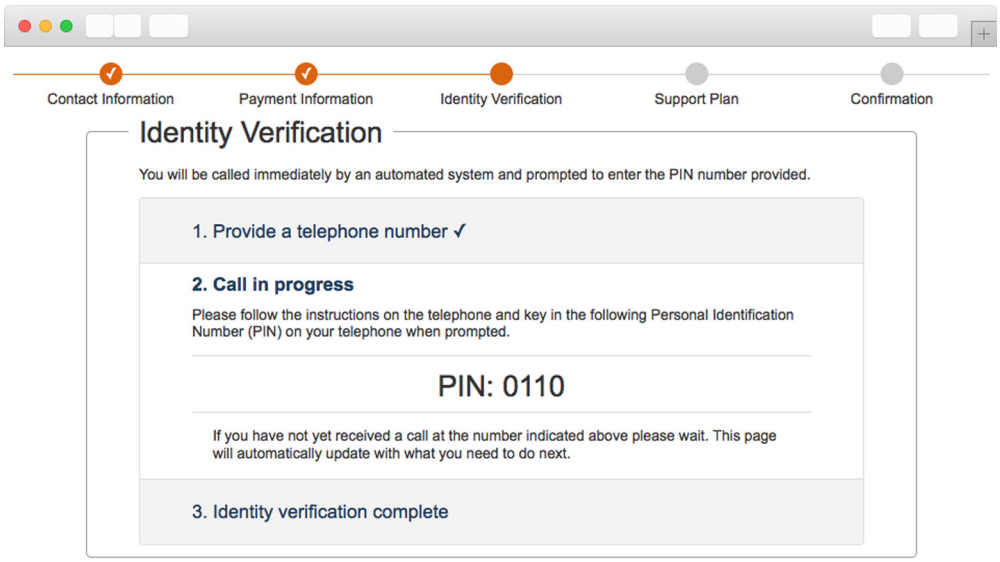


Figure 1.20 Creating an AWS account: verifying your identity (2 of 2)

5. Choosing your support plan

The last step is to choose a support plan; see figure 1.21. In this case, select the Basic plan, which is free. If you later create an AWS account for your business, we recommend the Business support plan. You can even switch support plans later.

High five! You're done. Now you can log in to your account with the AWS Management Console.

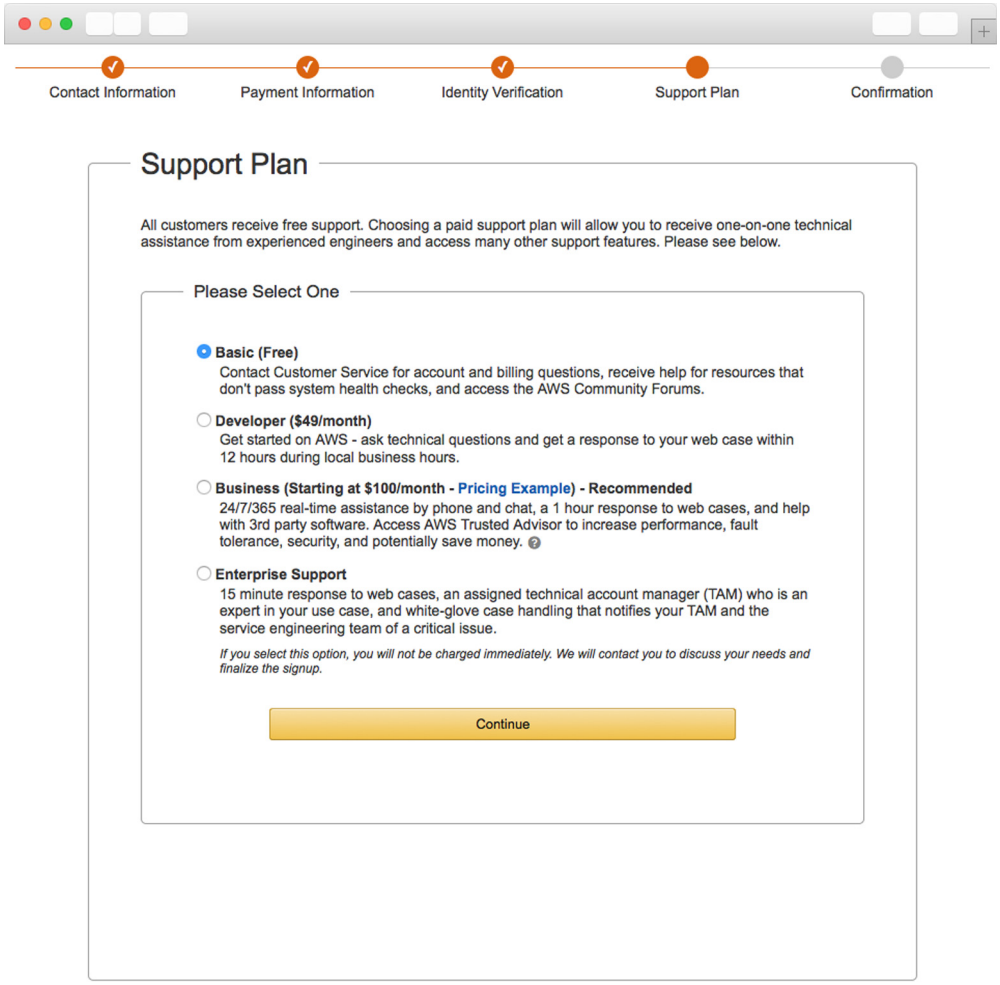


Figure 1.21 Creating an AWS account: choosing your support plan

1.8.2 Signing In

You have an AWS account and are ready to sign in to the AWS Management Console at https://console.aws.amazon.com. As mentioned earlier, the Management Console is a web-based tool you can use to control AWS resources. The Management Console

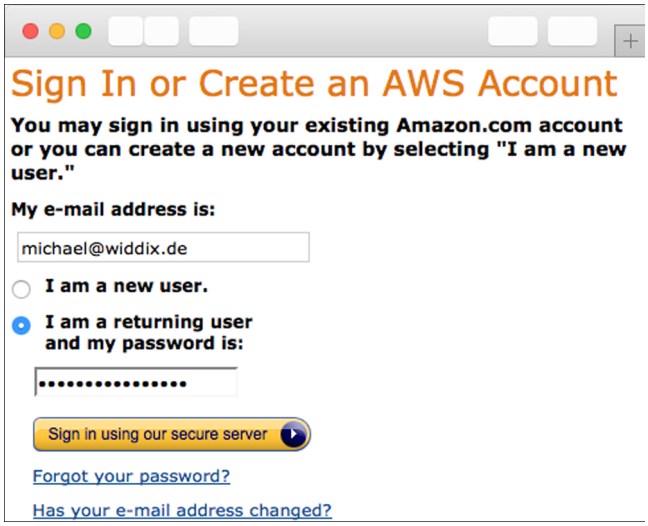


Figure 1.22 Sign in to the Management Console.

uses the AWS API to make most of the functionality available to you. Figure 1.22 shows the Sign In page.

Enter your login credentials and click Sign In Using Our Secure Server to see the Management Console, shown in figure 1.23.

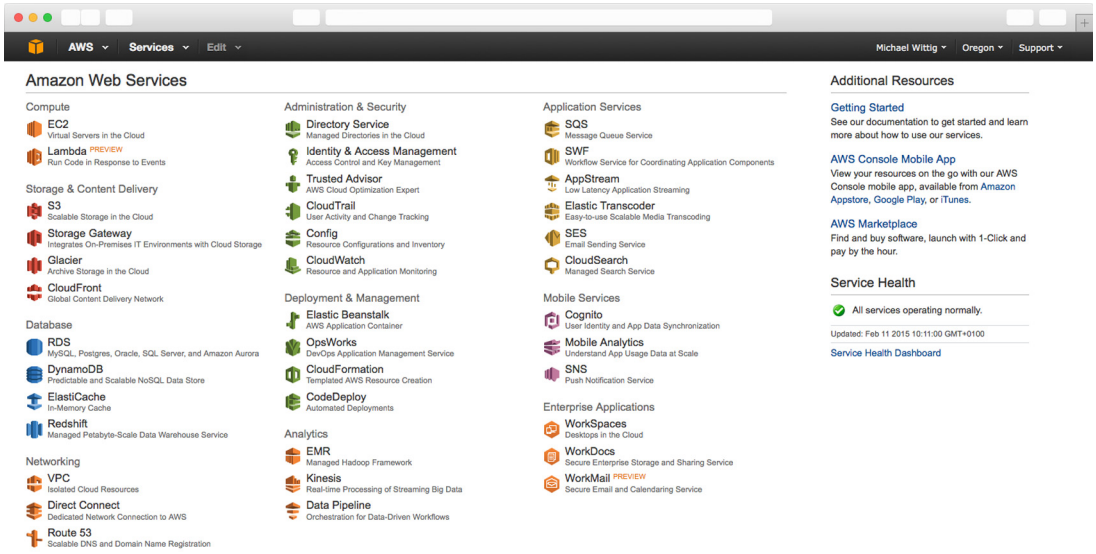


Figure 1.23 AWS Management Console

The most important part is the navigation bar at the top; see figure 1.24. It consists of six sections:

- *AWS*—Gives you a fast overview of all resources in your account.
- *Services*—Provides access to all AWS services.
- *Custom section (Edit)*—Click Edit and drag-and-drop important services here to personalize the navigation bar.
- *Your name*—Lets you access billing information and your account, and also lets you sign out.
- *Your region*—Lets you choose your region. You'll learn about regions in section 3.5. You don't need to change anything here now.
- *Support*—Gives you access to forums, documentation, and a ticket system.

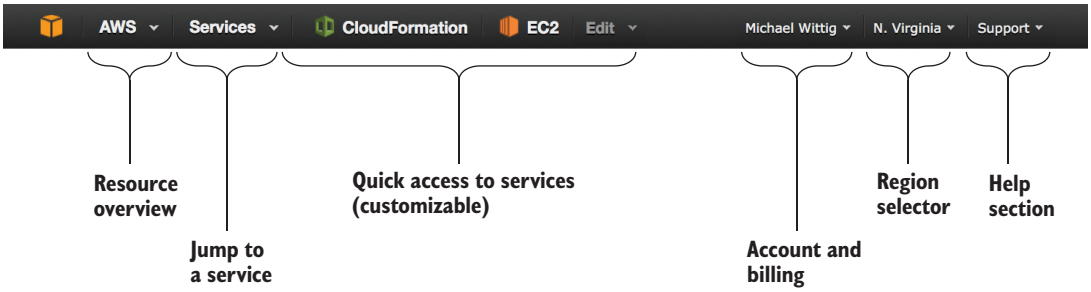


Figure 1.24 AWS Management Console navigation bar

Next, you'll create a key pair so you can connect to your virtual servers.

1.8.3 Creating a key pair

To access a virtual server in AWS, you need a *key pair* consisting of a private key and a public key. The public key will be uploaded to AWS and inserted into the virtual server. The private key is yours; it's like your password, but much more secure. Protect your private key as if it's a password. It's your secret, so don't lose it—you can't retrieve it.

To access a Linux server, you use the SSH protocol; you'll authenticate with the help of your key pair instead of a password during login. You access a Windows server via Remote Desktop Protocol (RDP); you'll need your key pair to decrypt the administrator password before you can log in.

The following steps will guide you to the dashboard of the EC2 service, which offers virtual servers, and where you can obtain a key pair:

1. Open the AWS Management Console at https://console.aws.amazon.com.
2. Click Services in the navigation bar, find the EC2 service, and click it.
3. Your browser shows the EC2 Management Console.

The EC2 Management Console, shown in figure 1.25, is split into three columns. The first column is the EC2 navigation bar; because EC2 is one of the oldest services, it has many

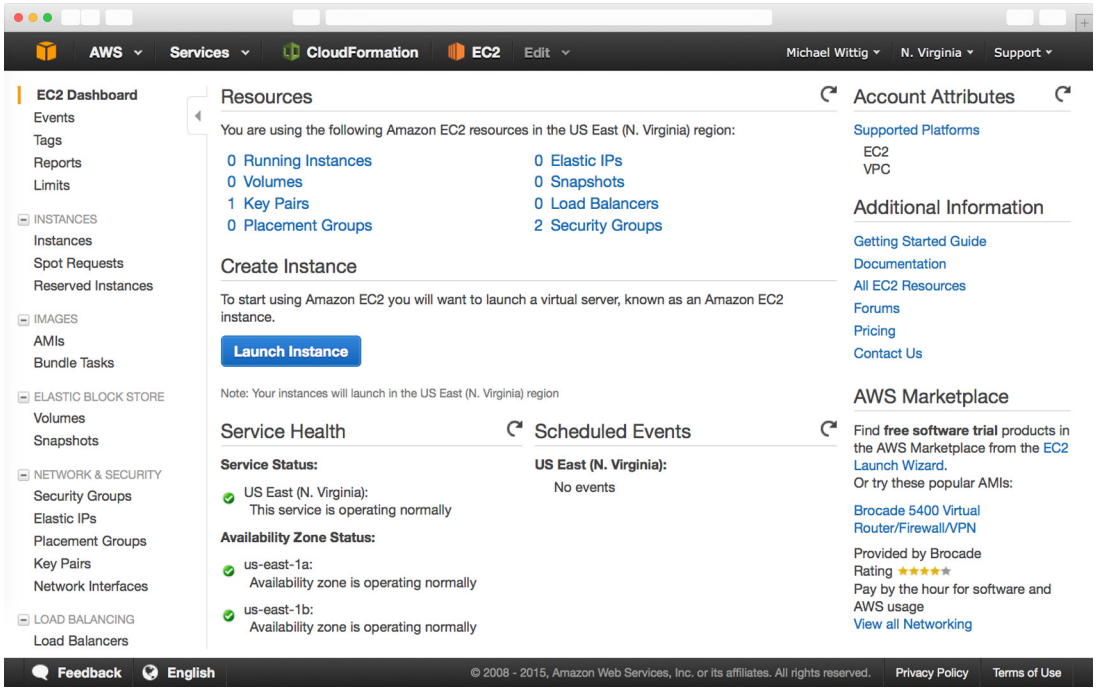


Figure 1.25 EC2 Management Console

features that you can access via the navigation bar. The second column gives you a brief overview of all your EC2 resources. The third column provides additional information.

Follow these steps to create a new key pair:

1. Click Key Pairs in the navigation bar under Network & Security.
2. Click the Create Key Pair button on the page shown in figure 1.26.
3. Name the Key Pair `mykey`. If you choose another name, you must replace the name in all the following examples!

During key-pair creation, you downloaded a file called mykey.pem. You must now prepare that key for future use. Depending on your operating system, you may need to do things differently, so please read the section that fits your OS.

Using your own key pair

It's also possible to upload the public key part from an existing key pair to AWS. Doing so has two advantages:

- You can reuse an existing key pair.
- You can be sure that only you know the private key part of the key pair. If you use the Create Key Pair button, AWS knows (at least briefly) your private key.

We decided against that approach in this case because it's less convenient to implement in a book.

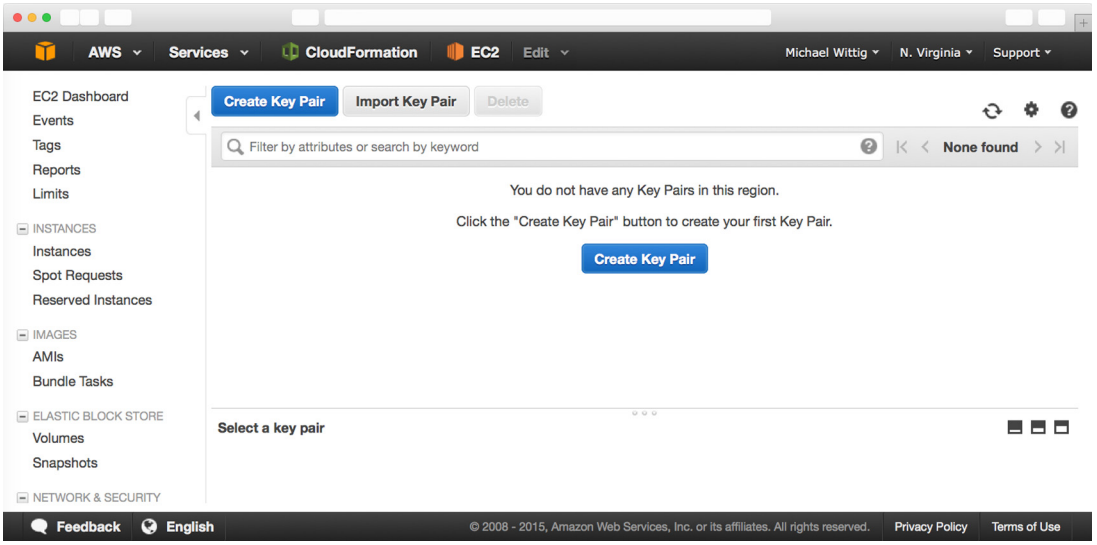


Figure 1.26 EC2 Management Console key pairs

Linux and Mac OS X

The only thing you need to do is change the access rights of mykey.pem so that only you can read the file. To do so, run `chmod 400 mykey.pem` in the terminal. You'll learn about how to use your key when you need to log in to a virtual server for the first time in this book.

Windows

Windows doesn't ship a SSH client, so you need to download the PuTTY installer for Windows from http://mng.bz/A1bY and install PuTTY. PuTTY comes with a tool called PuTTYgen that can convert the mykey.pem file into a mykey.ppk file, which you'll need:

1. Run the application PuTTYgen. The screen shown in figure 1.27 opens.
2. Select SSH-2 RSA under Type of Key to Generate.
3. Click Load.
4. Because PuTTYgen displays only *.pkk files, you need to switch the file extension of the File Name field to All Files.
5. Select the mykey.pem file, and click Open.
6. Confirm the dialog box.
7. Change Key Comment to `mykey`.
8. Click Save Private Key. Ignore the warning about saving the key without a passphrase.

Your .pem file is now converted to the .pkk format needed by PuTTY. You'll learn how to use your key when you need to log in to a virtual server for the first time in this book.

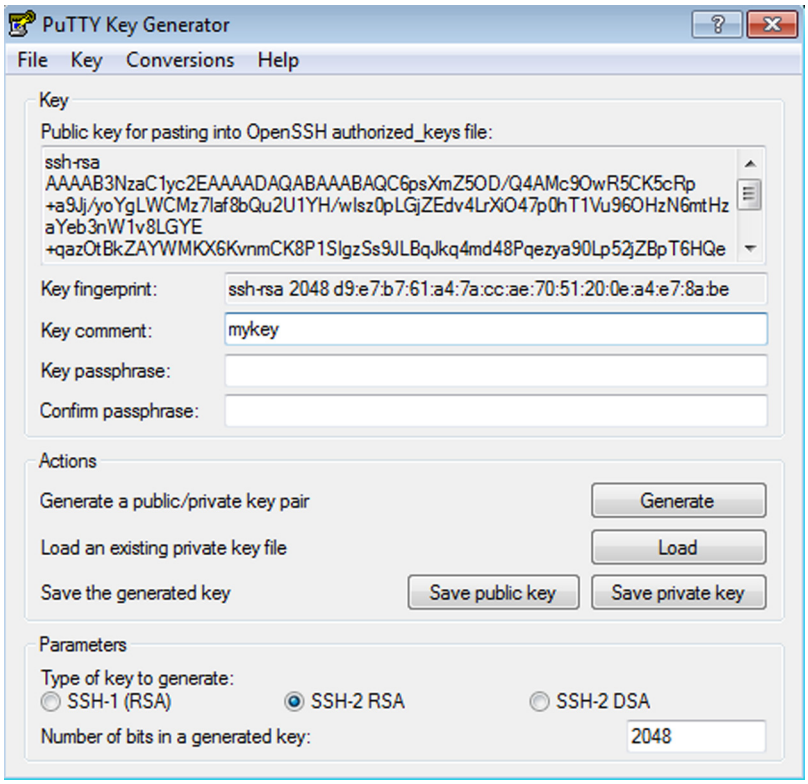


Figure 1.27 PuTTYgen allows you to convert the downloaded .pem file into the .pkk file format needed by PuTTY.

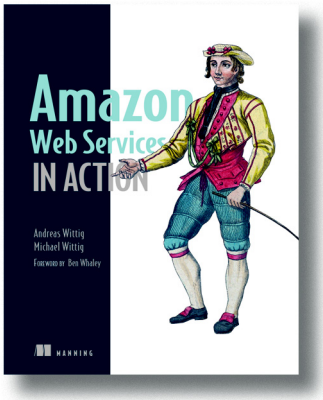
1.8.4 Creating a billing alarm

Before you use your AWS account in the next chapter, we advise you to create a billing alarm. If you exceed the Free Tier, an email is sent to you. The book warns you whenever an example isn't covered by the Free Tier. Please make sure that you carefully follow the cleanup steps after each example. To make sure you haven't missed something during cleanup, please create a billing alarm as advised by AWS: http://mng.bz/M7Sj.

1.9 Summary

- Amazon Web Services (AWS) is a platform of web services offering solutions for computing, storing, and networking that work well together.
- Cost savings aren't the only benefit of using AWS. You'll also profit from an innovative and fast-growing platform with flexible capacity, fault-tolerant services, and a worldwide infrastructure.
- Any use case can be implemented on AWS, whether it's a widely used web application or a specialized enterprise application with an advanced networking setup.

- You can interact with AWS in many different ways. You can control the different services by using the web-based GUI; use code to manage AWS programmatically from the command line or SDKs; or use blueprints to set up, modify, or delete your infrastructure on AWS.
- Pay-per-use is the pricing model for AWS services. Computing power, storage, and networking services are billed similarly to electricity.
- Creating an AWS account is easy. Now you know how to set up a key pair so you can log in to virtual servers for later use.



Amazon Web Services in Action introduces you to computing, storing, and networking in the AWS cloud. The book will teach you about the most important services on AWS. You'll also learn about best practices regarding security, high availability and scalability. You'll start with a broad overview of cloud computing and AWS and learn how to spin-up servers manually and from the command line. You'll learn how to automate your infrastructure by programmatically calling the AWS API to control every part of AWS. You'll be introduced to the concept of Infrastructure as Code with the help of AWS CloudFormation. You'll learn about different approaches to deploy applications on AWS. You'll also learn how to secure your infrastructure by isolating networks, controlling traffic and managing access to AWS resources. Next, you'll learn options and techniques for storing your data. You'll experience how to integrate AWS services into your own applications by the use of SDKs. Finally, this book teaches you how to design for high availability, fault tolerance, and scalability.

What's inside

- Overview of AWS cloud concepts and best practices
- Manage servers on EC2 for cost-effectiveness
- Infrastructure automation with Infrastructure as Code (AWS CloudFormation)
- Deploy applications on AWS
- Store data on AWS: SQL, NoSQL, object storage and block storage
- Integrate Amazon's pre-built services
- Architect highly available and fault tolerant systems

Written for developers and DevOps engineers moving distributed applications to the AWS platform.

Google Cloud Platform in Action

Install a common web application and you'll understand how a cloud provider works. JJ Geewax follows this approach in Chapter 2 of his book *Google Cloud Platform in Action*. You can also get a good understanding of the differences between cloud providers when you try to install the same application on each of them.

Chapter 2 from *Google Cloud Platform in Action*
by JJ Geewax

Trying it out: Deploying Wordpress on Google Cloud

This chapter covers

- What is Wordpress?
- Laying out the pieces of a Wordpress deployment
- Turning on a SQL database to store your data
- Turning on a VM to run Wordpress
- Turning everything off

The pieces we'll turn on here will be part of the free-trial from Google. If you leave them running past your free trial, your system will cost about $50 (?) per month.

If you've ever explored hosting your own website or blog, chances are you've come across (or maybe even installed) Wordpress. There's not much debate about Wordpress's popularity, with millions of websites relying on it for their websites or blogs, but many public blogs are hosted by other companies such as HostGator,

BlueHost, or Wordpress's own hosted service, WordPress.com (not to be confused with the open source project WordPress.org).

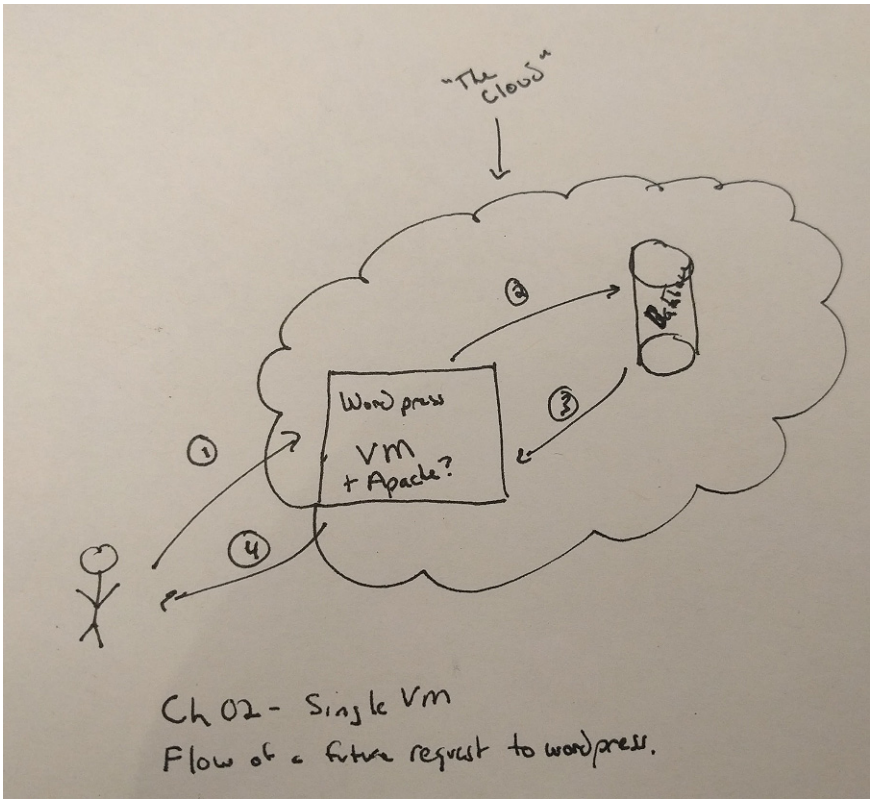
To demonstrate the simplicity of Google Cloud, this chapter is going to walk you through deploying Wordpress using Google Compute Engine and Google Cloud SQL to host your infrastructure.

First, let's put together an architectural plan for how we'll deploy Wordpress using all the cool new tools you learned in the last chapter.

2.1 Overall layout

Before we get down to the technical pieces of turning on machines, let's start by looking at what we need to turn on. The way we'll do this is to look at the flow of an "ideal request" through our future system. We're going to imagine a person visiting our future blog, and look at where their request needs to go in order to give them a great experience.

We'll start with a single machine, because it's the simplest possible configuration:



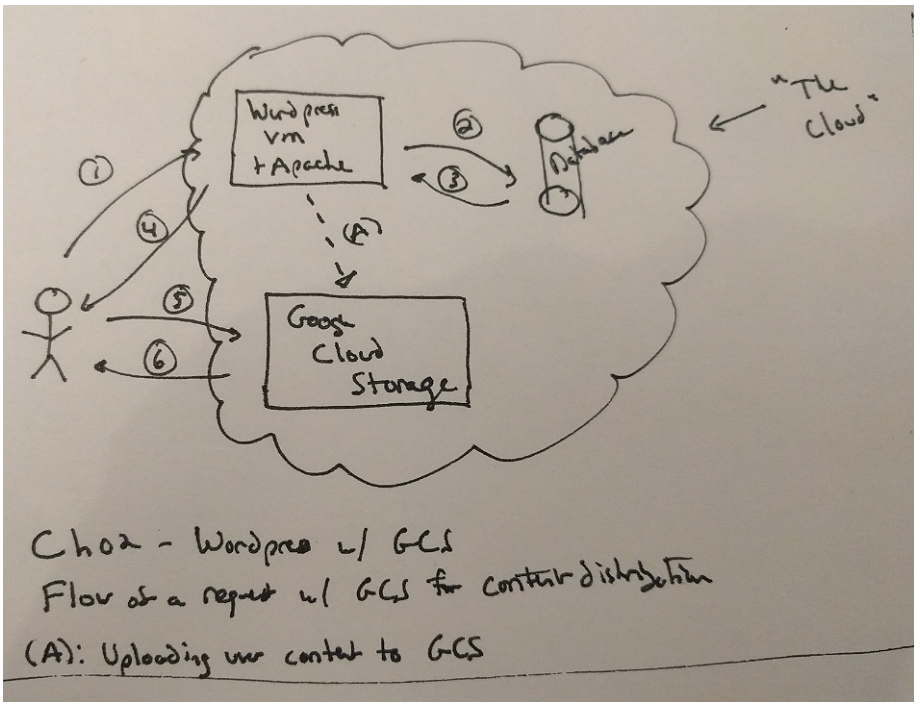
As you can see here, the flow is:

1. someone asks the Wordpress server for a page
2. the Wordpress server queries the database
3. the database sends back a result (ie, the content of the page)
4. the Wordpress server sends back a web page

Simple enough, right?

What happens as things get a bit more complex? What does that look like?

Although we won't demonstrate this configuration here, you might recall our discussion in Chapter 1 about relying on cloud services for more complicated hosting problems, like content distribution (for example, if your servers are in the US, what's the experience going to be like for your readers in Asia?). To show an idea of how this might look, here's a flow diagram for a Wordpress server using Google Cloud Storage to handle static content (like images):



In this case, the flow is initially the same, but unlike before, when static content is requested it doesn't re-use the same flow.

In this configuration, your Wordpress server will modify references to static content rather than request it from the Wordpress server, because the browser will request it from Google Cloud Storage (steps 5 and 6).

This means that requests for images and other static content will be handled by Google Cloud Storage directly, which can do fancy things like distributing your content around the world and caching the data close to your readers. This means that your static content will be delivered quickly, no matter how far they are from your Wordpress server.

Now that you've an idea of how the pieces will talk to each other, it's time to start exploring each piece individually and find out what is happening under the hood.

2.2 The database

We've drawn this picture involving a database, but we haven't said much about what type of database we're talking about. Many databases are available, but one of the most popular open-source databases is MySQL, which hopefully you've heard of. MySQL is great at storing relational data and has plenty of knobs to turn when you need to start squeezing out greater performance . For now, we're not concerned about performance, but it's nice to know that we'll have some wiggle room if things get bigger.

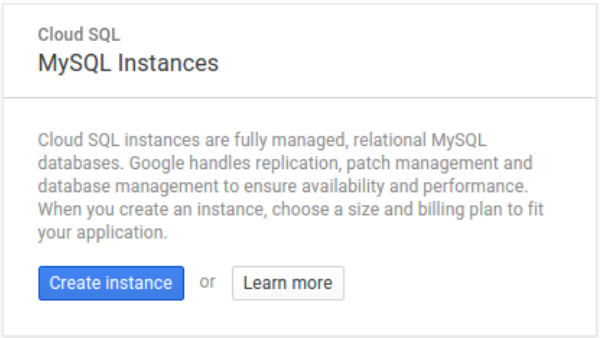
In the early days of cloud computing, the standard way to turn on a database like MySQL was to create a virtual machine, install the MySQL binary package, and then manage that virtual machine like any regular server. As time went on, cloud providers started noticing that databases all seemed to follow this same pattern, and they started offering "managed database services" where you don't have to configure the virtual machine yourself, but instead turn on a "managed virtual machine" running a specific binary.

All of the major cloud hosting providers offer this sort of service; Amazon has its Relational Database Service (RDS), Azure has its SQL Database service, and Google has its Cloud SQL service.

And because managing a database via Cloud SQL is quicker and easier than configuring and managing the underlying virtual machine and its software, we're going to use Cloud SQL for our database. This isn't always going to be the best choice (see Chapter X for much more detail about Cloud SQL), but for our Wordpress deployment, which is typical, Cloud SQL is a great fit as it looks almost identical to a MySQL server that you'd configure yourself, but is easier and faster to set up.

2.2.1 Turning on a Cloud SQL instance

The first step to turning on our database is to jump into the Cloud Console by going to https://cloud.google.com/console and then clicking on the Cloud SQL section. Once you get there, you'll see a nice blue button that says "Create instance":



After that, choose to create a "Second Generation" instance (see Chapter X for more detail on these) and then you'll be taken to a page where you can enter some information about your database.

The first thing you should notice is that this page looks similar to the page you saw when creating a virtual machine. This is intentional because, under the hood, you're creating a virtual machine which Google will manage, and kick-off by installing and configuring MySQL for you.

As with a virtual machine, you need to name your database. For this exercise, let's name the database `wordpress-db` (like VMs, the name must be unique inside your project, ensuring that only one database has this name at a time).

Then, also like a VM, you must choose where (geographically) you want your database to live. Like last time, let's use `us-central1-c` as our zone.

Let's leave the default instance type (1 vCPU with 3.75 GB of memory), but keep in mind that you'll be able to change this later if your discussion forum becomes popular enough that a single vCPU can't handle the load.

The next thing you should take notice of is the "storage capacity" field. This does what you might expect, in that it's the maximum amount of data you'll be able to store, but there's an important detail that might be new to you: size and performance are tied together.

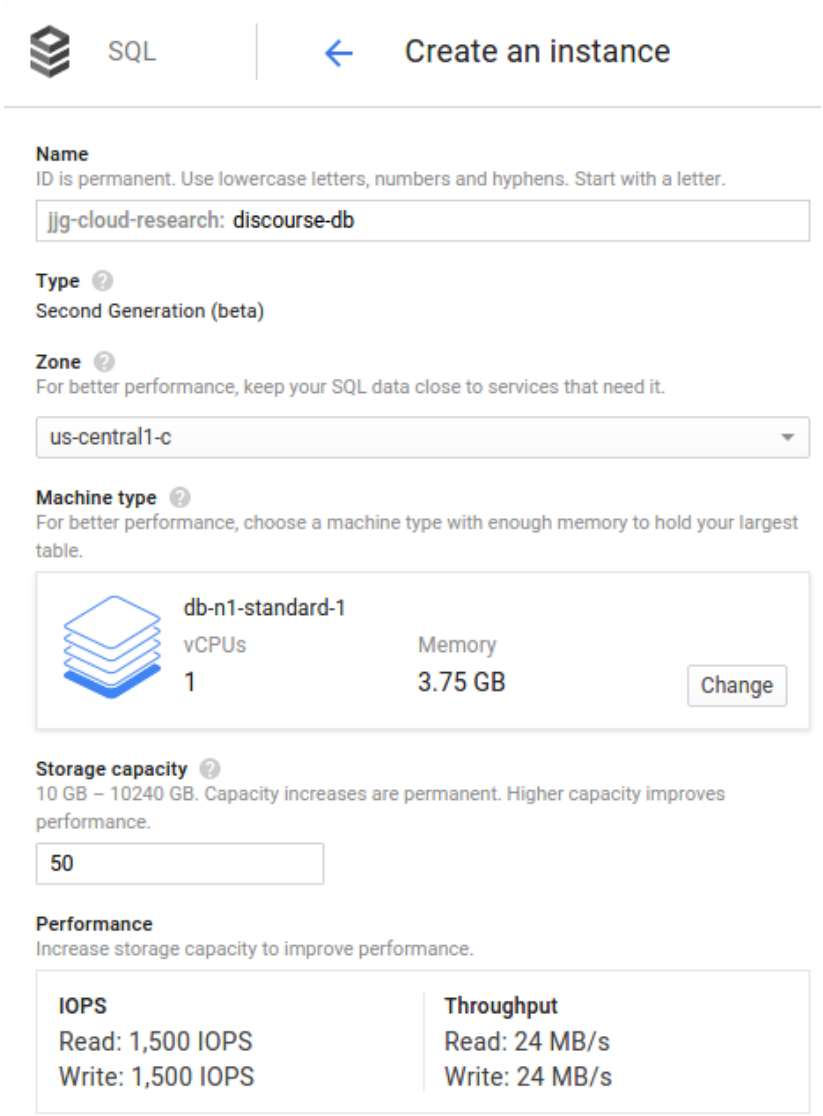
Size and performance are tied together? What does that mean?

This means that a bigger disk will be able to handle more read and write operations at a time. If you're an expert in storage, this'll seem obvious to you, but most of the time we think of storage in terms of size and rarely in terms speed. This is primarily because our day-to-day experience with disks includes more than enough "speed" for our needs, and performance isn't a concern. The problem shows up only when you have a disk being used by lots of people at the same time, as anyone with a large and overused network share will know. For our database, we'll certainly have lots of people asking for data at the same time, and we must consider speed as a factor.

What does this mean in real terms? How big of a disk do you need?

Because you can't choose performance and size as two separate things, you must choose a size that gives you the performance you need. This means that you'll often

have a disk that stores far more bytes than you need. This is OK, but if you're still worried, check out the chapter about Block Storage (Google's Persistent Disk) for more details. For this exercise, we should be perfectly safe with a 50GB disk:



The rest of the options on this page can be changed later. Let's leave them as they are and create our instance.

Once you've created your instance, you can use the `gcloud` command-line tool to show that it's all set with the `gcloud sql` command in Listing 2.1.

Listing 2.1 A list of running Cloud SQL instances

```
$ gcloud sql instances list
NAME          REGION  TIER              ADDRESS          STATUS
wordpress-db  -       db-n1-standard-1  104.197.207.227  RUNNABLE
```

Quiz

Can you think of a time that you might have a huge persistent disk that'll be mostly empty? Look at the chapter on Persistent Disk if you're not sure.

2.2.2 Securing your Cloud SQL instance

Before we go any further, you should change a few settings on your SQL instance to allow you (and hopefully only you) to connect to it. What we'll do for our testing phase is change the password on the instance, and then open it up to the world. Then, after we've tested it out, we'll change the network settings to only allow access from your Compute Engine VMs.

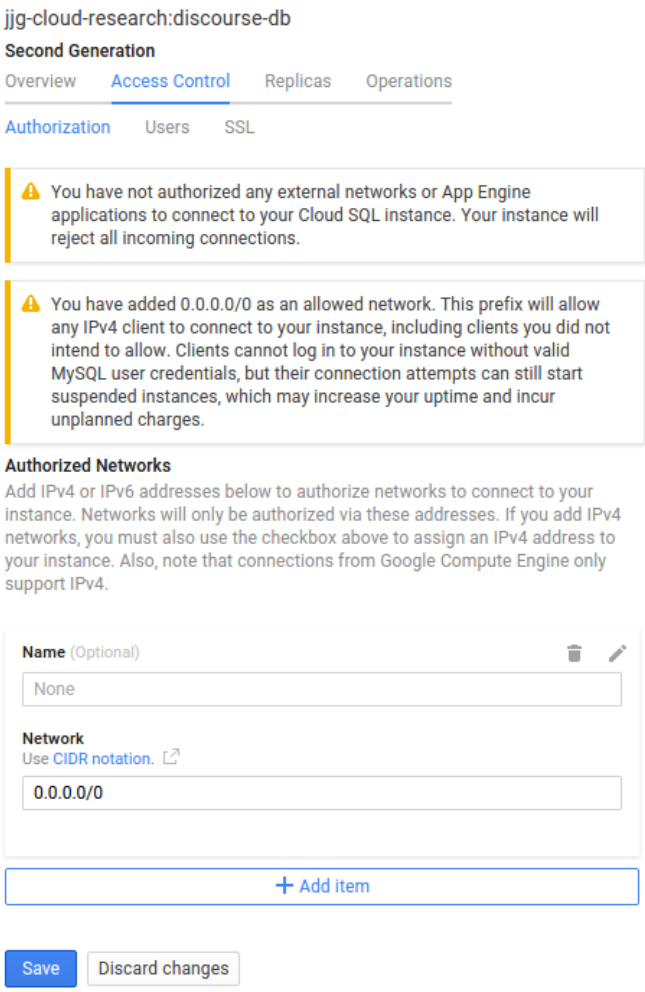
First, let's change the password. You can do this from the command line with the `gcloud sql set-root-password` command as shown in Listing 2.2.

Listing 2.2 Setting the root password for a Cloud SQL instance

```
$ gcloud sql instances set-root-password wordpress-db -p "my-very-long-
    password!"
Setting Cloud SQL instance password...done.
Set password for [https://www.googleapis.com/sql/v1beta3/projects/jjg-cloud-
    research/instances/wordpress-db].
```

Now that the password's set, let's (temporarily) open the SQL instance to the outside world.

To do this, go into the Cloud Console and navigate to your Cloud SQL instance. Towards the top are a few tabs; one of them is "Access Control". Under the Access Control tab, there's a bit button labelled "Add item" which allows you to control your "Authorized Networks". Click on that and add "the world" in CIDR notation (`0.0.0.0/0`) and click Save:



Caution

You'll notice a little warning about opening your database to any IP address. This is OK because we're doing some testing, but **you** *should never leave this setting for your production environments*. You'll learn more about securing your SQL instance for your cluster later on.

Now it's time to test whether all of this worked!

2.2.3 Connecting to your Cloud SQL instance

If you don't have a MySQL client, the first thing to do is install one. On a Linux environment like Ubuntu you can install it by typing

Listing 2.3 Installing the MySQL client library on Ubuntu Linux

```
$ sudo apt-get install mysql-client
```

On Windows or Mac, you can download the package from MySQL's website: http://dev.mysql.com/downloads/mysql/

After you've that installed, connecting to the database is easy: enter the IP address of your instance (you saw this before with `gcloud sql instances list`), use the username "root", and the password you set earlier.

For example, on a Linux computer, here's what you might do:

Listing 2.4 Connecting to a Cloud SQL instance using the MySQL client

```
$ mysql -h 104.197.207.227 -u root -p
Enter password: # <I typed my password here>
Welcome to the MySQL monitor.Commands end with ; or \g.
Your MySQL connection id is 4760

Server version: 5.6.25-google-log (Google)
Copyright (c) 2000, 2015, Oracle and/or its affiliates. All rights reserved.
Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
mysql>
```

Great work! You now have a SQL database server living in the cloud!

Next, let's run a few SQL commands to prepare your database for Wordpress.

2.2.4 Configuring your Cloud SQL instance for Wordpress

You've a MySQL database that you can successfully talk to. Let's get that database prepared for Wordpress to start talking to it.

Here's a basic outline of what we're going to do:

1. Create a database called `wordpress`
2. Create a user called `wordpress`
3. Give the `wordpress` user the appropriate permissions

The first thing you should do is get back to your MySQL command-line prompt. As you learned, you can do this by running the `mysql` command.

Next up is to create the database. Do this by running

Listing 2.5 Creating the `wordpress` database in MySQL

```
mysql> CREATE DATABASE wordpress;
Query OK, 1 row affected (0.10 sec)
```

Then we need to create a user account for Wordpress to use for access to the database:

Listing 2.6 Creating the `wordpress` user in MySQL

```
mysql> CREATE USER wordpress IDENTIFIED BY 'very-long-wordpress-password';
Query OK, 0 rows affected (0.21 sec)
```

And then we need to give this new user the right level of access to do things to the database (like create tables, add rows, run queries, etc):

Listing 2.7 Granting access to the `wordpress` user in MySQL

```
mysql> GRANT ALL PRIVILEGES ON wordpress.* TO wordpress;
Query OK, 0 rows affected (0.20 sec)
```

And finally let's tell MySQL to reload the list of users and privileges. If we forget this command, MySQL would know about the changes when it restarts, but we don't want to restart our Cloud SQL instance for this alone!

Listing 2.8 Refreshing the database's access controls after changes

```
mysql> FLUSH PRIVILEGES;
Query OK, 0 rows affected (0.12 sec)
```

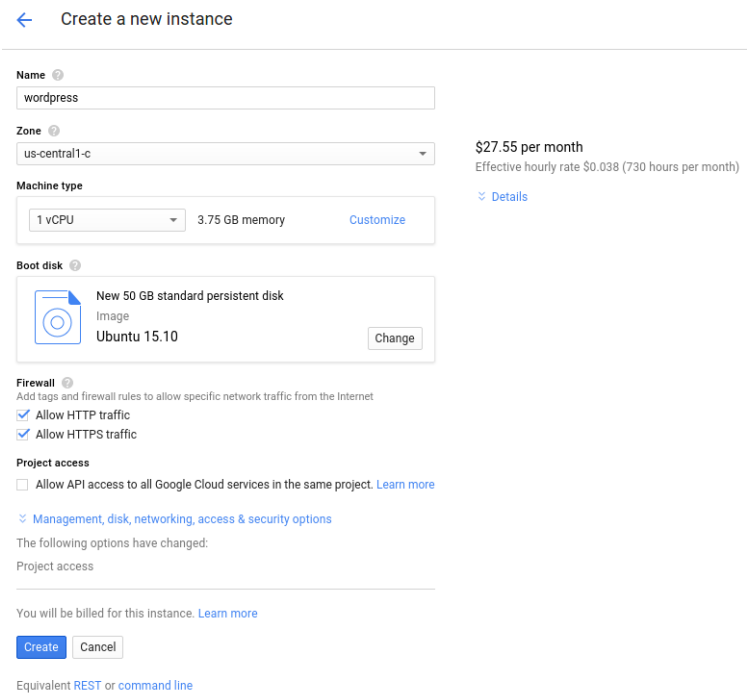
This is all you need to do on the database! Next, let's make it do something real rather than sit around as a toy.

Quiz

How does your database get backed up? Look at the chapter on Cloud SQL if you're not sure.

2.3 Deploying the Wordpress VM

Let's start by turning on the VM that'll host our Wordpress installation. As you learned previously, you can do this easily in the Cloud Console. Go ahead and do that once more:



Take note that the checkboxes for allowing HTTP and HTTPS traffic are checked, because we want our Wordpress server to be accessible to anyone through their browsers.

Before you click Create, you need to do one last thing. Click on "Management, disk, networking, access & security options" which'll expand to show new options. Under "Access & security", scroll down to the section called Cloud SQL and change the dropdown from "None" to "Enabled".

After that, you're ready to turn on your VM. Go ahead and click "Create".

Quiz

- Where does your virtual machine physically exist?
- What will happen if the hardware running your virtual machine has a problem?

Look at the chapter on the Cloud Datacenter if you're not sure.

2.4 Configuring Wordpress

The first thing to do now that your VM is up and running is to SSH into it. You can do this in the Cloud Console by clicking the SSH button, or use the Cloud SDK with the `gcloud compute ssh` command. For this walkthrough, we'll use the Cloud SDK to connect to our VM.

```
$ gcloud compute ssh us-central1-c/wordpress
Warning: Permanently added '104.197.86.115' (ECDSA) to the list of known
     hosts.
Welcome to Ubuntu 15.10 (GNU/Linux 4.2.0-18-generic x86_64)

 * Documentation:  https://help.ubuntu.com/

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

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

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

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

jj@wordpress:~$
```

Once you're connected, we need to install a bunch of packages:

1. Apache
2. MySQL Client
3. PHP

You can do this using `apt-get`:

```
jj@wordpress:~$ sudo apt-get update
jj@wordpress:~$ sudo apt-get install apache2 mysql-client php5-mysql php5
libapache2-mod-php5 php5-mcrypt php5-gd libssh2-php
```

When prompted, confirm (by typing `Y` and hitting enter).

Now that you've installed all the prerequisites, it's time to install Wordpress. Start by downloading the latest version from wordpress.org, and unzipping it into your home directory.

```
jj@wordpress:~$ wget http://wordpress.org/latest.tar.gz
jj@wordpress:~$ tar xzvf latest.tar.gz
```

After you've done that, you'll need to set some configuration parameters, primarily where Wordpress should store data, and how to authenticate. To do this, copy the sample config file to `wp-config.php`, and then edit the file to point to your Cloud SQL instance (in this example, I'm using vim, but you can use whichever text editor you're most comfortable with):

```
jj@wordpress:~$ cd wordpress
jj@wordpress:~/wordpress$ cp wp-config-sample.php wp-config.php
jj@wordpress:~/wordpress$ vim wp-config.php
```

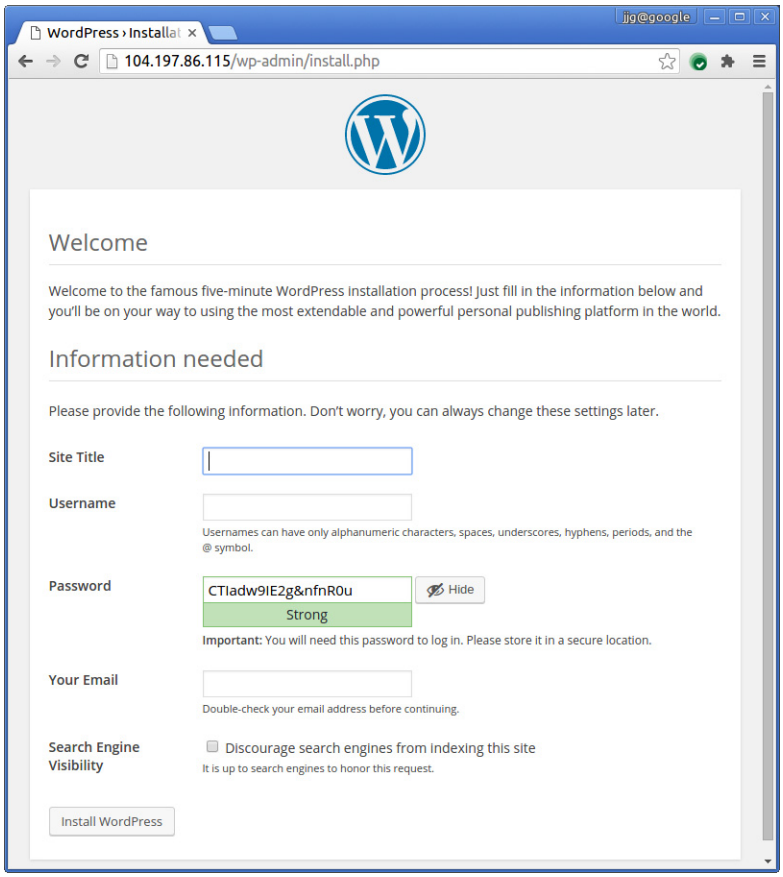
After editing `wp-config.php`, it should look something like this:

```
<?php
/**
* The base configuration for Wordpress
*
* The wp-config.php creation script uses this file during the
* installation. You don't have to use the web site, you can
* copy this file to "wp-config.php" and fill in the values.
*
* This file contains the following configurations:
*
* * MySQL settings
* * Secret keys
* * Database table prefix
* * ABSPATH
*
* @link https://codex.wordpress.org/Editing_wp-config.php
*
* @package WordPress
*/

// ** MySQL settings - You can get this info from your web host ** //
/** The name of the database for WordPress */
define('DB_NAME', 'wordpress');

/** MySQL database username */
define('DB_USER', 'wordpress');

/** MySQL database password */
define('DB_PASSWORD', 'very-long-wordpress-password');

/** MySQL hostname */
define('DB_HOST', '104.197.207.227');

/** Database Charset to use in creating database tables. */
define('DB_CHARSET', 'utf8');

/** The Database Collate type. Don't change this if in doubt. */
define('DB_COLLATE', '');
```

Once you have your configuration the way you want it (you should only need to change the database settings), it's time to move all those files out of your home directory and into somewhere that Apache can serve them. (We also need to remove the Apache default page, `index.html`) The easiest way to do this is using `rm` and then `rsync`:

```
jj@wordpress:~/wordpress$ sudo rm /var/www/html/index.html
jj@wordpress:~/wordpress$ sudo rsync -avP ~/wordpress/ /var/www/html/
```

Now all you must do is visit your web server in your browser, which should end up looking like this:



From there, following the prompts should take about five minutes and you'll have a working Wordpress installation!

2.5 Review the system

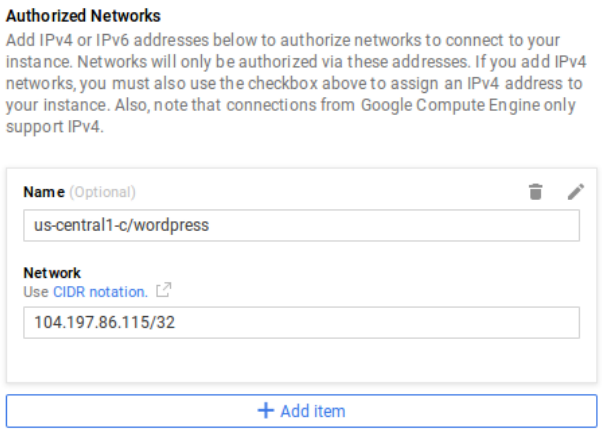
What did we do? Looking at this in order, we setup quite a few different pieces:

1. We turned on a Cloud SQL instance to store our data
2. We added a few users, and changed the security rules.
3. We turned on a Compute Engine virtual machine
4. We installed Wordpress on that VM Did we forget anything?

Do you remember when we set the security rules on the Cloud SQL instance to accept connections from anywhere (`0.0.0.0/0`)? Now that we know where to accept

requests (from our VM), we should fix that. If we don't the database is vulnerable to attacks from the whole world. But, if we lock down the database at the network level, even if someone discovers the password it's only useful if they're connecting from one of our known machines.

To do this, head back to the Cloud Console and navigate to your Cloud SQL instance. Under the "Access Control" tab, edit the Authorized Network you had before, changingn0.0.0.0/0 to the IP address followed by `/32` (ie, `104.197.86.115/32`) and rename the rule to read `us-central1-c/wordpress` to remember what this rule is for. When you're done, the access control rules should look like this:



2.6 Turning it off

If you want to keep your Wordpress instance up and running, you can skip past this section. (Maybe you've always wanted to host your own blog, and the demo we picked happened to be perfect for you?) If not, let's go through the process of turning off all those resources you created.

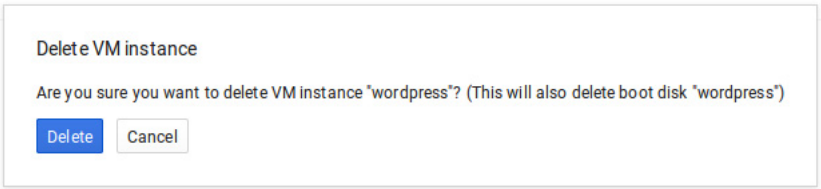
The first thing to turn off is the GCE virtual machine. You can do this using the Cloud Console in the Compute Engine section. When you click on your instance, you might notice there are two options, "Stop" and "Delete". The difference between these is subtle but important.

When you **delete** an instance, it's gone forever, like it never existed in the first place. When you **stop** an instance, it's still there, but in a "paused" state where you can pick up exactly where you left off.

Why wouldn't we stop instances rather than delete them? The catch with stopping an instance is that you must keep your persistent disks around, and those cost money. You won't be paying for CPU cycles on a stopped instance, but the disk that stores the operating system and all your configuration needs to stay around, and you'll be billed for your disks whether they're attached to a running virtual machine or not.

In this case, if we're done with our Wordpress installation, the right choice is probably deleting rather than stopping.

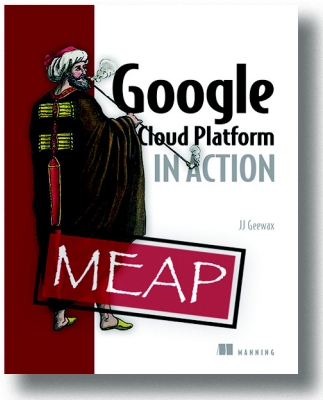
When you click delete, you should notice that the confirmation prompt reminds you that your disk (the boot disk) will also be deleted.



After that, you can do the same thing to your Cloud SQL instance, but keep in mind you don't have the option to "stop" a Cloud SQL instance.

2.7 Summary

- Google Compute Engine makes it simple and fast to turn on virtual machines in the cloud.
- Larger persistent disks come with more performance (IOPS); choose a size that meets your needs. (It's OK if your disk has empty space.)
- If you don't need any special customization, Google Cloud SQL is a simple and fast way to turn on a managed MySQL database, using Compute Engine under the hood.
- You can connect to Cloud SQL databases using the normal MySQL client, and there's no need for anything special in your software.
- Use Google's access control to limit connections at the network level.
- Never open your production database to the world (`0.0.0.0/0`).



Cloud services make it easy to set up technical infrastructure (such as computing resources or storage capacity) without those pesky long-term commitments, meaning you pay for exactly what you need, and can think beyond the limits of physical machines. Many cloud providers are out there to choose from, Google Cloud Platform's trusted by millions of applications and is backed by the infrastructure powering many of the services you use every day such as Google Search and YouTube. It's no wonder people are flocking to cloud providers in general, and GCP in particular, as these unique services let you focus less on your tools and more on your application.

Google Cloud Platform in Action teaches you to build and launch applications that scale, using the many services on GCP to move faster than ever. You'll begin with an introduction to what cloud services are in general, and a close up look at GCP and its benefits. You'll quickly get to deploy a basic application on GCP with step-by-step instructions. Then you'll move on to more advanced topics, such as how to architect your application to best take advantage of GCP services, how to launch large-scale web applications, how best to store and query huge amounts of data, and how to build the back-end of a social image sharing application. You'll also learn how to choose exactly the services that best suit your needs. By the end, you'll be able to build applications that run on Google Cloud Platform and start quicker, suffer fewer disasters, and require less maintenance.

What's inside

- Use Google's core infrastructure, data analytics and machine learning
- Get applications deployed quickly
- Choosing the most cost effective options
- Hands-on code examples

This book is for developers who've some experience developing web applications. No experience with cloud services required.

Serverless Architectures on AWS

A serverless architecture's about composing functions to take in an input and return an output. These functions are the new building blocks of your architecture. Each function's its own deployable unit that's independent of all the other functions. By passing inputs to other functions, they can communicate with each other. But functions can also be triggered by the external world. Follow a chapter of Serverless Architectures and start going Serverless!

Chapter 1 from *Serverless Architectures on AWS*
by Peter Sbarski with Sam Kroonenburg

Going serverless

This chapter covers

- Traditional system and application architectures
- Key characteristics of serverless architectures and their benefits
- How serverless architectures and microservices fit in to the picture
- Considerations when transitioning from server to serverless

If you ask software developers what software architecture is, you might get answers ranging from "it's a blueprint or a plan" to "a conceptual model" to "the big picture." It's undoubtedly true that architecture, or lack thereof, can make or break software. Good architecture may help to scale a web or mobile application, and poor architecture may cause serious issues that necessitate a costly rewrite. Understanding the

implication of choice regarding architecture and being able to plan ahead is paramount to creating effective, high-performing, and ultimately successful software systems.

This book is about how to go beyond traditional back end architectures that require us to interact with a server in some shape or form. It describes how to create *serverless* back ends that rely entirely on a compute service such as Amazon Web Services (AWS) Lambda and an assortment of useful third-party APIs, services, and products. It shows how to build the next generation of systems that can scale and handle demanding computational requirements without having to provision or manage a single machine. Importantly, this book describes techniques that can help developers quickly deliver products to market while maintaining a high-level of quality and performance by leveraging services and architectures that today's cloud has to offer.

The first chapter of this book is about why we think serverless is a game changer for software developers and solution architects. This chapter introduces key services such as AWS Lambda and describes the principles of serverless architecture to help you understand what makes a true serverless system.

What's in a name?

Before we start, we should mention that the word *serverless* is a bit of a misnomer. Whether you use a compute service such as AWS Lambda to execute your code or interact with an API, there are still servers running in the background. The difference is that these servers are hidden from you. There's no infrastructure for you to think about and no way to tweak the underlying operating system. Someone else takes care of the nitty-gritty details of infrastructure management, freeing your time for other things. Serverless is about running code in a compute service and interacting with services and APIs to get the job done.

1.1 How we got to where we are

If you look at systems powering most of today's web-enabled software, you'll see back end servers performing various forms of computation and client-side front ends providing an interface for users to operate via their browser, mobile, or desktop device.

In a typical web application, the server accepts HTTP requests from the front end and processes requests. Data might travel through numerous application layers before being saved to a database. The back end, finally, generates a response—it could be in the form of JSON or fully rendered markup—which is sent back to the client (figure 1.1). Naturally, most systems are more complex once elements such as load balancing, transactions, clustering, caching, messaging, and data redundancy are taken into account. Most of this software requires servers running in data centers or in the cloud that need to be managed, maintained, patched, and backed up.

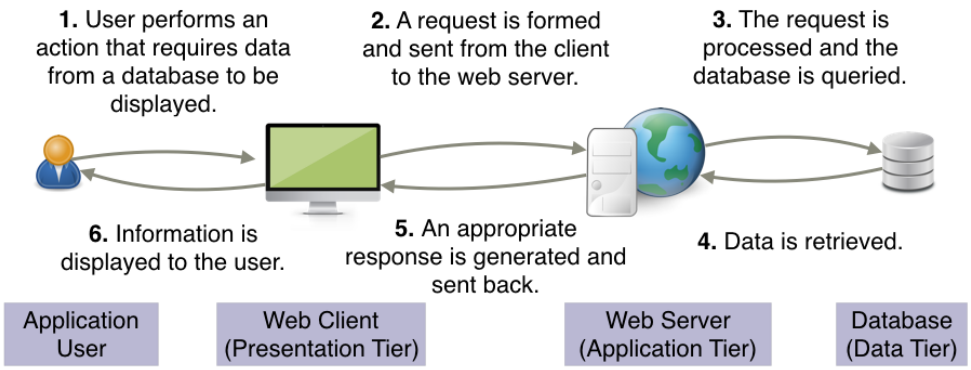


Figure 1.1 This is a basic request-response (client-server) message exchange pattern that most developers are familiar with. There's only one web server and one database in this figure. Most systems are much more complex.

Provisioning, managing, and patching of servers is a time-consuming task that often requires dedicated operations people. A non-trivial environment is hard to set up and operate effectively. Infrastructure and hardware are a necessary component of any IT system, but they're often also a distraction from what should be the core focus—solving the business problem.

Over the past few years, technologies such as platform as a service (PaaS) and containers have appeared as potential solutions to the headache of inconsistent infrastructure environments, conflicts, and server management overheard. PaaS is a form of cloud computing that provides a platform for users to run their software while hiding some of the underlying infrastructure. To make an effective use of PaaS, developers need to write software that targets the features and capabilities of the platform. Moving a legacy application designed to run on a standalone server to a PaaS service may require additional development effort because of the ephemeral nature of most PaaS implementations. Still, given a choice, many developers would understandably choose to use PaaS rather than more traditional, more manual solutions thanks to reduced maintenance and platform support requirements.

Containerization is a way of isolating an application with its own environment. It's a lightweight alternative to full-blown virtualization. Containers are isolated and lightweight but they need to be deployed to a server—whether in a public or private cloud or onsite. They're an excellent solution when dependencies are in play, but they have their own housekeeping challenges and complexities. They're not as easy as simply being able to run code directly in the cloud.

Finally, we make our way to Lambda, which is a compute service from Amazon Web Services. Lambda can execute code in a massively parallelized way in response to events. Lambda takes your code and runs it without any need to provision servers, install software, deploy containers, or worry about low-level detail. AWS takes care of provisioning and management of their Elastic Compute Cloud (EC2) servers that run

the actual code and provides a high-availability compute infrastructure—including capacity provisioning and automated scaling—that the developer doesn't need to think about. The words s*erverless architectures* refer to these new kinds of software architectures that do not rely on direct access to a server to work. By taking Lambda and making use of various powerful single-purpose APIs and web services, developers can build loosely coupled, scalable, and efficient architectures quickly. *Moving away from servers and infrastructure concerns, as well as allowing the developer to primarily focus on code is the ultimate goal behind serverless.*

1.1.1 Service-oriented architecture and microservices

Among a number of different system and application architectures, service-oriented architecture (SOA) has a lot of name recognition among software developers. It's an architecture that clearly conceptualized the idea that a system can be composed of many independent services. Much has been written about SOA but it remains controversial and misunderstood because developers often confuse design philosophy with specific implementation and attributes.

SOA doesn't dictate the use of any particular technology. Instead, it encourages an architectural approach in which developers create autonomous services that communicate via message passing and often have a schema or a contract that defines how messages are created or exchanged. Service reusability and autonomy, composability, granularity, and discoverability are all important principles associated with SOA.

Microservices and serverless architectures are spiritual descendants of service-oriented architecture. They retain many of the aforementioned principles and ideas while attempting to address the complexity of old-fashioned service-oriented architectures.

There has been a recent trend to implement systems using microservices architecture. Developers tend to think of microservices as small, standalone, fully independent services built around a particular business purpose or capability. A microservice may have an application tier with its own API and a database.

Ideally, microservices should be easy to replace, with each service written in an appropriate framework and language. The mere fact that microservices can be written in different general-purpose or domain-specific languages (DSL) is a draw card for many developers. Although benefits can be gained from using the right language or a specialized set of libraries for the job, it can often be a trap too. Having a mix of languages and frameworks can be hard to support and leads to confusion and difficulties down the road without a strict discipline.

Each microservice may maintain its state and store data, which adds to the complexity of the system. Consistency and coordination management can become an issue too, because state must often be synchronized across disparate services. Microservices can communicate indirectly via a message bus or directly by sending messages to one another.

It can be argued that serverless architecture embodies many principles from microservices too. After all, depending on how you design the system, every compute function could be considered its own standalone service. But you don't need to fully embrace the microservices mantra and develop every function or service around a particular business purpose, maintain its state, and so on.

Serverless architectures give you the freedom to apply as few or as many microservices principles as you would like without forcing you down a single path. This book shows examples of architectures where parts of a monolithic system are re-implemented as serverless architecture without applying all of the microservices tenets. It's then up to you to decide how much farther to take your architecture based on your requirements and preference.

1.1.2 Software design

Software design has evolved from the days of code running on a mainframe to multi-tier systems of today where the presentation, data, and application/logic tiers feature prominently in many designs. Within each tier there may be multiple logical layers that deal with particular aspects of functionality or domain. There are also cross-cutting components, such as logging or exception-handling systems, that can span numerous layers. The preference for layering is understandable. Layering allows developers to decouple concerns and have more maintainable applications.

But the converse can also be true. Having too many layers can lead to inefficiencies. A small change can often cascade and cause the developer to modify every layer throughout the system, costing considerable time and energy in implementation and testing. The more layers there are, the more complex and unwieldy the system might become over time. Figure 1.2 shows an example of a tiered architecture with multiple layers.

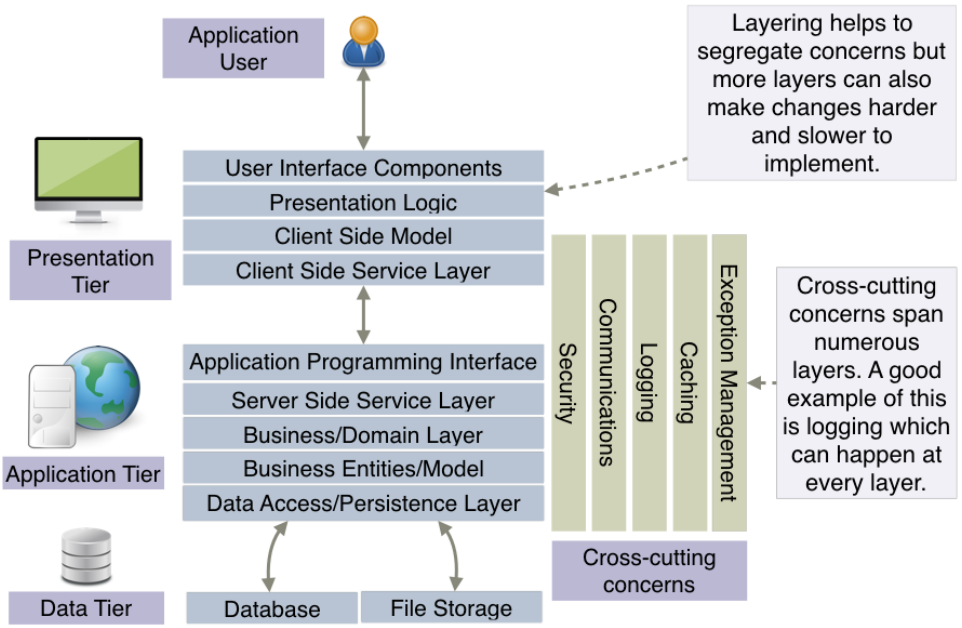


Figure 1.2 A typical three-tier application is usually made up of presentation, application, and data tiers. A tier may have multiple layers with specific responsibilities. A developer can choose how layers will interact with each other. This can be strictly top-down or in a loose way, where layers can bypass their immediate neighbors to talk to other layers.

Serverless architectures can help with the problem of layering and having to update too many things. There's room for developers to remove or minimize layering by breaking the system into functions and allowing the front end to securely communicate with services and even the database directly, as shown in figure 1.3. All of this can be done in an organized way to prevent spaghetti implementations and dependency nightmares by clearly defining service boundaries, allowing Lambda functions to be autonomous, and planning how functions and services will interact.

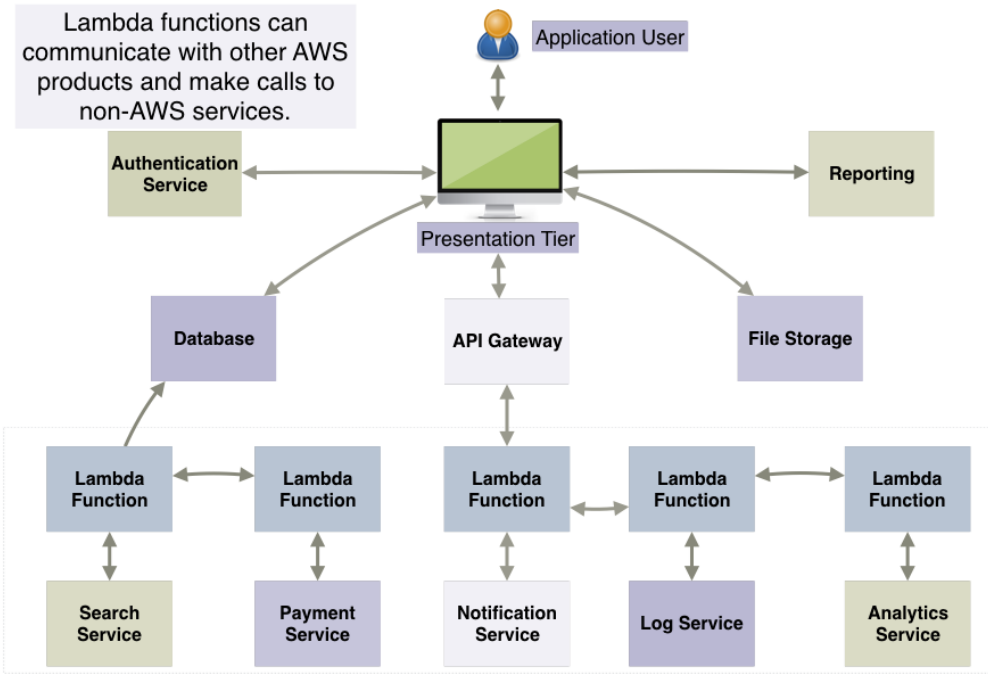


Figure 1.3 In a serverless architecture there's no single traditional back end. The front end of the application communicates directly with services, the database, or compute functions via an API gateway. Some services, however, must be hidden behind compute service functions, where additional security measures and validation can take place.

A serverless approach doesn't solve all problems, nor does it remove the underlying intricacies of the system. But when implemented correctly it can provide opportunities to reduce, organize, and manage complexity. A well-planned serverless architecture can make future changes easier, which is an important factor for any long-term application. The next section and later chapters discuss the organization and orchestration of services in more detail.

Tiers vs. layers

There is confusion among some developers about the difference between layers and tiers. A *tier* is a module boundary that exists to provide isolation between major components of a system. A presentation tier that's visible to the user is separate from the application tier, which encompasses business logic. In turn, the data tier is another separate system that can manage, persist, and provide access to data. Components grouped in a tier can physically reside on different infrastructure.

Layers are logical slices that carry out specific responsibilities in an application. Each tier can have multiple layers within it responsible for different elements of functionality such as domain services.

1.2 Principles of serverless architectures

Serverless architecture has five principles that describe how an ideal serverless system should be built. Use these principles to help guide your decisions when you create serverless architecture.

1. Use a compute service to execute code on demand (no servers).
2. Write single-purpose stateless functions.
3. Design push-based, event-driven pipelines.
4. Create thicker, more powerful front ends.
5. Embrace third-party services.

Let's look at each of these principles in more detail.

1.2.1 Use a compute service to execute code on demand

Serverless architectures are a natural extension of ideas raised in SOA. In serverless architecture all custom code is written and executed as isolated, independent, and often granular functions that are run in a stateless compute service such as AWS Lambda. Developers can write functions to carry out almost any common task, such as reading and writing to a data source, calling out to other functions, and performing a calculation. In more complex cases, developers can set up more elaborate pipelines and orchestrate invocations of multiple functions. There might be scenarios where a server is still needed to do something. These cases, however, may be far and few between, and as a developer you should avoid running and interacting with a server if possible.

So, what is Lambda exactly?

Lambda is a compute service that executes code written in JavaScript (Node.js), Python, or Java on AWS infrastructure. Source code is deployed to an isolated container that has its own allocation of memory, disk space, and CPU. The combination of your code, configuration, and dependencies is typically referred to as a *Lambda function*. The Lambda runtime can invoke a function multiple times in parallel. Lambda supports push and pull event models of operation and integrates with a large number of AWS services. Functions can be invoked by an HTTP request through the API Gateway or run on a scheduler. Chapter 6 covers Lambda in more detail, including its event model, methods of invocation, and best practice with regard to design. Note that Lambda is not the only game in town. Microsoft Azure Functions, IBM Bluemix OpenWhisk, and Google Cloud Functions are other compute services you might want to look at.

1.2.2 Write single-purpose stateless functions

As a software engineer, you should try to design your functions with the single responsibility principle (SRP) in mind. A function that does just one thing is more testable and robust and leads to fewer bugs and unexpected side effects. By composing and combining functions and services in a loose orchestration, you can build complex back end systems that are still understandable and easy to manage. A granular function with a well-defined interface is also more likely to be reused within a serverless architecture.

Code written for a compute service such as Lambda should be created in a *stateless* style. It must not assume that local resources or processes will survive beyond the immediate session. Statelessness is powerful because it allows the platform to quickly scale to handle an ever-changing number of incoming events or requests.

1.2.3 Design push-based, event-driven pipelines

Serverless architectures can be built to serve any purpose. Systems can be built serverless from scratch, or existing monolithic applications can be gradually re-engineered to take advantage of this architecture. The most flexible and powerful serverless designs are event driven. In chapter 3, for example, we'll build an event-driven, push-based pipeline to show how quickly we can put together a system to encode video to different bitrates and formats. We'll achieve this by connecting Amazon's Simple Storage Service (S3), Lambda, and Elastic Transcoder together (figure 1.4).

Building event-driven, push-based systems will often reduce cost and complexity (you won't need to run extra code to poll for changes) and potentially make the overall user experience smoother. It goes without saying that although event-driven, push-based models are a good goal, they might not be appropriate or achievable in all circumstances. Sometimes you'll have to implement a Lambda function that polls the event source or runs on a schedule. We'll cover different event models and work through examples in later chapters.

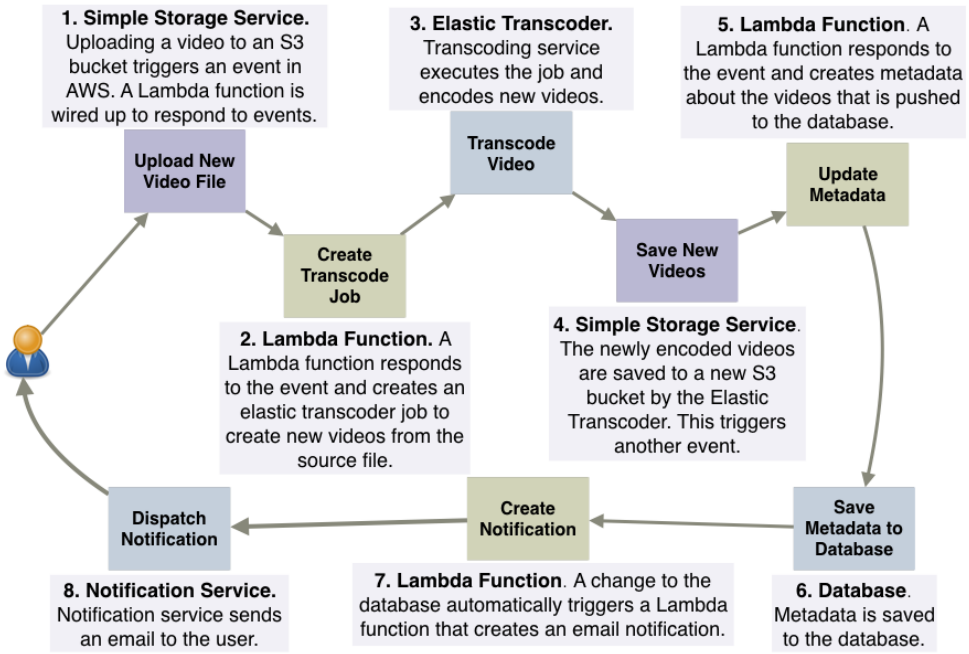


Figure 1.4 A push-based pipeline style of design works well with serverless architectures. In this example a user uploads a video, which is transcoded to a different format.

1.2.4 Create thicker, more powerful front ends

It's important to remember that custom code running in Lambda should be quick to execute. Functions that terminate sooner are cheaper, because Lambda pricing is based on the number of requests, the duration of execution, and the amount of memory allocated. Having less to do in Lambda is cheaper. Furthermore, having a richer front end that can invoke services can be conducive to a better user experience. Fewer hops between online resources and reduced latency will result in a better perception of performance and usability of the application.

Digitally signed tokens can allow front ends to communicate with disparate services, including databases directly. This is in contrast to traditional systems where all communication flows through via the back end server. Having the front end communicate with services helps to create systems that need far fewer hops to get to the required resource.

Not everything, however, can or should be done in the front end. There are secrets that cannot be trusted to the client device. Processing a credit card or sending emails to subscribers must be done only by a service that runs outside the end user's control. In this case, a compute service is required to coordinate action, validate data, and enforce security.

The other important point to consider is consistency. If the front end is responsible for writing to multiple services and fails midway through, it can leave the system in an inconsistent state. In this scenario, a Lambda function should be used because it can be designed to gracefully handle errors and retry failed operations. Atomicity and consistency are much easier to enforce and control in a Lambda function than in the front end.

1.2.5 Embrace third-party services

Third-party services are welcome to join the show if they can provide value and reduce custom code. Developers can leverage many services these days from Auth0 for authentication to Stripe or Braintree for payment processing. As long as factors such as price, capability, and availability are considered, developers should try to adopt third-party services. It's far more useful for developers to spend time solving a problem unique to their domain than re-creating functionality already implemented by someone else. Don't build for the sake of building if viable third-party services and APIs are available. Stand on the shoulders of giants to reach new heights. Appendix A has a short list of Amazon Web Services and non-Amazon Web Services we have found useful. We'll look at most of those services in more detail as we move through the book.

1.3 Transitioning from a server to services

One advantage of the serverless approach is that existing applications can be gradually converted to serverless architecture. If a developer is faced with a monolithic code base, they can gradually tease it apart and create Lambda functions that the application can communicate with.

The best approach is to initially create a prototype to test developer assumptions about how the system would function if it were going to be partly or fully serverless. Legacy systems tend to have interesting constraints that require creative solutions; as with any architectural refactor at a large scale, there will be compromises. The system may end up being a hybrid—see figure 1.5—but it may be better to have some of its aspects run out of Lambda and use third-party services than remain with a legacy architecture that no longer scales or requires expensive infrastructure to run.

The transition from a legacy, server-based application to a scalable serverless architecture may take time to get right. It needs to be approached carefully and slowly, and developers need to have a good test plan and a great DevOps strategy in place before they begin.

1.4 Serverless pros and cons

There are advantages to implementing a system as fully or partially serverless, including reduced cost and accelerated time to market. But we need to carefully consider the road to serverless architecture in the context of the application being created.

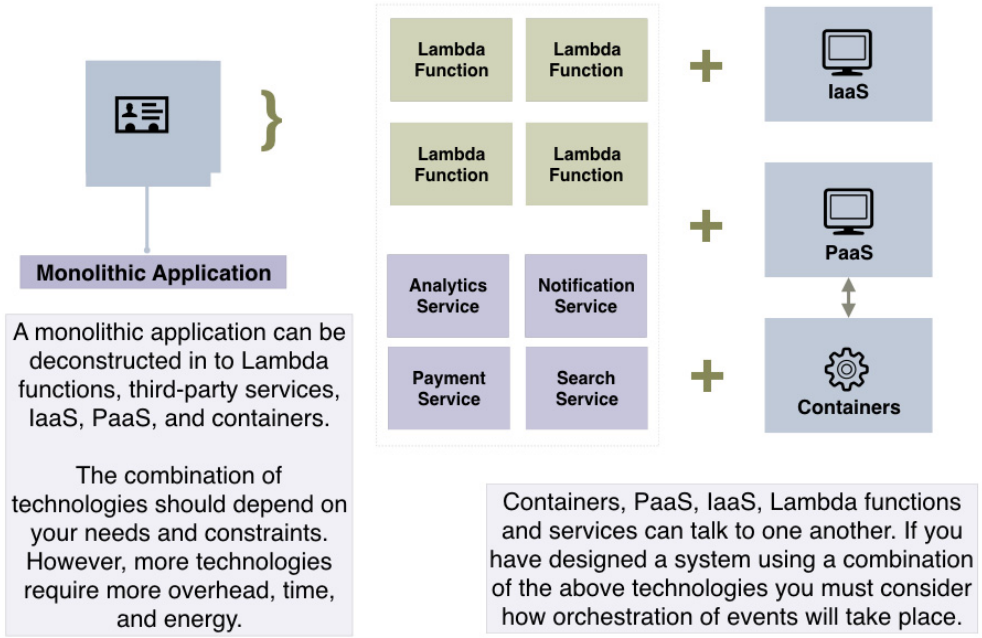


Figure 1.5 Serverless architecture is not an all-or-nothing proposition. If you currently have a monolithic application running on servers, you can begin to gradually extract components and run them in isolated services or compute functions. You can decouple a monolithic application into an assortment of infrastructure as a service (IaaS), PaaS, containers, Lambda functions, and third-party services if it helps.

1.4.1 Decision drivers

Serverless is not a silver bullet in all circumstances. It may not be appropriate for applications that have special performance requirements or service-level agreements (SLA). Vendor lock-in can be an issue for enterprise and government clients, and decentralization of services can be a challenge.

NOT FOR EVERYONE

Lambda runs in a public cloud, so mission-critical applications shouldn't necessarily be based on it. A banking system that performs high-volume transactions or a patient life-support system requires a higher level of performance and reliability than a public cloud system can provide. It's possible that organizations could employ dedicated hardware or run private or hybrid clouds with their own compute services that might meet serviceability and reliability requirements. In that case, these architectures could be adopted.

SERVICE LEVELS AND CUSTOMIZATION

AWS has a an SLA for some services but not for others, so that may be a factor in your decision. Lambda runs on top of EC2, which has an SLA of 99.95%. S3 has an SLA of

99.9%. Other third-party services may have different SLAs or not have an SLA at all. For most systems, the reliability offered by AWS is sufficient, but some enterprise use cases may require additional guarantees. With Lambda, the efficiencies gained from having Amazon look after the platform and scale functions to handle thousands of requests per second come at the expense of being able to customize the operating system or tweak the underlying instance.

VENDOR LOCK-IN

Vendor lock-in is another issue. If a developer decides to leverage third-party APIs and services, including AWS, there's a reasonable chance that architecture could become strongly coupled to the platform being used. The implication of vendor lock-in and the risk of using third-party services—including company viability, data sovereignty and privacy, cost, support, documentation, and available feature set—need to be thoroughly considered.

DECENTRALIZATION

Moving from a monolithic approach to a more decentralized serverless approach doesn't automatically reduce the complexity of the underlying system either. The distributed nature of the solution can introduce its own challenges because of the need to make remote rather than in-process calls and the need to handle failures and latency across a network.

1.4.2 When to use serverless

Serverless architecture allows developers to focus on software design and code rather than infrastructure. Scalability and high-availability are easier to achieve, and the pricing is often fairer because you pay only for what you use. Importantly with serverless you have a potential to reduce some of the complexity of the system by minimizing the number of layers and amount of code you need.

NO MORE SERVERS

Tasks such as server configuration and management, patching, and maintenance are taken care of by the vendor, which saves time and money. Amazon looks after the health of its fleet of servers that power Lambda. If you don't have specific requirements to manage or modify compute resources, then having Amazon or another vendor look after them is a great solution. You're responsible only for your own code, leaving operational and administrative tasks to a different set of capable hands.

MANY USES

The statelessness and scalability of compute can be used to solve problems that benefit from parallel processing. Back ends for CRUD applications, e-commerce, back-office systems, complex web apps, and all kinds of mobile and desktop software can be built very quickly using serverless architectures. Tasks that used to take weeks can be done in days or hours as long as the right combination of technologies is chosen. A serverless approach can work exceptionally well for startups that want to innovate and move quickly.

LOW COST

The traditional server-based architecture requires servers that don't necessarily run at full capacity all of the time. Scaling, even with automated systems, involves a new server, which is often wasted until there's a temporary upsurge in traffic or new data. Serverless systems are much more granular with regard to scaling and are cost effective, especially when peak loads are uneven or unexpected. With Lambda you only pay for what you use (chapter 4 shows how to calculate cost for Lambda and the API Gateway).

LESS CODE

We mentioned at the start of the chapter that serverless architecture provides an opportunity to reduce some of the complexity and code in comparison to more traditional systems. There's less need to have a multilayered back end system especially if you allow the front end to do more work and talk to services (and the database) directly.

SCALABLE AND FLEXIBLE

As a developer you don't need to use serverless architecture to replace your entire back end if you don't want to or are unable to do so. You can use Lambda to solve specific problems, especially if they stand to benefit from parallelization. It goes without saying that serverless systems can scale more easily than traditional systems. For example, consider the following solutions:

- ConnectWise, an IT services company, uses Lambda to process inbound logs, which has reduced their server maintenance needs from weeks to hours (http://amzn.to/1PFQ2hZ).
- Netflix uses Lambda to automate validation of backup completions and automate the encoding process of media files (http://amzn.to/1Qj5LQg).

You can use Lambda for extract, transform, and load (ETL) jobs, real-time file processing, and virtually anything else without having to touch your existing codebase. Just write a function and run it.

1.5 Summary

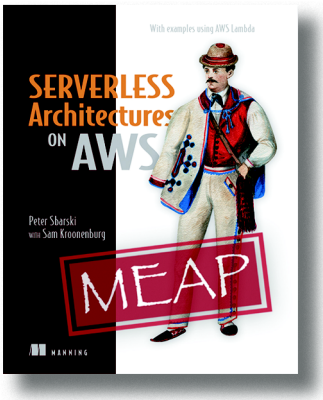
Cloud has been and continues to be a game changer for IT infrastructure and software development. Software developers need to think about the way they can maximize use from cloud platforms to gain a competitive advantage.

Serverless architectures are the latest advance for developers and organizations to think about, study, and adopt. This exciting new shift in architecture will grow quickly as developers embrace compute services such as AWS Lambda. Today serverless applications support thousands of users and carry out complex operations, including heavy-duty tasks such as video encoding and data processing. In many cases, serverless architectures can achieve a better outcome than traditional models and are cheaper and faster to implement.

There's also a need to reduce complexity and costs associated with running infrastructure and carrying out development of traditional software systems. The reduction in cost and time spent on infrastructure maintenance and the benefits of scalability

are good reasons for organizations and developers to consider serverless architectures. It's likely that the push for serverless back ends will accelerate over the coming years.

In this chapter you learned what serverless architecture is and saw how it compares to traditional architectures. We looked at core principles and considered some challenges associated with this architecture. In the next chapter, we'll dive straight into creating a serverless application to show how quickly something can be built using Lambda and AWS.



There's a shift underway toward serverless cloud architectures. With the release of serverless compute technologies, such as AWS Lambda, developers are now building entirely serverless platforms at scale. In these new architectures, traditional back-end servers are replaced with cloud functions acting as discrete single-purpose services. By composing and combining these serverless cloud functions together in a loose orchestration, and adopting useful third-party services, you can build powerful, yet easy to understand applications. Serverless architecture's about building rich, scalable, high-performing, and cost-effective systems without having to worry about traditional compute infrastructure, having more time to focus on code, and moving quickly.

Serverless Architectures on AWS teaches you how to build, secure and manage serverless architectures that can power the most demanding web and mobile apps. You'll get going quickly with this book's ready-made and real-world examples, code snippets, diagrams, and descriptions of architectures that can be readily applied. This book describes a traditional application and its back-end concerns, and then shows how to solve these same problems with a serverless approach. You'll begin with a high-level overview of what serverless is all about, start creating your own media transcoding system, and learn more about AWS. Next, you'll go in depth and learn about Lambda, API Gateway and other important serverless technologies. This section will teach you how to compose Lambda functions and discuss important considerations when it comes to building serverless systems. The third part of the book focuses on more advanced topics as your architecture grows. By the end, you'll be able to reason about serverless systems and be able to compose your own systems by applying these ideas and examples.

What's inside

- Creating a serverless back end
- Using Lambda and the API Gateway
- Connecting multiple services
- Authorization and authentication in a serverless environment
- Securely communicating with third-party services
- Interacting with a database from the front end
- Setting up continuous integration and deployment
- Building high-performance systems using messaging and eventing
- Using AWS to your advantage

This book is for all software developers interested in back-end technologies. Experience with JavaScript (node.js) and AWS is useful, but not required.

AWS Lambda in Action

AWS Lambda is an execution environment for small pieces of code called "functions" in the cloud. No matter how often the function is invoked, the cloud provider will scale the underlying infrastructure accordingly, and you only pay for what you use: compute time. If your function isn't invoked, you pay nothing. *AWS Lambda in Action* introduces you to this brave new world: Running Functions in the Cloud.

Chapter 1 from *AWS Lambda in Action* by Danilo Poccia

Running functions in the cloud

This chapter covers

- Understanding why functions can be the primitives of your application
- Getting an overview of AWS Lambda
- Using functions for the back end of your application
- Building event-driven applications with functions
- Calling functions from a client

In recent years, cloud computing has changed the way we think about and implement IT services, allowing companies of every size to build powerful and scalable applications that could disrupt the industries in which they operated. Think of how Dropbox changed the way we use digital storage and share files with each other, or how Spotify changed the way we buy and listen to music.

Those two companies started small, and needed the capacity to focus their time and resources on bringing their ideas to life quickly. In fact, one of the most important advantages of cloud computing is that it frees developers from spending their time on tasks that don't add real value to their work, such as managing and scaling the infrastructure, patching the operating system (OS), or maintaining the software stack used to run their code. Cloud computing lets them concentrate on the unique and important features they want to build.

You can use cloud computing to provide the *infrastructure* for your application, in the form of virtual servers, storage, network, load balancers, and so on. The infrastructure can be scaled automatically using specific configurations. But with this approach you still need to prepare a whole environment to execute the code you write. You install and prepare an operating system or a virtual environment; you choose and configure a programming framework; and finally, when the overall stack is ready, you can plug in our code. Even if you use a container-based approach in building the environment, with tools such as Docker, you're still in change of managing versioning and updates of the containers you use.

Sometimes you need infrastructure-level access because you want to view or manage low-level resources. But you can also use cloud computing services that abstract from the underlying infrastructure implementation, acting like a *platform* on top of which you deploy your own customizations. For example, you can have services that provide you with a database, and you only need to plug in your data (together with a data model) without having to manage the installation and availability of the database itself. Another example is services where you provide the code of your application, and a standard infrastructure to support the execution of your application is automatically implemented.

If that's true for a development environment, as soon as you get closer to production things become more complex and you may have to take care of the scalability and availability of the solution. And you must never forget to think about security—considering who can do what, and on which resources—during the course of the design and implementation of an application.

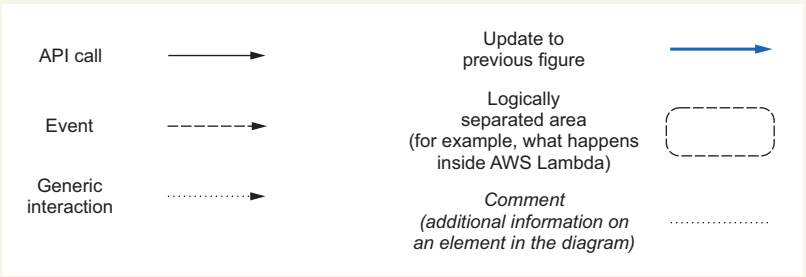
With the introduction of AWS Lambda, the abstraction layer is set higher, allowing developers to upload their code grouped in *functions*, and letting those functions be executed by the platform. In this way you don't have to manage the programming framework, the OS, or the availability and scalability. Each function has its own configuration that will help you use standard security features provided by Amazon Web Services (AWS) to define what a function can do and on which resources.

Those functions can be invoked directly or can *subscribe* to events generated by other *resources*. When you subscribe a function to a resource such as a file repository or a database, the function is automatically executed when something happens in that resource, depending on which kinds of events you've subscribed to. For example, when a file has been uploaded or a database item has been modified, an AWS Lambda function can react to those changes and do something with the new file or the updated data. If a picture has been uploaded, a function can create thumbnails to show the pictures on the screens with different resolutions. If a new record is written

in an operational database, a function can keep the data warehouse in sync. In this way you can design applications that are driven by events.

Book graphical conventions

This book uses the following graphical conventions to help present information more clearly.



Using multiple functions together, some of them called directly from a user device, such as a smartphone, and other functions subscribed to multiple repositories, such as a file share and a database, you can build a complete event-driven application. You can see a sample flow of a media-sharing application built in this way in figure 1.1. Users use a mobile app to upload pictures and share them with their friends.

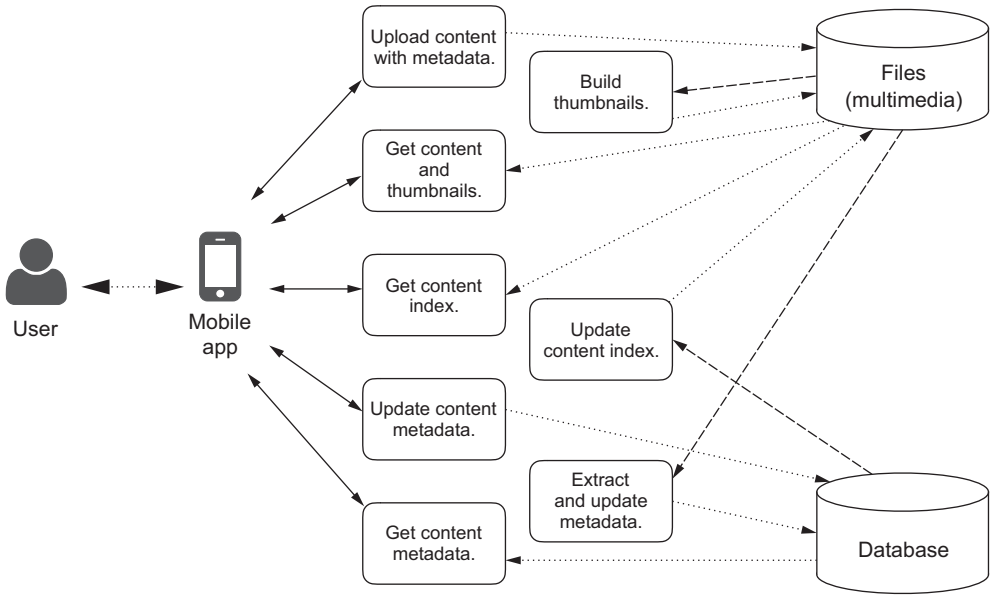


Figure 1.1 An event-driven, media-sharing application built using multiple AWS Lambda functions, some invoked directly by the mobile app. Other functions are subscribed to storage repositories such as a file share or a database.

NOTE Don't worry if you don't completely understand the flow of the application in figure 1.1. Reading this book, you'll first learn the architectural principles used in the design of event-driven applications, and then you'll implement this media-sharing application using AWS Lambda together with an authentication service to recognize users.

When using third-party software or a service not natively integrated with AWS Lambda, it's still easy to use that component in an event-driven architecture, adding the capacity to generate those events by using one of the AWS software development kits (SDKs), which are available for multiple platforms.

The event-driven approach not only simplifies the development of production environments, but also makes it easier to design and scale the *logic* of the application. For example, let's take a function that's subscribed to the upload of a file in a repository. Every time this function is invoked, it extracts information from the content of the file and writes this in a database table. You can think of this function as a logical connection between the file repository and the database table: every time any component of the application—including the client—uploads a file, the subscribed events are triggered and, in this case, the database is updated.

As you add more features, the logic of any application becomes more and more complex to manage. But in this case you created a *relationship* between the file repository and the database, and this connection works independently from the process that uploads the file. You'll see more advantages of this approach in this book, along with more practical examples.

If you're building a new application for either a small startup or a large enterprise, the simplifications introduced by using functions as the building blocks of your application will allow you to be more efficient in where to spend your time and faster in introducing new features to your users.

1.1 Introducing AWS Lambda

AWS Lambda is different from a traditional approach based on physical or virtual servers. You only need to give your logic, grouped in functions, and the service itself takes care of executing the functions, if and when required, by managing the software stack used by the runtime you chose, the availability of the platform, and the scalability of the infrastructure to sustain the throughput of the invocations.

Functions are executed in *containers*. Containers are a server virtualization method where the kernel of the OS implements multiple isolated environments. With AWS Lambda, physical servers still execute the code, but because you don't need to spend time managing them, it's common to define this kind of approach as *serverless*.

TIP For more details on the execution environment used by Lambda functions, please visit http://docs.aws.amazon.com/lambda/latest/dg/current-supported-versions.html.

When you create a new function with AWS Lambda, you choose a *function name*, create your code, and specify the configuration of the execution environment that will be used to run the function, including the following:

- The maximum *memory size* that can be used by the function
- A *timeout* after which the function is terminated, even if it hasn't completed
- A *role* that describes what the function can do, and on which resources, using AWS Identity and Access Management (IAM)

TIP When you choose the amount of memory you want for your function, you're allocated proportional CPU power. For example, choosing 256 MB of memory allocates approximately twice as much CPU power to your Lambda function as requesting 128 MB of memory and half as much CPU power as choosing 512 MB of memory.

AWS Lambda implements the execution of those functions with an efficient use of the underlying compute resources that allows for an interesting and innovative cost model. With AWS Lambda you pay for

- The number of invocations
- The hundreds of milliseconds of execution time of all invocations, depending on the memory given to the functions

The execution time costs grow linearly with the memory: if you double the memory and keep the execution time the same, you double that part of the cost. To enable you to get hands-on experience, a free tier allows you to use AWS Lambda without any cost. Each month there's no charge for

- The first one million invocations
- The first 400,000 seconds of execution time with 1 GB of memory

If you use less memory, you have more compute time at no cost; for example, with 128 MB of memory (1 GB divided by 8) you can have up to 3.2 million seconds of execution time (400,000 seconds multiplied by 8) per month. To give you a scale of the monthly free tier, 400,000 seconds corresponds to slightly more than 111 hours or 4.6 days, whereas 3.2 million seconds comes close to 889 hours or 37 days.

TIP You'll need an AWS account to follow the examples in this book. If you create a new AWS account, all the examples that I provide fall in the *Free Tier* and you'll have no costs to sustain. Please look here for more information on the AWS Free Tier and how to create a new AWS account: http://aws.amazon.com/free/.

Throughout the book we'll use JavaScript (Node.js, actually) and Python in the examples, but other runtimes are available. For example, you can use Java and other

languages running on top of the Java Virtual Machine (JVM), such as Scala or Clojure. For object-oriented languages such as Java, the function you want to expose is a method of an object.

To use platforms that aren't supported by AWS Lambda, such as C or PHP, it's possible to use one of the supported runtimes as a *wrapper* and bring together with the function a static binary or anything that can be executed in the OS container used by the function. For example, a statically linked program written in C can be embedded in the archive used to upload a function.

When you call a function with AWS Lambda, you provide an event and a context in the input:

- The *event* is the way to send input parameters for your function and is expressed using JSON syntax.
- The *context* is used by the service to describe the execution environment and how the event is received and processed.

Functions can be called *synchronously* and return a *result* (figure 1.2). I use the term "synchronous" to indicate this kind of invocation in the book, but in other sources, such as the AWS Lambda API Reference documentation or the AWS command-line interface (CLI), this is described as the `RequestResponse` invocation type.

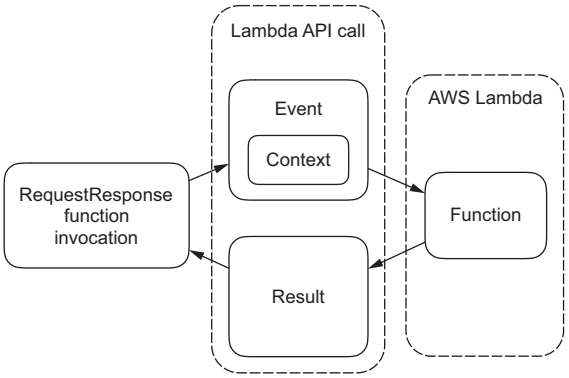


Figure 1.2 Calling an AWS Lambda function synchronously with the RequestResponse invocation type. Functions receive input as an event and a context and return a result.

For example, a simple synchronous function computing the sum of two numbers can be implemented in AWS Lambda using the JavaScript runtime as

```
exports.handler = (event, context, callback) => {
    var result = event.value1 + event.value2;
    callback(null, result);
};
```

The same can be done using the Python runtime:

```
def lambda_handler(event, context):
    result = event['value1'] + event['value2']
    return result
```

We'll dive deep into the syntax in the next chapter, but for now let's focus on what the functions are doing. Giving as input to those functions an event with the following JSON payload would give back a result of `30`:

```
{
  "value1": 10,
  "value2": 20
}
```

NOTE The values in JSON are given as numbers, without quotation marks; otherwise the + used in both the Node.js and Python functions would change the meaning, becoming a concatenation of two strings.

Functions can also be called *asynchronously*. In this case the call returns immediately and no result is given back, while the function is continuing its work (figure 1.3). I use the term "asynchronous" to indicate this kind of invocation in the book, but in other sources, such as the AWS Lambda API Reference documentation and the AWS CLI, this is described as the `Event` invocation type.

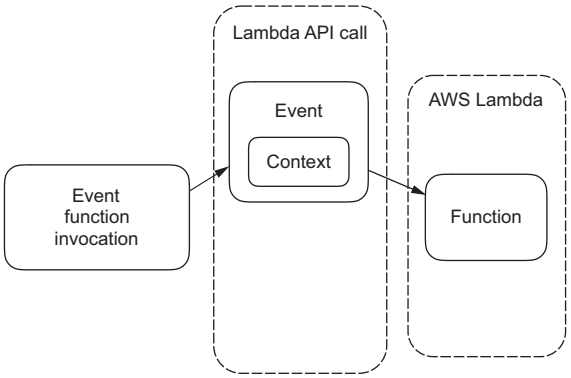


Figure 1.3 Calling an AWS Lambda function asynchronously with the Event invocation type. The invocation returns immediately while the function continues its work.

When a Lambda function terminates, no session information is retained by the AWS Lambda service. This kind of interaction with a server is usually defined as *stateless*. Considering this behavior, calling Lambda functions asynchronously (returning no value) is useful when they are accessing and modifying the status of other resources (such as files in a shared repository, records in a database, and so on) or calling other services (for example, to send an email or to send a push notification to a mobile device), as illustrated in figure 1.4.

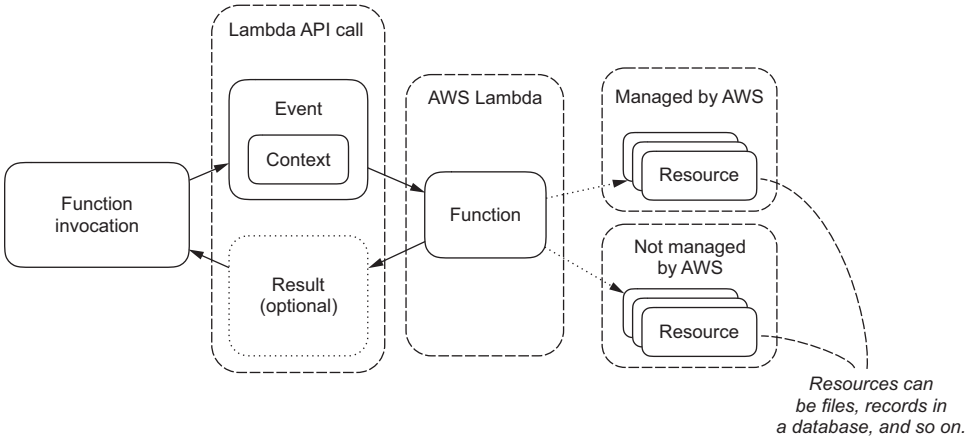


Figure 1.4 Functions can create, update, or delete other resources. Resources can also be other services that can do some actions, such as sending an email.

For example, it's possible to use the logging capabilities of AWS Lambda to implement a simple logging function (that you can call asynchronously) in Node.js:

```
exports.handler = function(event, context) {
    console.log(event.message);
    context.done();
};
```

In Python that's even easier because you can use a normal `print` to log the output:

```
def lambda_handler(event, context):
    print(event['message'])
    return
```

You can send input to the function as a JSON event to log a message:

```
{
  "message": "This message is being logged!"
}
```

In these two logging examples, we used the integration of AWS Lambda with Amazon CloudWatch Logs. Functions are executed without a default output device (in what is usually called a *headless environment*) and a default logging capability is given for each AWS Lambda runtime to ship the logs to CloudWatch. You can then use all the features provided by CloudWatch Logs, such as choosing the retention period or creating metrics from logged data. We'll give more examples and use cases regarding logging in part 4.

Asynchronous calls are useful when functions are *subscribed* to events generated by other resources, such as Amazon S3, an object store, or Amazon DynamoDB, a fully managed NoSQL database.

When you subscribe a function to events generated by other resources, the function is called asynchronously when the events you selected are generated, passing the events as input to the function (figure 1.5).

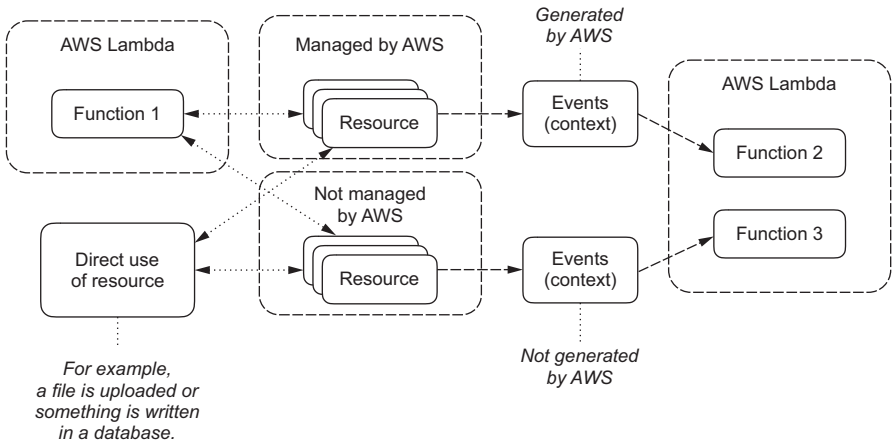


Figure 1.5 Functions can subscribe to events generated by direct use of resources, or by other functions interacting with resources. For resources not managed by AWS, you should find the best way to generate events to subscribe functions to those resources.

For example, if a user of a mobile application uploads a new high-resolution picture to a file store, a function can be triggered with the location of the new file in its input as part of the event. The function could then read the picture, build a small thumbnail to use in an index page, and write that back to the file store.

Now you know how AWS Lambda works at a high level, and that you can expose your code as functions and directly call those functions or subscribe them to events generated by other resources.

In the next section, you'll see how to use those functions in your applications.

1.2 Functions as your back end

Imagine you're a mobile developer and you're working on a new application. You can implement features in the mobile app running on the client device of the end user, but you'd probably keep part of the logic and status outside of the mobile app. For example:

- A mobile banking app wouldn't allow an end user to add money to their bank account without a good reason; only logic executed outside of the mobile device, involving the business systems of the bank, can decide if a transfer of money can be done or not.
- An online multiplayer game wouldn't allow a player to go to the next level without validating that the player has completed the current level.

This is a common pattern when developing client/server applications and the same applies to web applications. You need to keep part of the logic outside of the client (be it a web browser or a mobile device) for a few reasons:

- *Sharing*, because the information must be used (directly or indirectly) by multiple users of the application
- *Security*, because the data can be accessed or changed only if specific requirements are satisfied and the client cannot be trusted to check those requirements by itself
- *Access* to computing resources or storage capacity not available on a client device

We refer to this external logic required by a front end application as the *back end* of the application.

To implement this external logic, the normal approach is either to build a web application that can be called by the mobile app or to integrate it into an already existing web application rendering the content for a web browser. But instead of building and deploying a whole back end web application or extending the functionalities of your current back end, you can have your web page or your mobile application call one or more AWS Lambda functions that implement the logic you need. Those functions become your *serverless back end*.

Security is one of the reasons why you implement back end logic for an application, and you must always check the authentication and authorization of end users accessing your back end. AWS Lambda uses the standard security framework provided by AWS to control what a function can do, and on which resources. For example, a function can read from only a specific path of a file share, and write in a certain database table. This framework is based on AWS Identity and Access Management policies and roles. In this way, taking care of the security required to execute the code is simpler and becomes part of the development process itself. You can tailor security permissions specifically for each function, making it much easier to implement a least-privilege approach for each module (function, in this case) of your application.

DEFINITION By *least privilege*, I mean a security practice in which you always use the least privilege you need to perform an action in your application. For example, if you have a part of your application that's reading the user profiles from a central repository to publish them on a web page, you don't need to have write access to the repository in that specific module; you only need to read the subset of information you need to publish. Every other permission on top of that is in excess of what's required and can amplify the effects of a possible attack—for example, allowing malicious users that discover a security breach in your application to do more harm.

1.3 A single back end for everything

We can use AWS Lambda functions to expose the back end logic of our applications. But is that enough, or do we need something different to cover all the possible use cases for a back end application? Do we still need to develop traditional web applications, beyond the functions provided by AWS?

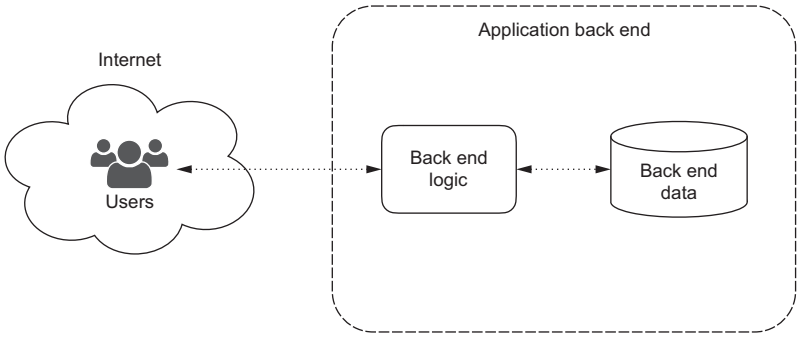


Figure 1.6 How users interact via the internet with the back end of an application. Note that the back end has some logic and some data.

Let's look at the overall flow and interactions of an application that can be used via a web browser or a mobile app (figure 1.6). Users interact with the back end via the internet. The back end has some logic and some data to manage.

The users of your application can use different devices, depending on what you decide to support. Supporting multiple ways to interact with your application, such as a web interface, a mobile app, and public application programming interfaces (APIs) that more advanced users can use to integrate third-party products with your application, is critical to success and is a common practice for new applications.

But if we look at the interfaces used by those different devices to communicate with the back end, we discover that they aren't always the same: a web browser expects more than the others, because both the content required by the user interface (dynamically generated HTML, CSS, JavaScript, multimedia files) and the application back end logic (exposed via APIs) are required (figure 1.7).

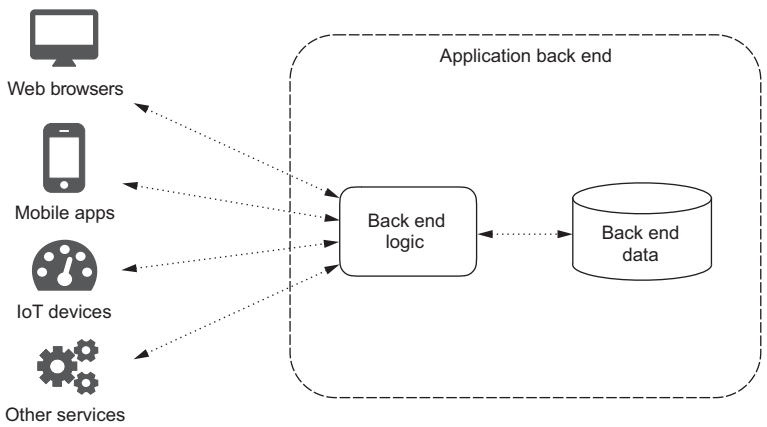


Figure 1.7 Different ways in which users can interact with the back end of an application. Users using a web browser receive different data than other front end clients.

If the mobile app of a specific service is developed after the web browser interface is already implemented, the back end application should be refactored to split API functionalities from web rendering—but that's usually not an easy task, depending on how the original application was developed. This sometimes causes developers to support two different back end platforms: one for web browsers serving web content and one for mobile apps, new devices (for example, wearable, home automation, and Internet of Things devices), and external services consuming their APIs. Even if the two back end platforms are well designed and share most of the functionalities (and hence the code), this wastes the developer's resources, because for each new feature they have to understand the impact on both platforms and run more tests to be sure those features are correctly implemented, while not adding value for their end users.

If we split the back end data between structured content that can go in one or more databases and unstructured content, such as files, we can simplify the overall architecture in a couple of steps:

1. Adding a (secure) web interface to the file repository so that it becomes a standalone resource that clients can directly access
2. Moving part of the logic into the web browser using a JavaScript client application and bringing it on par with the logic of the mobile app

Such a JavaScript client application, from an architectural point of view, behaves in the same way as a mobile app, in terms of functionality implemented, security, and (most importantly for our use case) the interactions with the back end (figure 1.8).

Looking at the back end logic, we now have a *single architecture for all clients* and the same interactions and data flows for all the consuming applications. We can abstract our back end from the actual implementation of the client and design it to serve a

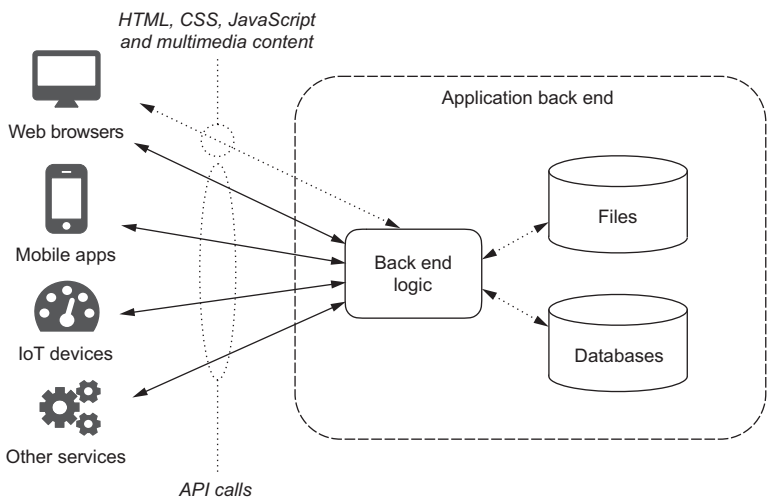


Figure 1.8 Using a JavaScript application running in the browser, back end architecture is simplified by serving only APIs to all clients.

generic *client application* using standard API calls that we define once and for all possible end users.

This is an important step because we've now *decoupled* the front end implementations, which could be different depending on the supported client devices, from the back end architecture (figure 1.9). Also, later you can add a new kind of client application (for example, an application running on wearable devices) without affecting the back end.

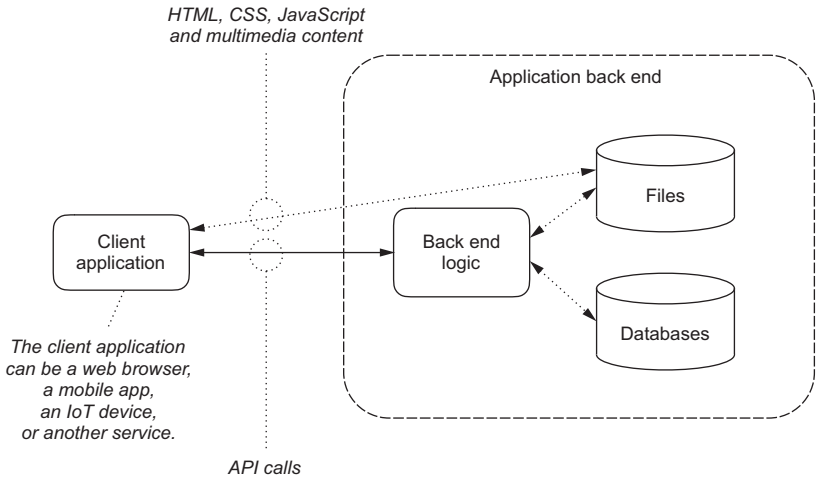


Figure 1.9 Think of your clients as a single client application consuming your APIs, which is possible when you decouple the implementation of the back end from the different user devices that interact with your application.

Looking again at the decoupled architecture, you can see that each of those API calls takes input parameters, does something in the back end, and returns a result. Does that remind you of something? Each API call is a *function exposed by the back end* that you can implement using AWS Lambda. Applying the same approach, all back end APIs can be implemented as functions managed by AWS Lambda.

In this way you have a *single serverless back end*, powered by AWS Lambda, that serves the same APIs to all clients of your application.

1.4 Event-driven applications

Up to now, we've used the functions provided by AWS Lambda directly, calling them as back end APIs from the client application. This is what's usually referred to as a *custom event* approach. But you could subscribe a function to receive events from another resource, for example if a file is uploaded to a repository or if a record in a database is updated.

Using subscriptions, you can change the internal behavior of the back end so that it can react not only to direct requests from client applications, but also to changes in the

resources that are used by the application. Instead of implementing a centralized workflow to support all the interactions among the resources, each interaction is described by the *relationship between the resources* involved. For example, if a file is added in a repository, a database table is updated with new information extracted from the file.

> **NOTE** This approach simplifies the design and the future evolution of the application, because we're inherently capitalizing on one of the advantages that microservices architectures bring: bottom-up *choreography* among software modules is much easier to manage than top-down *orchestration*.

With this approach, our back end becomes a distributed application, because it's not centrally managed and executed anymore, and we should apply best practices from distributed systems. For example, it's better to avoid synchronous transactions across multiple resources, which are difficult and slow to manage, and design each function to work independently (thanks to event subscriptions) with eventual consistency of data.

> **DEFINITION** By *eventual consistency*, I mean that we shouldn't expect the state of data to always be in sync across all resources used by the back end, but that the data will eventually converge over time to the last updated state.

Applications designed to react to internal and external events without a centralized workflow to coordinate processing on the resources are *event-driven applications*. Let's introduce this concept with a practical example.

Imagine you want to implement a media-sharing application, where the users can upload pictures from their client, a web browser or a mobile app, and share those pictures publicly with everyone or only with their friends.

To do that, you need two repositories:

- A file repository for the multimedia content (pictures)
- A database to handle user profiles (user table), friendships among the users (friendship table), and content metadata (content table).

You need to implement the following basic functionalities:

- Allow users to upload new multimedia content (pictures) with its own metadata. (By metadata, I mean: Is this content public or shared only among friends? Who uploaded the file? Where was the picture taken? At what time? Is there a caption?)
- Allow users to get specific content (pictures) shared by other users, but only if they have permission.
- Get an index of the content a specific user can see (all public content plus what has been shared with that user by their friends).
- Update content metadata. For example, a user can upload pictures only for their friends, and then change their mind and make a picture public for everyone to see.
- Get content metadata to be shown on the client together with the picture thumbnails; for example, adding the owner of the content, a date, a location, and a caption.

Of course, a real application needs more features (and more functions), but for the sake of simplicity we'll consider only the features listed here for now. You'll build a more complex (but still relatively simple) media-sharing application in chapter 8.

Because the content won't change too quickly, it's also effective to compute in advance (precompute) what each user can see in terms of content: end users will probably look at recent content often, and when they do, they want to see the result quickly. Using a precomputed *index* for the most recent content makes the rendering fast for users and makes the application use fewer computing resources in the back end. If users go back to older content outside the scope of the precomputed index, you can still compute that dynamically, but it happens less often and is easier to manage. The precomputed indexes must be updated each time the content (files or metadata) is updated and when the friendships between users change (because picture visibility is based on friendship).

You can see those features, and how they access repositories, implemented using one AWS Lambda function for each feature in figure 1.10.

In this way all interactions from the client application are covered, but you still miss basic back end functionalities here:

- What happens if a user uploads a new piece of content?
- What happens to the index if the user changes the metadata?
- You need to build thumbnails for the pictures to show them as a preview to end users.

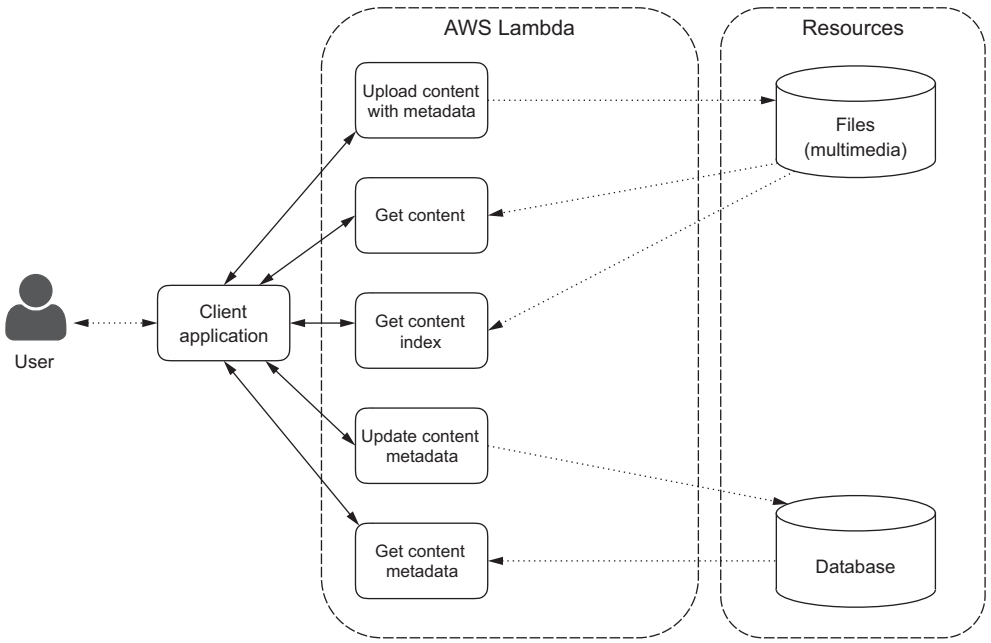


Figure 1.10 Features of a sample media-sharing application implemented as AWS Lambda functions, still missing basic back end functionalities

Those new back end features that you want to introduce are different from the previous ones, because they depend on what's happening in the back end repositories (files and database tables, in this case). You can implement those new features as additional functions that are subscribed to events coming from the repositories. For example:

- If a file (picture) is added or updated, you build the new thumbnail and add it back to the file repository.
- If a file (picture) is added or updated, you extract the new metadata and update the database (in the content table).
- Whenever the database is updated (user, friendship, or content table), you rebuild the dependent precomputed indexes, changing what a user can see.

Implementing those functionalities as AWS Lambda functions and subscribing those functions to the relevant events allows you to have an efficient architecture that drives updates when something relevant happens in the repositories, without enforcing a centralized workflow of activities that are required when data is changed by the end users. You can see a sample architecture implementing those new features as functions subscribed to events in figure 1.11.

Consider in our example the function subscribed to database events: that function is activated when the database is changed directly by end users (explicitly changing something in the metadata) or when an update is made by another function (because a new picture has been uploaded, bringing new metadata with it).

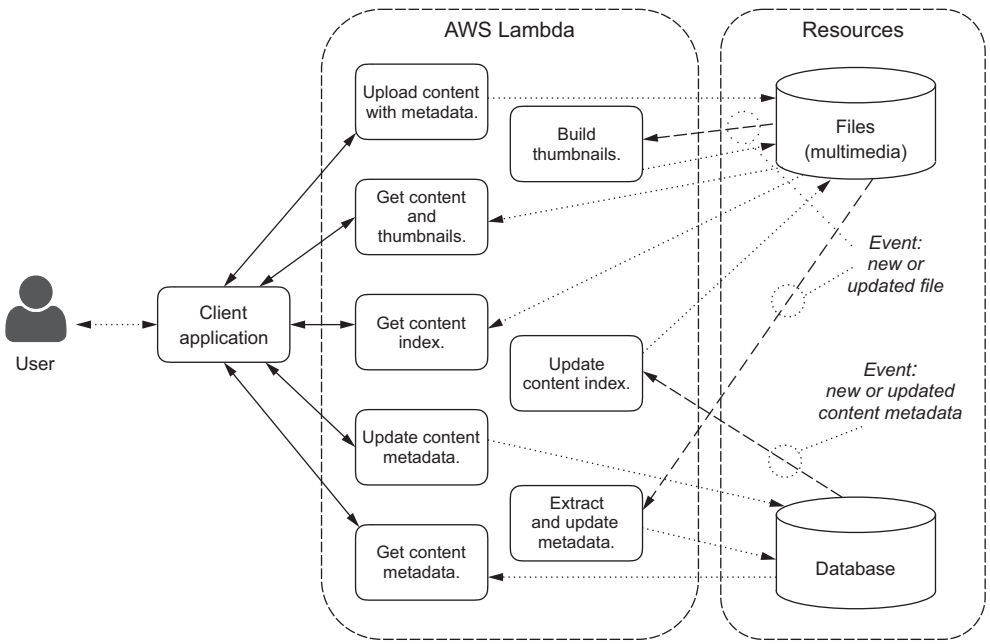


Figure 1.11 Sample media-sharing application with event-driven functions in the back end, subscribed to events from back end resources, such as file shares or databases

You don't need to manage the two use cases separately; they're both managed by the same subscription, a subscription that describes the relationship among the resources and the action you need to do when something changes.

You'll see when implementing this media-sharing application that some of the Lambda functions can be replaced by direct interactions to back end resources. For example, you can upload new or updated content (together with its own metadata) directly in a file share. Or update content metadata by directly writing to a database. The Lambda functions subscribed to those resources will implement the required back end logic.

This is a simplified but working example of a media-sharing application with an event-driven back end. Functions are automatically chained one after the other by the relationships we created by subscribing them to events. For example, if a picture is updated with new metadata (say, a new caption), a first function is invoked by the event generated in the file repository, updating the metadata in the database content table. This triggers a new event that invokes a second function to update the content index for all users who can see that content.

> **NOTE** In a way, the behavior I described is similar to a spreadsheet, where you update one cell and all the dependent cells (sums, average, more complex functions) are recomputed automatically. A spreadsheet is a good example of an event-driven application. This is a first step toward reactive programming, as you'll see later in the book.

Try to think of more features for our sample media-sharing application, such as creating, updating, and deleting a user; changing friendships (adding or removing a friend) and adding the required functions to the previous diagram to cover those aspects; subscribing (when it makes sense) the new functions to back end resources to have the flow of the application driven by events and avoid putting all the workflow logic in the functions themselves.

For example, suppose you have access to a mobile push notification service such as the Amazon Simple Notification Service (SNS). Think about the best way to use that in the back end to notify end users if new or updated content is available for them. What would you need to add, in terms of resources, events, and functions, to figure 1.11?

1.5 Calling functions from a client

In the previous discussion we didn't consider how, technically, the client application interacts with the AWS Lambda functions, assuming that a sort of direct invocation is possible.

As mentioned previously, each function can be invoked synchronously or asynchronously, and a specific AWS Lambda API exists to do that: the Invoke API (figure 1.12).

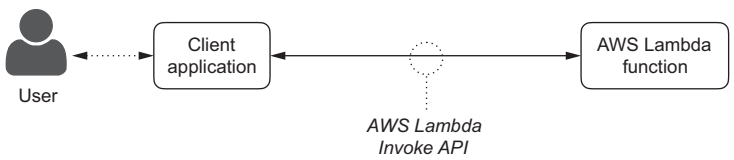


Figure 1.12 Calling AWS Lambda functions from a client application using the Invoke API

To call the Invoke API, AWS applies the standard security checks and requires that the client application has the right permissions to invoke the function. As per all other AWS APIs, you need AWS credentials to authenticate, and based on that authentication, AWS verifies whether those credentials have the right authorization to execute that API call (Invoke) on that specific resource (the function).

TIP We'll discuss the security model used by AWS Lambda in more detail in chapter 4. The most important thing to remember now is to *never put security credentials in a client application*, be that a mobile app or a JavaScript web application. If you put security credentials in something you deliver to end users, such as a mobile app or HTML or JavaScript code, an advanced user can find the credentials and compromise your application. In those cases, you need to use a different approach to authenticate a client application with the back end.

In the case of AWS Lambda, and all other AWS APIs, it's possible to use a specific service to manage authentication and authorization in an easy way: Amazon Cognito.

With Amazon Cognito, the client can authenticate using an external social or custom authentication (such as Facebook or Amazon) and get temporary AWS credentials to invoke the AWS Lambda functions the client is authorized to use (figure 1.13).

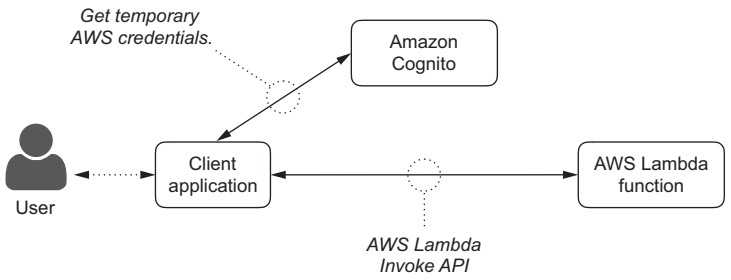


Figure 1.13 Using Amazon Cognito to authenticate and authorize invocation for AWS Lambda functions

NOTE Amazon Cognito provides a simplified interface to other AWS services, such as AWS Identity and Access Management (IAM) and AWS Security Token Service (STS). Figure 1.12 makes the flow easier to visualize, not including all details for the sake of simplicity.

Moving a step forward, it's possible to replace the direct use of the AWS Lambda Invoke API by clients with your own web APIs that you can build by mapping the access to AWS Lambda functions to more generic HTTP URLs and verbs.

For example, let's implement the web API for a bookstore. Users may need to list books, get more information for a specific book, and add, update, or delete a book. Using the Amazon API Gateway, you can map the access to a specific resource (the URL of the bookstore or a specific book) with an HTTP verb (`GET`, `POST`, `PUT`, `DELETE`, and so on) to the invocation of an AWS Lambda function. See table 1.1 for a sample configuration.

Table 1.1 A sample web API for a bookstore

Resource	+	HTTP verb	→	Method (function)
/books	+	GET	→	GetAllBooksByRange
/books	+	POST	→	CreateNewBook
/books/{id}	+	GET	→	GetBookById
/books/{id}	+	PUT	→	CreateOrUpdateBookById
/books/{id}	+	DELETE	→	DeleteBookById

Let's look at the example in table 1.1 in more detail:

- If you do an HTTP `GET` on the `/books` resource, you execute a Lambda function (`GetAllBooksByRange`) that will return a list of books, depending on a range you can optionally specify.
- If you do an HTTP `POST` on the same URL, you create a new book (using the `CreateNewBook` function) and get the `ID` of the book as the result.
- With an HTTP `GET` on `/books/ID`, you execute a function (`GetBookById`) that will give you a description (a representation, according to the REST architecture style) of the book with that specific `ID`.
- And so on for the other examples in the table.

NOTE You don't need to have a different Lambda function for every resource and HTTP verb (method) combination. You can send the resource and the method as part of the input parameters of a single function that can then process it to understand if it has been triggered by a `GET` or a `POST`. The choice between having more and smaller functions, or fewer and bigger ones, depends on your programming habits.

But the Amazon API Gateway adds more value than that, such as caching results to reduce load on the back end, throttling to avoid overloading the back end in peak moments, managing developer keys, generating the SDKs for the web API you design for multiple platforms, and other features that we'll start to see in chapter 2.

What's important is that by using the Amazon API Gateway we're *decoupling* the client from directly using AWS Lambda, exposing a clean web API that can be consumed by external services that should have no knowledge of AWS. However, even with the web API exposed by the Amazon API Gateway, we can optionally use AWS credentials (and hence Amazon Cognito) to manage authentication and authorization for the clients (figure 1.14).

With the Amazon API Gateway, we can also give public access to some of our web APIs. By *public access* I mean that no credentials are required to access those web APIs. Because one of the possible HTTP verbs that we can use in configuring an API is `GET`,

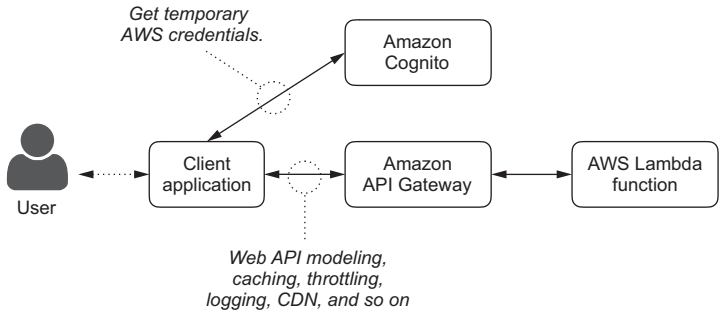


Figure 1.14 Using the Amazon API Gateway to access functions via web APIs

and `GET` is the default that is used when you type a URL in a web browser, we can use this configuration to create public websites whose URLs are dynamically served by AWS Lambda functions (figure 1.15).

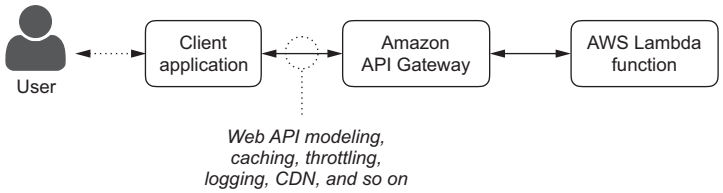


Figure 1.15 Using the Amazon API Gateway to give public access to an API and create public websites backed by AWS Lambda

In fact, the web API exposed publicly via the HTTP `GET` method can return any content type, including HTML content, such as a web page that can be seen in a browser.

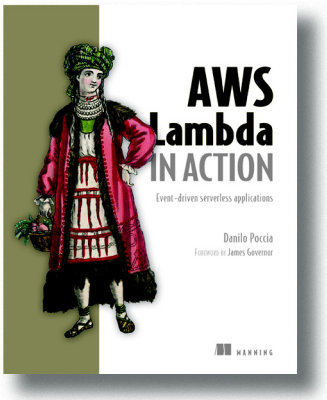
> **TIP** For an example of a joint use of AWS Lambda and the Amazon API Gateway to build dynamic websites, see the Serverless framework at http://www.serverless.com/.

1.6 Summary

In this first chapter, I introduced the core topics that will be seen in depth in the rest of the book:

- An overview of AWS Lambda functions.
- Using functions to implement the back end of an application.
- Having a single back end for different clients, such as web browsers and mobile apps.
- An overview of how event-driven applications work.
- Managing authentication and authorization from a client.
- Using Lambda functions from a client, directly or via the Amazon API Gateway.

Now let's put all this theory into practice and build our first functions.



AWS Lambda in Action is an example-driven tutorial that teaches you how to build applications that use an event-driven approach on the back-end. Starting with an overview of AWS Lambda, the book moves on to show you common examples and patterns that you can use to call Lambda functions from a web page or a mobile app. The second part of the book puts these smaller examples together to build larger applications. By the end, you'll be ready to create applications that take advantage of the high availability, security, performance, and scalability of AWS.

With AWS Lambda, you write your code and upload it to the AWS cloud. AWS Lambda responds to the events triggered by your application or your users, and automatically manages the underlying computer resources for you. Back-end tasks like analyzing a new document or processing requests from a mobile app are easy to implement. Your application is divided into small functions, leading naturally to a reactive architecture and the adoption of microservices.

What's inside

- Create a simple API
- Create an event-driven media-sharing application
- Secure access to your application in the cloud
- Use functions from different clients, like web pages or mobile apps
- Connect your application with external services

Requires basic knowledge of JavaScript. Some examples are also provided in Python. No AWS experience is assumed.

index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

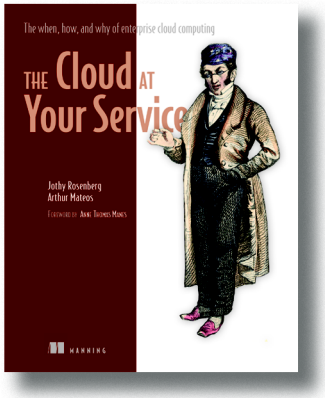
U

V

W

X

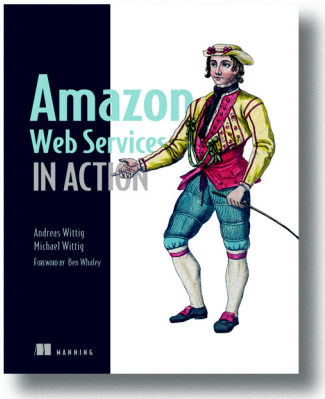
Y



The Cloud at Your Service

by Jothy Rosenberg and Arthur Mateos

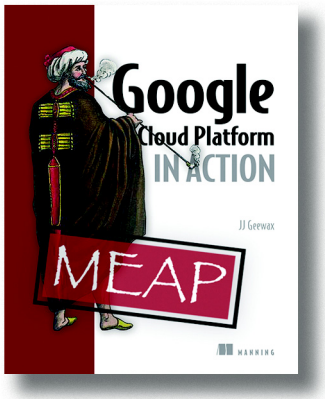
ISBN: 9781935192528
272 pages
$29.99
November 2010



Amazon Web Services in Action

by Michael Wittig and Andreas Wittig

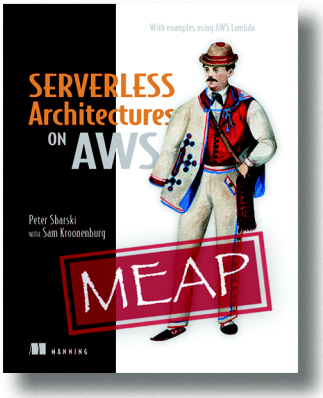
ISBN: 9781617292880
424 pages
$49.99
September 2015



Google Cloud Platform in Action

by JJ Geewax

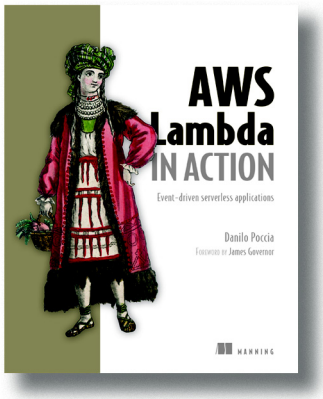
ISBN: 9781617293528
400 pages
$49.99
Spring 2017



Serverless Architectures on AWS

by Peter Sbarski with Sam Kroonenburg

ISBN: 9781617293825
425 pages
$44.99
Spring 2017



AWS Lambda in Action

by Danilo Poccia

ISBN: 9781617293719
384 pages
$49.99
November 2016