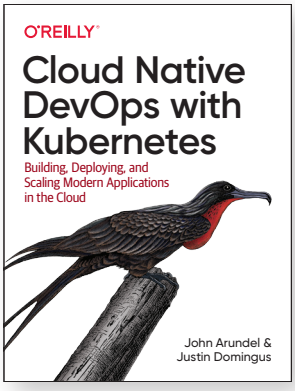
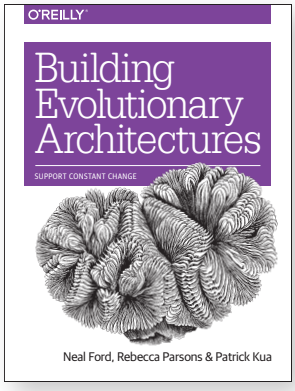
O'REILLY®

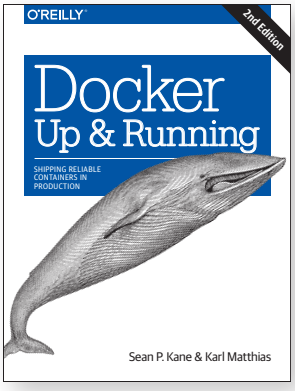
COMPILATION
FREE DOWNLOAD

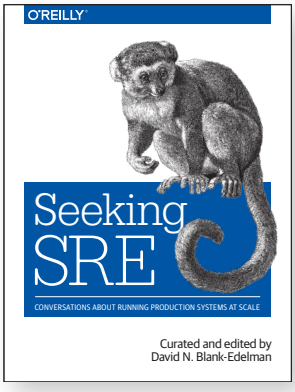
Next Architecture

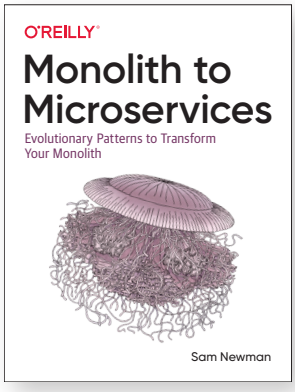
Learn the Architectures You Need to Better Plan, Code, Test & Deploy Applications

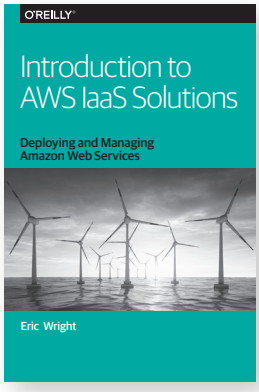








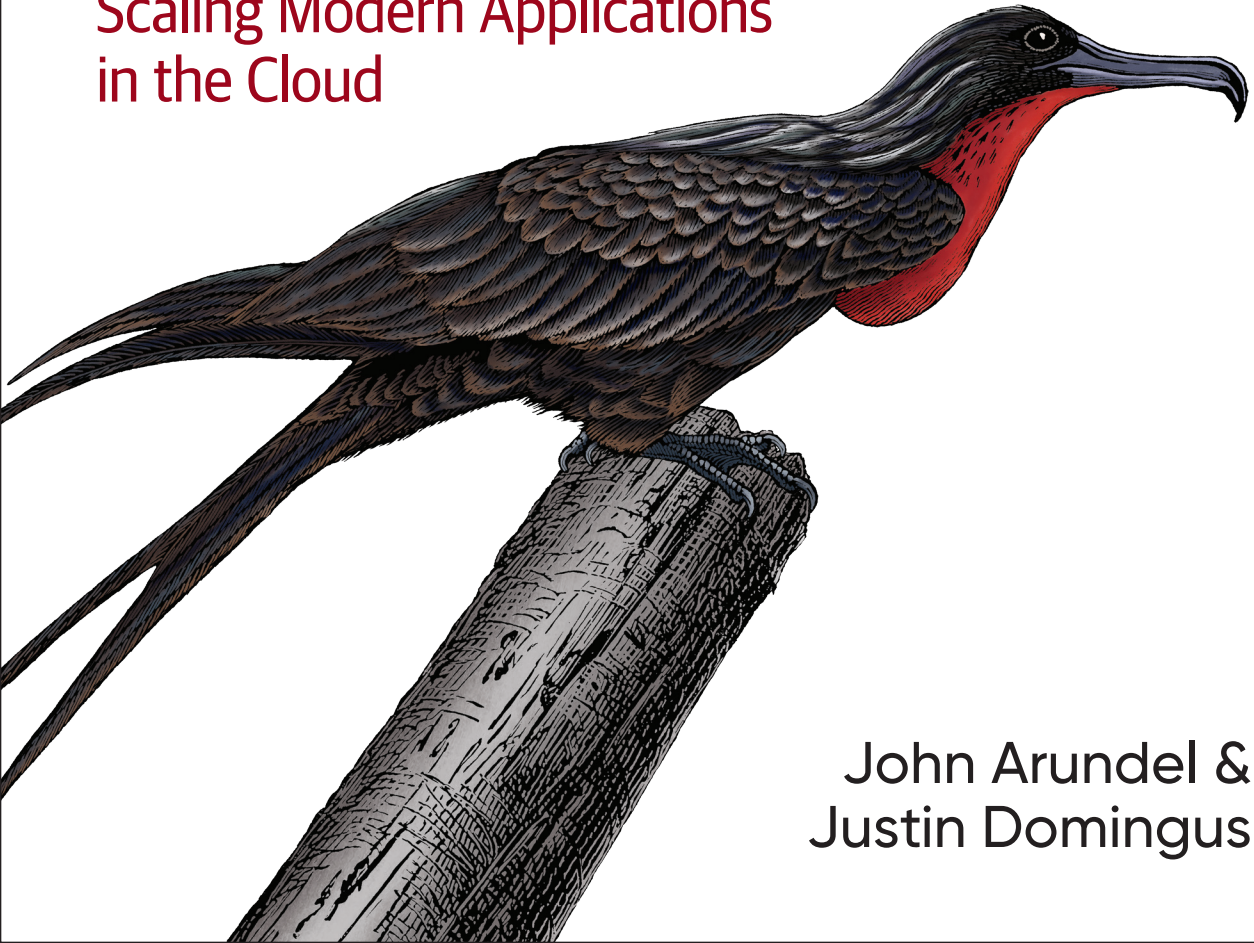




O'REILLY®

Cloud Native DevOps with Kubernetes

Building, Deploying, and Scaling Modern Applications in the Cloud



John Arundel &
Justin Domingus

CHAPTER 1
Revolution in the Cloud

> There was never a time when the world began, because it goes round and round like a circle, and there is no place on a circle where it begins.
>
> —Alan Watts

There's a revolution going on. Actually, three revolutions.

The first revolution is the creation of the cloud, and we'll explain what that is and why it's important. The second is the dawn of DevOps, and you'll find out what that involves and how it's changing operations. The third revolution is the coming of containers. Together, these three waves of change are creating a new software world: the *cloud native* world. The operating system for this world is called Kubernetes.

In this chapter, we'll briefly recount the history and significance of these revolutions, and explore how the changes are affecting the way we all deploy and operate software. We'll outline what *cloud native* means, and what changes you can expect to see in this new world if you work in software development, operations, deployment, engineering, networking, or security.

Thanks to the effects of these interlinked revolutions, we think the future of computing lies in cloud-based, containerized, distributed systems, dynamically managed by automation, on the Kubernetes platform (or something very like it). The art of developing and running these applications—*cloud native DevOps*—is what we'll explore in the rest of this book.

If you're already familiar with all of this background material, and you just want to start having fun with Kubernetes, feel free to skip ahead to Chapter 2. If not, settle down comfortably, with a cup of your favorite beverage, and we'll begin.

The Creation of the Cloud

In the beginning (well, the 1960s, anyway), computers filled rack after rack in vast, remote, air-conditioned data centers, and users would never see them or interact with them directly. Instead, developers submitted their jobs to the machine remotely and waited for the results. Many hundreds or thousands of users would all share the same computing infrastructure, and each would simply receive a bill for the amount of processor time or resources she used.

It wasn't cost-effective for each company or organization to buy and maintain its own computing hardware, so a business model emerged where users would share the computing power of remote machines, owned and run by a third party.

If that sounds like right now, instead of last century, that's no coincidence. The word *revolution* means "circular movement," and computing has, in a way, come back to where it began. While computers have gotten a lot more powerful over the years—today's Apple Watch is the equivalent of about three of the mainframe computers shown in Figure 1-1—shared, pay-per-use access to computing resources is a very old idea. Now we call it *the cloud*, and the revolution that began with timesharing mainframes has come full circle.

Figure 1-1. Early cloud computer: the IBM System/360 Model 91, at NASA's Goddard Space Flight Center

Buying Time

The central idea of the cloud is this: instead of buying a *computer*, you buy *compute*. That is, instead of sinking large amounts of capital into physical machinery, which is hard to scale, breaks down mechanically, and rapidly becomes obsolete, you simply buy time on someone else's computer, and let them take care of the scaling, maintenance, and upgrading. In the days of bare-metal machines—the "Iron Age", if you like—computing power was a capital expense. Now it's an operating expense, and that has made all the difference.

The cloud is not just about remote, rented computing power. It is also about distributed systems. You may buy raw compute resource (such as a Google Compute instance, or an AWS Lambda function) and use it to run your own software, but increasingly you also rent *cloud services*: essentially, the use of someone else's software. For example, if you use PagerDuty to monitor your systems and alert you when something is down, you're using a cloud service (sometimes called *software as a service*, or SaaS).

Infrastructure as a Service

When you use cloud infrastructure to run your own services, what you're buying is *infrastructure as a service* (IaaS). You don't have to expend capital to purchase it, you don't have to build it, and you don't have to upgrade it. It's just a commodity, like electricity or water. Cloud computing is a revolution in the relationship between businesses and their IT infrastructure.

Outsourcing the hardware is only part of the story; the cloud also allows you to outsource the *software* that you don't write: operating systems, databases, clustering, replication, networking, monitoring, high availability, queue and stream processing, and all the myriad layers of software and configuration that span the gap between your code and the CPU. Managed services can take care of almost all of this *undifferentiated heavy lifting* for you (you'll find out more about the benefits of managed services in Chapter 3).

The revolution in the cloud has also triggered another revolution in the people who use it: the DevOps movement.

The Dawn of DevOps

Before DevOps, developing and operating software were essentially two separate jobs, performed by two different groups of people. *Developers* wrote software, and they passed it on to *operations* staff, who ran and maintained the software *in production* (that is to say, serving real users, instead of merely running under test conditions). Like computers that need their own floor of the building, this separation has its roots

in the middle of the last century. Software development was a very specialist job, and so was computer operation, and there was very little overlap between the two.

Indeed, the two departments had quite different goals and incentives, which often conflicted with each other (Figure 1-2). Developers tend to be focused on shipping new features quickly, while operations teams care about making services stable and reliable over the long term.

Figure 1-2. Separate teams can lead to conflicting incentives (photo by Dave Roth)

When the cloud came on the horizon, things changed. Distributed systems are complex, and the internet is very big. The technicalities of operating the system—recovering from failures, handling timeouts, smoothly upgrading versions—are not so easy to separate from the design, architecture, and implementation of the system.

Further, "the system" is no longer just your software: it comprises in-house software, cloud services, network resources, load balancers, monitoring, content distribution networks, firewalls, DNS, and so on. All these things are intimately interconnected and interdependent. The people who write the software have to understand how it relates to the rest of the system, and the people who operate the system have to understand how the software works—or fails.

The origins of the DevOps movement lie in attempts to bring these two groups together: to collaborate, to share understanding, to share responsibility for systems reliability and software correctness, and to improve the scalability of both the software systems and the teams of people who build them.

Nobody Understands DevOps

DevOps has occasionally been a controversial idea, both with people who insist it's nothing more than a modern label for existing good practice in software development, and with those who reject the need for greater collaboration between development and operations.

There is also widespread misunderstanding about what DevOps actually is: A job title? A team? A methodology? A skill set? The influential DevOps writer John Willis has identified four key pillars of DevOps, which he calls culture, automation, measurement, and sharing (CAMS). Another way to break it down is what Brian Dawson has called the DevOps trinity: people and culture, process and practice, and tools and technology.

Some people think that cloud and containers mean that we no longer need DevOps—a point of view sometimes called *NoOps*. The idea is that since all IT operations are outsourced to a cloud provider or another third-party service, businesses don't need full-time operations staff.

The NoOps fallacy is based on a misapprehension of what DevOps work actually involves:

> With DevOps, much of the traditional IT operations work happens before code reaches production. Every release includes monitoring, logging, and A/B testing. CI/CD pipelines automatically run unit tests, security scanners, and policy checks on every commit. Deployments are automatic. Controls, tasks, and non-functional requirements are now implemented before release instead of during the frenzy and aftermath of a critical outage.
>
> —Jordan Bach (AppDynamics)

The most important thing to understand about DevOps is that it is primarily an organizational, human issue, not a technical one. This accords with Jerry Weinberg's *Second Law of Consulting*:

> No matter how it looks at first, it's always a people problem.
>
> —Gerald M. Weinberg, *Secrets of Consulting*

The Business Advantage

From a business point of view, DevOps has been described as "improving the quality of your software by speeding up release cycles with cloud automation and practices, with the added benefit of software that actually stays up in production" (The Register).

Adopting DevOps requires a profound cultural transformation for businesses, which needs to start at the executive, strategic level, and propagate gradually to every part of

the organization. Speed, agility, collaboration, automation, and software quality are key goals of DevOps, and for many companies that means a major shift in mindset.

But DevOps works, and studies regularly suggest that companies that adopt DevOps principles release better software faster, react better and faster to failures and problems, are more agile in the marketplace, and dramatically improve the quality of their products:

> DevOps is not a fad; rather it is the way successful organizations are industrializing the delivery of quality software today and will be the new baseline tomorrow and for years to come.
>
> —Brian Dawson (Cloudbees), Computer Business Review

Infrastructure as Code

Once upon a time, developers dealt with software, while operations teams dealt with hardware and the operating systems that run on that hardware.

Now that hardware is in the cloud, everything, in a sense, is software. The DevOps movement brings software development skills to operations: tools and workflows for rapid, agile, collaborative building of complex systems. Inextricably entwined with DevOps is the notion of *infrastructure as code*.

Instead of physically racking and cabling computers and switches, cloud infrastructure can be automatically provisioned by software. Instead of manually deploying and upgrading hardware, operations engineers have become the people who write the software that automates the cloud.

The traffic isn't just one-way. Developers are learning from operations teams how to anticipate the failures and problems inherent in distributed, cloud-based systems, how to mitigate their consequences, and how to design software that degrades gracefully and fails safe.

Learning Together

Both development teams and operations teams are also learning how to work together. They're learning how to design and build systems, how to monitor and get feedback on systems in production, and how to use that information to improve the systems. Even more importantly, they're learning to improve the experience for their users, and to deliver better value for the business that funds them.

The massive scale of the cloud and the collaborative, code-centric nature of the DevOps movement have turned operations into a software problem. At the same time, they have also turned software into an operations problem, all of which raises these questions:

- How do you deploy and upgrade software across large, diverse networks of different server architectures and operating systems?
- How do you deploy to distributed environments, in a reliable and reproducible way, using largely standardized components?

Enter the third revolution: the container.

The Coming of Containers

To deploy a piece of software, you need not only the software itself, but its *dependencies*. That means libraries, interpreters, subpackages, compilers, extensions, and so on.

You also need its *configuration*. Settings, site-specific details, license keys, database passwords: everything that turns raw software into a usable service.

The State of the Art

Earlier attempts to solve this problem include using *configuration management* systems, such as Puppet or Ansible, which consist of code to install, run, configure, and update the shipping software.

Alternatively, some languages provide their own packaging mechanism, like Java's JAR files, or Python's eggs, or Ruby's gems. However, these are language-specific, and don't entirely solve the dependency problem: you still need a Java runtime installed before you can run a JAR file, for example.

Another solution is the *omnibus package*, which, as the name suggests, attempts to cram everything the application needs inside a single file. An omnibus package contains the software, its configuration, its dependent software components, *their* configuration, *their* dependencies, and so on. (For example, a Java omnibus package would contain the Java runtime as well as all the JAR files for the application.)

Some vendors have even gone a step further and included the entire computer system required to run it, as a *virtual machine image*, but these are large and unwieldy, time-consuming to build and maintain, fragile to operate, slow to download and deploy, and vastly inefficient in performance and resource footprint.

From an operations point of view, not only do you need to manage these various kinds of packages, but you also need to manage a fleet of servers to run them on.

Servers need to be provisioned, networked, deployed, configured, kept up to date with security patches, monitored, managed, and so on.

This all takes a significant amount of time, skill, and effort, just to provide a platform to run software on. Isn't there a better way?

Thinking Inside the Box

To solve these problems, the tech industry borrowed an idea from the shipping industry: the *container*. In the 1950s, a truck driver named Malcolm McLean proposed that, instead of laboriously unloading goods individually from the truck trailers that brought them to the ports and loading them onto ships, trucks themselves simply be loaded onto the ship—or rather, the truck bodies.

A truck trailer is essentially a big metal box on wheels. If you can separate the box—the container—from the wheels and chassis used to transport it, you have something that is very easy to lift, load, stack, and unload, and can go right onto a ship or another truck at the other end of the voyage (Figure 1-3).

McLean's container shipping firm, Sea-Land, became very successful by using this system to ship goods far more cheaply, and containers quickly caught on. Today, hundreds of millions of containers are shipped every year, carrying trillions of dollars worth of goods.

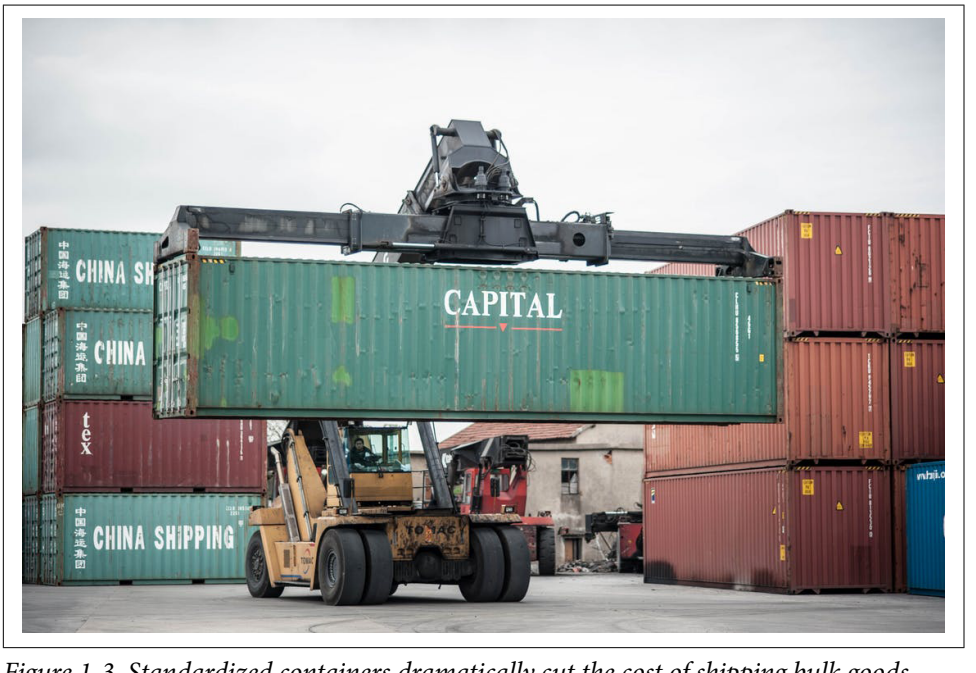


Figure 1-3. Standardized containers dramatically cut the cost of shipping bulk goods (photo by Pixabay, licensed under Creative Commons 2.0)

Putting Software in Containers

The software container is exactly the same idea: a standard packaging and distribution format that is generic and widespread, enabling greatly increased carrying

capacity, lower costs, economies of scale, and ease of handling. The container format contains everything the application needs to run, baked into an *image file* that can be executed by a *container runtime*.

How is this different from a virtual machine image? That, too, contains everything the application needs to run—but a lot more besides. A typical virtual machine image is around 1 GiB.[1] A well-designed container image, on the other hand, might be a hundred times smaller.

Because the virtual machine contains lots of unrelated programs, libraries, and things that the application will never use, most of its space is wasted. Transferring VM images across the network is far slower than optimized containers.

Even worse, virtual machines are *virtual*: the underlying physical CPU effectively implements an *emulated* CPU, which the virtual machine runs on. The virtualization layer has a dramatic, negative effect on performance: in tests, virtualized workloads run about 30% slower than the equivalent containers.

In comparison, containers run directly on the real CPU, with no virtualization overhead, just as ordinary binary executables do.

And because containers only hold the files they need, they're much smaller than VM images. They also use a clever technique of addressable filesystem *layers*, which can be shared and reused between containers.

For example, if you have two containers, each derived from the same Debian Linux base image, the base image only needs to be downloaded once, and each container can simply reference it.

The container runtime will assemble all the necessary layers and only download a layer if it's not already cached locally. This makes very efficient use of disk space and network bandwidth.

Plug and Play Applications

Not only is the container the unit of deployment and the unit of packaging; it is also the unit of *reuse* (the same container image can be used as a component of many different services), the unit of *scaling*, and the unit of *resource allocation* (a container can run anywhere sufficient resources are available for its own specific needs).

Developers no longer have to worry about maintaining different versions of the software to run on different Linux distributions, against different library and language

1 The *gibibyte* (GiB) is the International Electrotechnical Commission (IEC) unit of data, defined as 1,024 *mebibytes* (MiB). We'll use IEC units (GiB, MiB, KiB) throughout this book to avoid any ambiguity.

versions, and so on. The only thing the container depends on is the operating system kernel (Linux, for example).

Simply supply your application in a container image, and it will run on any platform that supports the standard container format and has a compatible kernel.

Kubernetes developers Brendan Burns and David Oppenheimer put it this way in their paper "Design Patterns for Container-based Distributed Systems":

> By being hermetically sealed, carrying their dependencies with them, and providing an atomic deployment signal ("succeeded"/"failed"), [containers] dramatically improve on the previous state of the art in deploying software in the datacenter or cloud. But containers have the potential to be much more than just a better deployment vehicle—we believe they are destined to become analogous to objects in object-oriented software systems, and as such will enable the development of distributed system design patterns.

Conducting the Container Orchestra

Operations teams, too, find their workload greatly simplified by containers. Instead of having to maintain a sprawling estate of machines of various kinds, architectures, and operating systems, all they have to do is run a *container orchestrator*: a piece of software designed to join together many different machines into a *cluster*: a kind of unified compute substrate, which appears to the user as a single very powerful computer on which containers can run.

The terms *orchestration* and *scheduling* are often used loosely as synonyms. Strictly speaking, though, *orchestration* in this context means coordinating and sequencing different activities in service of a common goal (like the musicians in an orchestra). *Scheduling* means managing the resources available and assigning workloads where they can most efficiently be run. (Not to be confused with scheduling in the sense of *scheduled jobs*, which execute at preset times.)

A third important activity is *cluster management*: joining multiple physical or virtual servers into a unified, reliable, fault-tolerant, apparently seamless group.

The term *container orchestrator* usually refers to a single service that takes care of scheduling, orchestration, and cluster management.

Containerization (using containers as your standard method of deploying and running software) offered obvious advantages, and a de facto standard container format has made possible all kinds of economies of scale. But one problem still stood in the way of the widespread adoption of containers: the lack of a standard container orchestration system.

As long as several different tools for scheduling and orchestrating containers competed in the marketplace, businesses were reluctant to place expensive bets on which technology to use. But all that was about to change.

Kubernetes

Google was running containers at scale for production workloads long before anyone else. Nearly all of Google's services run in containers: Gmail, Google Search, Google Maps, Google App Engine, and so on. Because no suitable container orchestration system existed at the time, Google was compelled to invent one.

From Borg to Kubernetes

To solve the problem of running a large number of services at global scale on millions of servers, Google developed a private, internal container orchestration system it called Borg.

Borg is essentially a centralized management system that allocates and schedules containers to run on a pool of servers. While very powerful, Borg is tightly coupled to Google's own internal and proprietary technologies, difficult to extend, and impossible to release to the public.

In 2014, Google founded an open source project named Kubernetes (from the Greek word κυβερνήτης, meaning "helmsman, pilot") that would develop a container orchestrator that everyone could use, based on the lessons learned from Borg and its successor, Omega.

Kubernetes's rise was meteoric. While other container orchestration systems existed before Kubernetes, they were commercial products tied to a vendor, and that was always a barrier to their widespread adoption. With the advent of a truly free and open source container orchestrator, adoption of both containers and Kubernetes grew at a phenomenal rate.

By late 2017, the orchestration wars were over, and Kubernetes had won. While other systems are still in use, from now on companies looking to move their infrastructure to containers only need to target one platform: Kubernetes.

What Makes Kubernetes So Valuable?

Kelsey Hightower, a staff developer advocate at Google, coauthor of *Kubernetes Up & Running* (O'Reilly), and all-around legend in the Kubernetes community, puts it this way:

> Kubernetes does the things that the very best system administrator would do: automation, failover, centralized logging, monitoring. It takes what we've learned in the DevOps community and makes it the default, out of the box.
>
> —Kelsey Hightower

Many of the traditional sysadmin tasks like upgrading servers, installing security patches, configuring networks, and running backups are less of a concern in the

cloud native world. Kubernetes can automate these things for you so that your team can concentrate on doing its core work.

Some of these features, like load balancing and autoscaling, are built into the Kubernetes core; others are provided by add-ons, extensions, and third-party tools that use the Kubernetes API. The Kubernetes ecosystem is large, and growing all the time.

Kubernetes makes deployment easy

Ops staff love Kubernetes for these reasons, but there are also some significant advantages for developers. Kubernetes greatly reduces the time and effort it takes to deploy. Zero-downtime deployments are common, because Kubernetes does rolling updates by default (starting containers with the new version, waiting until they become healthy, and then shutting down the old ones).

Kubernetes also provides facilities to help you implement continuous deployment practices such as *canary deployments*: gradually rolling out updates one server at a time to catch problems early (see "Canary Deployments" on page 242). Another common practice is *blue-green* deployments: spinning up a new version of the system in parallel, and switching traffic over to it once it's fully up and running (see "Blue/Green Deployments" on page 241).

Demand spikes will no longer take down your service, because Kubernetes supports autoscaling. For example, if CPU utilization by a container reaches a certain level, Kubernetes can keep adding new replicas of the container until the utilization falls below the threshold. When demand falls, Kubernetes will scale down the replicas again, freeing up cluster capacity to run other workloads.

Because Kubernetes has redundancy and failover built in, your application will be more reliable and resilient. Some managed services can even scale the Kubernetes cluster itself up and down in response to demand, so that you're never paying for a larger cluster than you need at any given moment (see "Autoscaling" on page 102).

The business will love Kubernetes too, because it cuts infrastructure costs and makes much better use of a given set of resources. Traditional servers, even cloud servers, are mostly idle most of the time. The excess capacity that you need to handle demand spikes is essentially wasted under normal conditions.

Kubernetes takes that wasted capacity and uses it to run workloads, so you can achieve much higher utilization of your machines—and you get scaling, load balancing, and failover for free too.

While some of these features, such as autoscaling, were available before Kubernetes, they were always tied to a particular cloud provider or service. Kubernetes is *provider-agnostic*: once you've defined the resources you use, you can run them on any Kubernetes cluster, regardless of the underlying cloud provider.

That doesn't mean that Kubernetes limits you to the lowest common denominator. Kubernetes maps your resources to the appropriate vendor-specific features: for example, a load-balanced Kubernetes service on Google Cloud will create a Google Cloud load balancer, on Amazon it will create an AWS load balancer. Kubernetes abstracts away the cloud-specific details, letting you focus on defining the behavior of your application.

Just as containers are a portable way of defining software, Kubernetes resources provide a portable definition of how that software should run.

Will Kubernetes Disappear?

Oddly enough, despite the current excitement around Kubernetes, we may not be talking much about it in years to come. Many things that once were new and revolutionary are now so much part of the fabric of computing that we don't really think about them: microprocessors, the mouse, the internet.

Kubernetes, too, is likely to disappear and become part of the plumbing. It's boring, in a good way: once you learn what you need to know to deploy your application to Kubernetes, you're more or less done.

The future of Kubernetes is likely to lie largely in the realm of managed services. Virtualization, which was once an exciting new technology, has now simply become a utility. Most people rent virtual machines from a cloud provider rather than run their own virtualization platform, such as vSphere or Hyper-V.

In the same way, we think Kubernetes will become so much a standard part of the plumbing that you just won't know it's there anymore.

Kubernetes Doesn't Do It All

Will the infrastructure of the future be entirely Kubernetes-based? Probably not. Firstly, some things just aren't a good fit for Kubernetes (databases, for example):

> Orchestrating software in containers involves spinning up new interchangeable instances without requiring coordination between them. But database replicas are not interchangeable; they each have a unique state, and deploying a database replica requires coordination with other nodes to ensure things like schema changes happen everywhere at the same time:
>
> —Sean Loiselle (Cockroach Labs)

While it's perfectly possible to run stateful workloads like databases in Kubernetes with enterprise-grade reliability, it requires a large investment of time and engineering that it may not make sense for your company to make (see "Run Less Software" on page 47). It's usually more cost-effective to use managed services instead.

Secondly, some things don't actually need Kubernetes, and can run on what are sometimes called *serverless* platforms, better named *functions as a service*, or *FaaS* platforms.

Cloud functions and funtainers

AWS Lambda, for example, is a FaaS platform that allows you to run code written in Go, Python, Java, Node.js, C#, and other languages, without you having to compile or deploy your application at all. Amazon does all that for you.

Because you're billed for the execution time in increments of 100 milliseconds, the FaaS model is perfect for computations that only run when you need them to, instead of paying for a cloud server, which runs all the time whether you're using it or not.

These *cloud functions* are more convenient than containers in some ways (though some FaaS platforms can run containers as well). But they are best suited to short, standalone jobs (AWS Lambda limits functions to fifteen minutes of run time, for example, and around 50 MiB of deployed files), especially those that integrate with existing cloud computation services, such as Microsoft Cognitive Services or the Google Cloud Vision API.

Why don't we like to refer to this model as *serverless*? Well, it isn't: it's just somebody else's server. The point is that you don't have to provision and maintain that server; the cloud provider takes care of it for you.

Not every workload is suitable for running on FaaS platforms, by any means, but it is still likely to be a key technology for cloud native applications in the future.

Nor are cloud functions restricted to public FaaS platforms such as Lambda or Azure Functions: if you already have a Kubernetes cluster and want to run FaaS applications on it, OpenFaaS and other open source projects make this possible. This hybrid of functions and containers is sometimes called *funtainers*, a name we find appealing.

A more sophisticated software delivery platform for Kubernetes that encompasses both containers and cloud functions, called Knative, is currently under active development (see "Knative" on page 238). This is a very promising project, which may mean that in the future the distinction between containers and functions may blur or disappear altogether.

Cloud Native

The term *cloud native* has become an increasingly popular shorthand way of talking about modern applications and services that take advantage of the cloud, containers, and orchestration, often based on open source software.

Indeed, the Cloud Native Computing Foundation (CNCF) was founded in 2015 to, in their words, "foster a community around a constellation of high-quality projects that orchestrate containers as part of a microservices architecture."

Part of the Linux Foundation, the CNCF exists to bring together developers, end-users, and vendors, including the major public cloud providers. The best-known project under the CNCF umbrella is Kubernetes itself, but the foundation also incubates and promotes other key components of the cloud native ecosystem: Prometheus, Envoy, Helm, Fluentd, gRPC, and many more.

So what exactly do we mean by *cloud native*? Like most such things, it means different things to different people, but perhaps there is some common ground.

Cloud native applications run in the cloud; that's not controversial. But just taking an existing application and running it on a cloud compute instance doesn't make it cloud native. Neither is it just about running in a container, or using cloud services such as Azure's Cosmos DB or Google's Pub/Sub, although those may well be important aspects of a cloud native application.

So let's look at a few of the characteristics of cloud native systems that most people can agree on:

Automatable
: If applications are to be deployed and managed by machines, instead of humans, they need to abide by common standards, formats, and interfaces. Kubernetes provides these standard interfaces in a way that means application developers don't even need to worry about them.

Ubiquitous and flexible
: Because they are decoupled from physical resources such as disks, or any specific knowledge about the compute node they happen to be running on, containerized microservices can easily be moved from one node to another, or even one cluster to another.

Resilient and scalable
: Traditional applications tend to have single points of failure: the application stops working if its main process crashes, or if the underlying machine has a hardware failure, or if a network resource becomes congested. Cloud native applications, because they are inherently distributed, can be made highly available through redundancy and graceful degradation.

Dynamic
: A container orchestrator such as Kubernetes can schedule containers to take maximum advantage of available resources. It can run many copies of them to achieve high availability, and perform rolling updates to smoothly upgrade services without ever dropping traffic.

Observable
: Cloud native apps, by their nature, are harder to inspect and debug. So a key requirement of distributed systems is *observability*: monitoring, logging, tracing, and metrics all help engineers understand what their systems are doing (and what they're doing wrong).

Distributed
: Cloud native is an approach to building and running applications that takes advantage of the distributed and decentralized nature of the cloud. It's about how your application works, not where it runs. Instead of deploying your code as a single entity (known as a *monolith*), cloud native applications tend to be composed of multiple, cooperating, distributed *microservices*. A microservice is simply a self-contained service that does one thing. If you put enough microservices together, you get an application.

It's not just about microservices

However, microservices are not a panacea. Monoliths are easier to understand, because everything is in one place, and you can trace the interactions of different parts. But it's hard to scale monoliths, both in terms of the code itself, and the teams of developers who maintain it. As the code grows, the interactions between its various parts grow exponentially, and the system as a whole grows beyond the capacity of a single brain to understand it all.

A well-designed cloud native application is composed of microservices, but deciding what those microservices should be, where the boundaries are, and how the different services should interact is no easy problem. Good cloud native service design consists of making wise choices about how to separate the different parts of your architecture. However, even a well-designed cloud native application is still a distributed system, which makes it inherently complex, difficult to observe and reason about, and prone to failure in surprising ways.

While cloud native systems tend to be distributed, it's still possible to run monolithic applications in the cloud, using containers, and gain considerable business value from doing so. This may be a step on the road to gradually migrating parts of the monolith outward to modern microservices, or a stopgap measure pending the redesign of the system to be fully cloud native.

The Future of Operations

Operations, infrastructure engineering, and system administration are highly skilled jobs. Are they at risk in a cloud native future? We think not.

Instead, these skills will only become more important. Designing and reasoning about distributed systems is hard. Networks and container orchestrators are compli-

cated. Every team developing cloud native applications will need operations skills and knowledge. Automation frees up staff from boring, repetitive manual work to deal with more complex, interesting, and fun problems that computers can't yet solve for themselves.

That doesn't mean all current operations jobs are guaranteed. Sysadmins used to be able to get by without coding skills, except maybe cooking up the odd simple shell script. In the cloud, that won't fly.

In a software-defined world, the ability to write, understand, and maintain software becomes critical. If you can't or won't learn new skills, the world will leave you behind —and it's always been that way.

Distributed DevOps

Rather than being concentrated in a single operations team that services other teams, ops expertise will become distributed among many teams.

Each development team will need at least one ops specialist, responsible for the health of the systems or services the team provides. She will be a developer, too, but she will also be the domain expert on networking, Kubernetes, performance, resilience, and the tools and systems that enable the other developers to deliver their code to the cloud.

Thanks to the DevOps revolution, there will no longer be room in most organizations for devs who can't ops, or ops who don't dev. The distinction between those two disciplines is obsolete, and is rapidly being erased altogether. Developing and operating software are merely two aspects of the same thing.

Some Things Will Remain Centralized

Are there limits to DevOps? Or will the traditional central IT and operations team disappear altogether, dissolving into a group of roving internal consultants, coaching, teaching, and troubleshooting ops issues?

We think not, or at least not entirely. Some things still benefit from being centralized. It doesn't make sense for each application or service team to have its own way of detecting and communicating about production incidents, for example, or its own ticketing system, or deployment tools. There's no point in everybody reinventing their own wheel.

Developer Productivity Engineering

The point is that self-service has its limits, and the aim of DevOps is to speed up development teams, not slow them down with unnecessary and redundant work.

Yes, a large part of traditional operations can and should be devolved to other teams, primarily those that involve code deployment and responding to code-related incidents. But to enable that to happen, there needs to be a strong central team building and supporting the DevOps ecosystem in which all the other teams operate.

Instead of calling this team *operations*, we like the name *developer productivity engineering* (DPE). DPE teams do whatever's necessary to help developers do their work better and faster: operating infrastructure, building tools, busting problems.

And while developer productivity engineering remains a specialist skill set, the engineers themselves may move outward into the organization to bring that expertise where it's needed.

Lyft engineer Matt Klein has suggested that, while a pure DevOps model makes sense for startups and small firms, as an organization grows, there is a natural tendency for infrastructure and reliability experts to gravitate toward a central team. But he says that team can't be scaled indefinitely:

> By the time an engineering organization reaches ~75 people, there is almost certainly a central infrastructure team in place starting to build common substrate features required by product teams building microservices. But there comes a point at which the central infrastructure team can no longer both continue to build and operate the infrastructure critical to business success, while also maintaining the support burden of helping product teams with operational tasks.
>
> —Matt Klein

At this point, not every developer can be an infrastructure expert, just as a single team of infrastructure experts can't service an ever-growing number of developers. For larger organizations, while a central infrastructure team is still needed, there's also a case for embedding *site reliability engineers* (SREs) into each development or product team. They bring their expertise to each team as consultants, and also form a bridge between product development and infrastructure operations.

You Are the Future

If you're reading this book, it means you're going to be part of this new cloud native future. In the remaining chapters, we'll cover all the knowledge and skills you'll need as a developer or operations engineer working with cloud infrastructure, containers, and Kubernetes.

Some of these things will be familiar, and some will be new, but we hope that when you've finished the book you'll feel more confident in your own ability to acquire and master cloud native skills. Yes, there's a lot to learn, but it's nothing you can't handle. You've got this!

Now read on.

Summary

We've necessarily given you a rather quick tour of the cloud native DevOps landscape, but we hope it's enough to bring you up to speed with some of the problems that cloud, containers, and Kubernetes solve, and how they're likely to change the IT business. If you're already familiar with this, then we appreciate your patience.

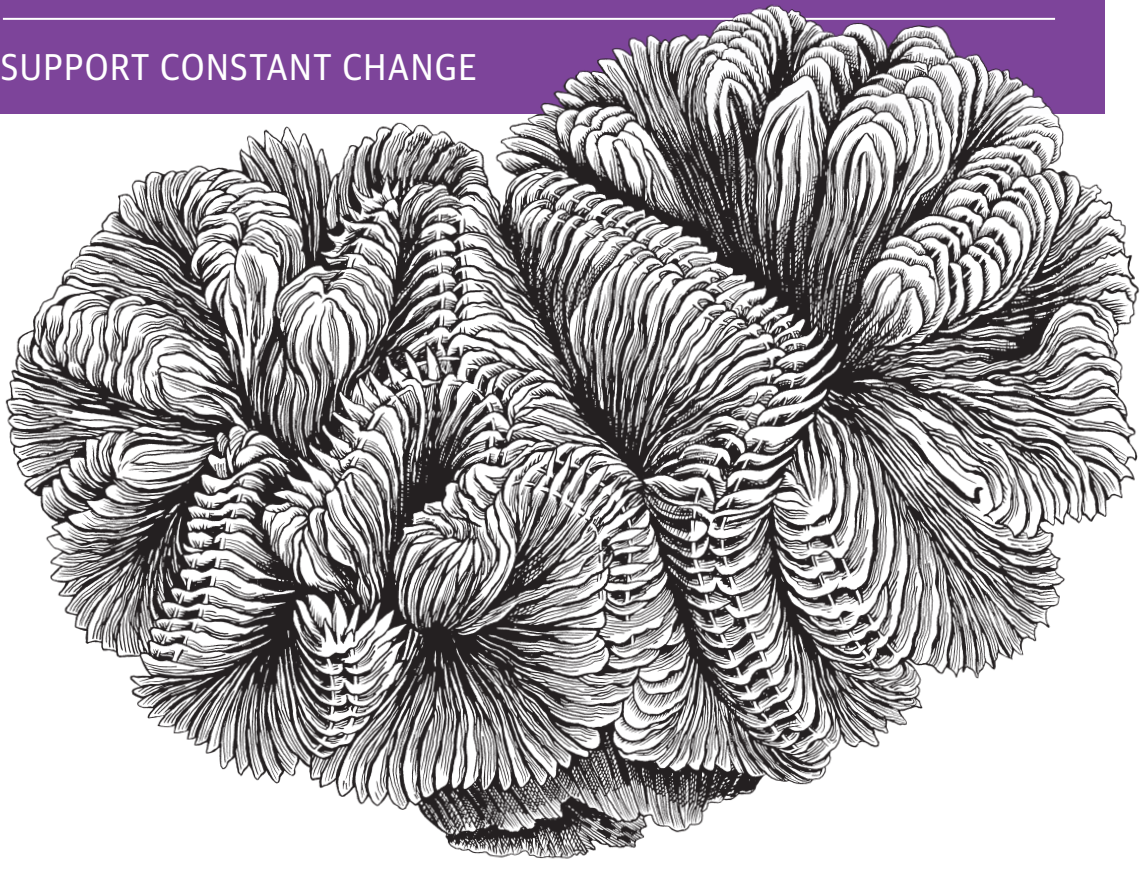
A quick recap of the main points before we move on to meet Kubernetes in person in the next chapter:

- Cloud computing frees you from the expense and overhead of managing your own hardware, making it possible for you to build resilient, flexible, scalable distributed systems.
- DevOps is a recognition that modern software development doesn't stop at shipping code: it's about closing the feedback loop between those who write the code and those who use it.
- DevOps also brings a code-centric approach and good software engineering practices to the world of infrastructure and operations.
- Containers allow you to deploy and run software in small, standardized, self-contained units. This makes it easier and cheaper to build large, diverse, distributed systems, by connecting together containerized microservices.
- Orchestration systems take care of deploying your containers, scheduling, scaling, networking, and all the things that a good system administrator would do, but in an automated, programmable way.
- Kubernetes is the de facto standard container orchestration system, and it's ready for you to use in production right now, today.
- *Cloud native* is a useful shorthand for talking about cloud-based, containerized, distributed systems, made up of cooperating microservices, dynamically managed by automated infrastructure as code.
- Operations and infrastructure skills, far from being made obsolete by the cloud native revolution, are and will become more important than ever.
- It still makes sense for a central team to build and maintain the platforms and tools that make DevOps possible for all the other teams.
- What will go away is the sharp distinction between software engineers and operations engineers. It's all just software now, and we're all engineers.

O'REILLY®

Building Evolutionary Architectures

SUPPORT CONSTANT CHANGE



Neal Ford, Rebecca Parsons & Patrick Kua

CHAPTER 1
Software Architecture

Developers have long struggled to coin a succinct, concise definition of software architecture because the scope is large and ever-changing. Ralph Johnson famously defined software architecture as "the important stuff (whatever that is)." The architect's job is to understand and balance all of those important things (whatever they are).

An initial part of an architect's job is to understand the business or domain requirements for a proposed solution. Though these requirements operate as the motivation for utilizing software to solve a problem, they are ultimately only one factor that architects should contemplate when crafting an architecture. Architects must also consider numerous other factors, some explicit (e.g., performance service-level agreements) and others implicit to the nature of the business (e.g., the company is embarking on a mergers and acquisition spree). Therefore, the craft of software architecture manifests in the ability of architects to analyze business and domain requirements along with other important factors to find a solution that balances all concerns optimally. The scope of software architecture is derived from the combination of all these architectural factors, as shown in Figure 1-1.

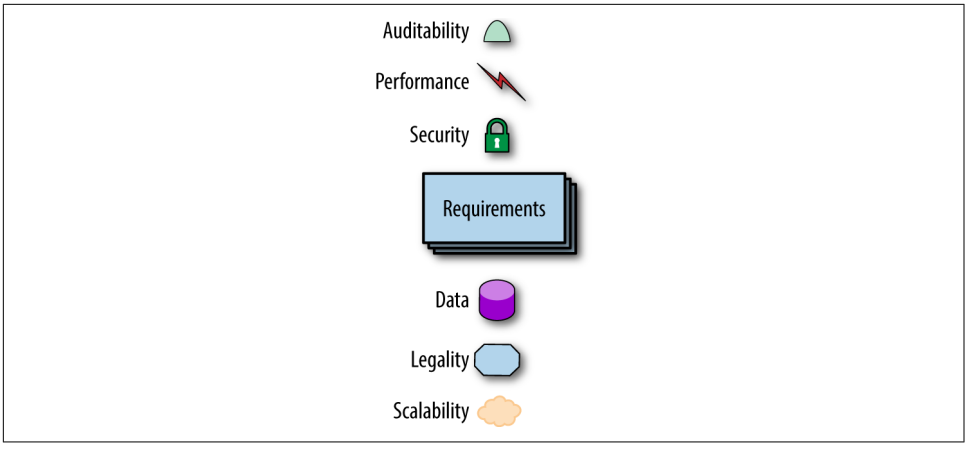


Figure 1-1. The entire scope of architecture encompasses requirements plus "-ilities"

As seen in Figure 1-1, business and domain requirements exist alongside other architecture concerns (defined by architects). This includes a wide range of external factors that can alter the decision process on what and how to build a software system. For a sampling, check out the list in Table 1-1:

Table 1-1. Partial list of "-ilities"

accessibility	accountability	accuracy	adaptability	administrability
affordability	agility	auditability	autonomy	availability
compatibility	composability	configurability	correctness	credibility
customizability	debugability	degradability	determinability	demonstrability
dependability	deployability	discoverability	distributability	durability
effectiveness	efficiency	usability	extensibility	failure transparency
fault tolerance	fidelity	flexibility	inspectability	installability
integrity	interoperability	learnability	maintainability	manageability
mobility	modifiability	modularity	operability	orthogonality
portability	precision	predictability	process capabilities	producibility
provability	recoverability	relevance	reliability	repeatability
reproducibility	resilience	responsiveness	reusability	robustness
safety	scalability	seamlessness	self-sustainability	serviceability
securability	simplicity	stability	standards compliance	survivability
sustainability	tailorability	testability	timeliness	traceability

When building software, architects must determine the most important of these "-ilities." However, many of these factors oppose one another. For example, achieving both high performance and extreme scalability can be difficult because achieving

both requires a careful balance of architecture, operations, and many other factors. As a result, the necessary analysis in architecture design and the inevitable clash of competing factors requires balance, but balancing the pros and cons of each architectural decision leads to the *tradeoffs* so commonly lamented by architects. In the last few years, incremental developments in core engineering practices for software development have laid the foundation for rethinking how architecture changes over time and on ways to protect important architectural characteristics as this evolution occurs. This book ties those parts together with a new way to think about *architecture* and *time*.

We want to add a new standard "-ility" to software architecture—*evolvability*.

Evolutionary Architecture

Despite our best efforts, software becomes harder to change over time. For a variety of reasons, the parts that comprise software systems defy easy modification, becoming more brittle and intractable over time. Changes in software projects are usually driven by a reevaluation of functionality and/or scope. But another type of change occurs outside the control of architects and long-term planners. Though architects like to be able to strategically plan for the future, the constantly changing software development ecosystem makes that difficult. Since we can't avoid change, we need to exploit it.

How Is Long-term Planning Possible When Everything Changes All the Time?

In the biological world, the environment changes constantly from both natural and man-made causes. For example, in the early 1930s, Australia had problems with cane beetles, which rendered the production and harvesting sugar cane crops less profitable. In response, in June 1935, the then Bureau of Sugar Experiment Stations introduced a predator, the cane toad, previously only native to south and middle America.[1] After being bred in captivity a number of young toads were released in North Queensland in July and August 1935. With poisonous skin and no native predators, the cane toads spread widely; there are an estimated 200 million in existence today. The moral: introducing changes to a highly dynamic (eco)system can yield unpredictable results.

The software development ecosystem consists of all the tools, frameworks, libraries, and best practices—the accumulated state of the art in software development. This

1 Clarke, G. M., Gross, S., Matthews, M., Catling, P. C., Baker, B., Hewitt, C. L., Crowther, D., & Saddler, S. R. 2000, Environmental Pest Species in Australia, Australia: State of the Environment, Second Technical Paper Series (Biodiversity), Department of the Environment and Heritage, Canberra.

ecosystem forms an equilibrium—much like a biological system—that developers can understand and build things within. However, that equilibrium is dynamic—new things come along constantly, initially upsetting the balance until a new equilibrium emerges. Visualize a unicyclist carrying boxes: *dynamic* because she continues to adjust to stay upright and *equilibrium* because she continues to maintain balance. In the software development ecosystem, each new innovation or practice may disrupt the status quo, forcing the establishment of a new equilibrium. Metaphorically, we keep tossing more boxes onto the unicyclist's load, forcing her to reestablish balance.

In many ways, architects resemble our hapless unicyclist, constantly both balancing and adapting to changing conditions. The engineering practices of Continuous Delivery represent such a tectonic shift in the equilibrium: Incorporating formerly siloed functions such as operations into the software development lifecycle enabled new perspectives on what *change* means. Enterprise architects can no longer rely on static 5-year plans because the entire software development universe will evolve in that timeframe, rendering every long-term decision potentially moot.

Disruptive change is hard to predict even for savvy practitioners. The rise of containers via tools like Docker is an example of an unknowable industry shift. However, we can trace the rise of containerization via a series of small, incremental steps. Once upon a time, operating systems, application servers, and other infrastructure were commercial entities, requiring licensing and great expense. Many of the architectures designed in that era focused on efficient use of shared resources. Gradually, Linux became good enough for many enterprises, reducing the *monetary* cost of operating systems to zero. Next, DevOps practices like automatic machine provisioning via tools like Puppet or Chef made Linux *operationally* free. Once the ecosystem became free and widely used, consolidation around common portable formats was inevitable; thus, Docker. But containerization couldn't have happened without all the evolutionary steps leading to that end.

The programming platforms we use exemplify constant evolution. Newer versions of a programming language offer better application programming interfaces (APIs) to improve the flexibility or applicability toward new problems; newer programming languages offer a different paradigm and different set of constructs. For example, Java was introduced as a C++ replacement to ease the difficulty of writing networking code and to improve memory management issues. When we look at the past 20 years, we observe that many languages still continually evolve their APIs while totally new programming languages appear to regularly attack newer problems. The evolution of programming languages is demonstrated in Figure 1-2.

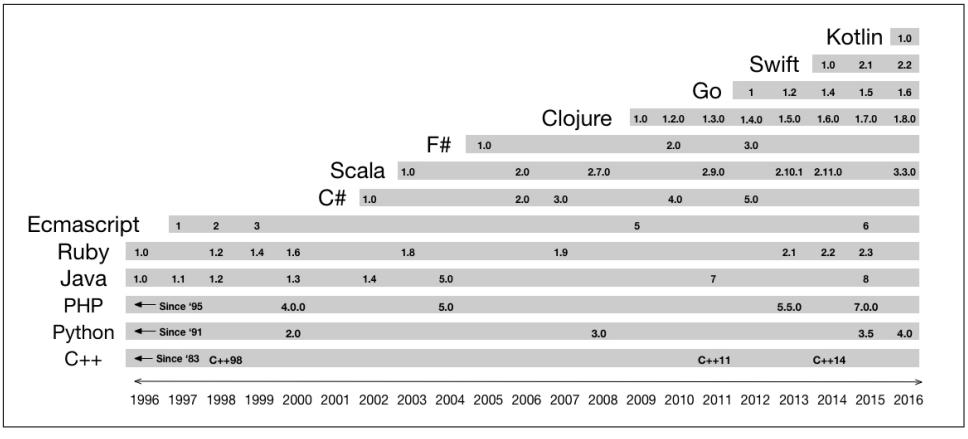


Figure 1-2. The evolution of popular programming languages

Regardless of which particular aspect of software development—the programming platform, languages, the operating environment, persistence technologies, and so on—we expect constant change. Although we cannot predict when changes in the technical or domain landscape will occur, or which changes will persist, we know change is inevitable. Consequently, we should architect our systems knowing the technical landscape will change.

If the ecosystem constantly changes in unexpected ways, and predictability is impossible, what is the *alternative* to fixed plans? Enterprise architects and other developers must learn to adapt. Part of the traditional reasoning behind making long-term plans was financial; software changes were expensive. However, modern engineering practices invalidate that premise by making change less expensive by automating formerly manual processes and other advances such as DevOps.

For years, many smart developers recognized that some parts of their systems were harder to modify than others. That's why *software architecture* is defined as the "parts hard to change later." This convenient definition partitioned the things you *can* modify without much effort from truly difficult changes. Unfortunately, this definition also evolved into a blind spot when thinking about architecture: Developers' assumption that change is difficult becomes a self-fulfilling prophecy.

Several years ago, some innovative software architects revisited the "hard to change later" problem in a new light: what if we build changeability *into* the architecture? In other words, if *ease of change* is a bedrock principle of the architecture, then change is no longer difficult. Building evolvability into architecture in turn allows a whole new set of behaviors to emerge, upsetting the dynamic equilibrium again.

Even if the ecosystem doesn't change, what about the gradual erosion of architectural characteristics that occurs? Architects design architectures, but then expose them to

the messy real world of *implementing* things atop the architecture. How can architects protect the important parts they have defined?

Once I've Built an Architecture, How Can I Prevent It from Gradually Degrading Over Time?

An unfortunate decay, often called *bit rot*, occurs in many organizations. Architects choose particular architectural patterns to handle the business requirements and "-ilities," but those characteristics often accidentally degrade over time. For example, if an architect has created a layered architecture with presentation at the top, persistence at the bottom, and several layers in between, developers who are working on reporting will often ask permission to directly access persistence from the presentation layer, bypassing the other layers, for performance reasons. Architects build layers to isolate change. Developers then bypass those layers, increasing coupling and invalidating the reasoning behind the layers.

Once they have defined the important architectural characteristics, how can architects *protect* those characteristics to ensure they don't erode? Adding *evolvability* as an architectural characteristics implies protecting the other characteristics as the system evolves. For example, if an architect has designed an architecture for scalability, she doesn't want that characteristic to degrade as the system evolves. Thus, *evolvability* is a meta-characteristic, an architectural wrapper that protects all the other architectural characteristics.

In this book, we illustrate that a side effect of an evolutionary architecture is mechanisms to protect the important architecture characteristics. We explore the ideas behind *continual architecture*: building architectures that have no end state and are designed to evolve with the ever-changing software development ecosystem, and including built-in protections around important architectural characteristics. We don't attempt to define software architecture in totality; many other definitions exist. We focus instead on extending current definitions to adding *time* and *change* as first-class architectural elements.

Here is our definition of evolutionary architecture:

> An evolutionary architecture supports guided, incremental change across multiple dimensions.

Incremental Change

Incremental change describes two aspects of software architecture: how teams build software incrementally and how they deploy it.

During development, an architecture that allows small, incremental changes is easier to evolve because developers have a smaller scope of change. For deployment, incre-

mental change refers to the level of modularity and decoupling for business features and how they map to architecture. An example is in order.

Let's say that PenultimateWidgets, a large seller of widgets, has a catalog page backed by a microservice architecture and modern engineering practices. One of the page's features is the ability of users to rate different widgets with star ratings. Other services within PenultimateWidgets' business also need ratings (customer service representatives, shipping provider evaluation, and so on), so they all share the star rating service. One day, the star rating team releases a new version alongside the existing one that allows half-star ratings—a small but significant upgrade. The other services that require ratings aren't required to move to the new version, but rather gradually migrate as convenient. Part of PenultimateWidgets' DevOps practices include architectural monitoring of not only the services but also the routes between services. When the operations group observes that no one has routed to a particular service within a given time interval, they automatically disintegrate that service from the ecosystem.

This is an example of incremental change at the architectural level: the original service can run alongside the new one as long as other services need it. Teams can migrate to new behavior at their leisure (or as need dictates), and the old version is automatically garbage collected.

Making incremental change successful requires coordination of a handful of Continuous Delivery practices. Not all these practices are required in all cases but rather commonly occur together in the wild. We discuss how to achieve incremental change in Chapter 3.

Guided Change

Once architects have chosen important characteristics, they want to *guide* changes to the architecture to protect those characteristics. For that purpose, we borrow a concept from evolutionary computing called *fitness functions*. A fitness function is an objective function used to summarize how close a prospective design solution is to achieving the set aims. In evolutionary computing, the fitness function determines whether an algorithm has improved over time. In other words, as each variant of an algorithm is generated, the fitness functions determine how "fit" each variant is based on how the designer of the algorithm defined "fit."

We have a similar goal in evolutionary architecture—as architecture evolves, we need mechanisms to evaluate how changes impact the important characteristics of the architecture and prevent degradation of those characteristics over time. The fitness function metaphor encompasses a variety of mechanisms we employ to ensure architecture doesn't change in undesirable ways, including metrics, tests, and other verification tools. When an architect identifies an architectural characteristic they want to

protect as things evolve, they define one or more fitness functions to protect that feature.

Historically, a portion of architecture has often been viewed as a governance activity, and architects have only recently accepted the notion of enabling change through architecture. Architectural fitness functions allow decisions in the context of the organization's needs and business functions, while making the basis for those decisions explicit and testable. Evolutionary architecture is not an unconstrained, irresponsible approach to software development. Rather, it is an approach that balances the need for rapid change and the need for rigor around systems and architectural characteristics. The fitness function drives architectural decision making, guiding the architecture while allowing the changes needed to support changing business and technology environments.

We use *fitness functions* to create evolutionary guidelines for architectures; we cover them in detail in Chapter 2.

Multiple Architectural Dimensions

> There are no separate systems. The world is a continuum. Where to draw a boundary around a system depends on the purpose of the discussion.
>
> —Donella H. Meadows

Classical Greek physics gradually learned to analyze the universe based on fixed points, culminating in Classical Mechanics. However, more precise instruments and more complex phenomena gradually refined that definition toward relativity in the early 20th century. Scientists realized that what they previously viewed as isolated phenomenon in fact interact relative to one another. Since the 1990s, enlightened architects have increasingly viewed software architecture as multidimensional. Continuous Delivery expanded that view to encompass operations. However, software architects often focus primarily on *technical* architecture, but that is only one dimension of a software project. If architects want to create an architecture that can evolve, they must consider all parts of the system that change affects. Just like we know from physics that everything is relative to everything else, architects know there are many dimensions to a software project.

To build evolvable software systems, architects must think beyond just the technical architecture. For example, if the project includes a relational database, the structure and relationship between database entities will evolve over time as well. Similarly, architects don't want to build a system that evolves in a manner that exposes a security vulnerability. These are all examples of *dimensions* of architecture—the parts of architecture that fit together in often orthogonal ways. Some dimensions fit into what are often called *architectural concerns* (the list of "-ilities" above), but *dimensions* are actually broader, encapsulating things traditionally outside the purview of technical

architecture. Each project has dimensions the architect must consider when thinking about evolution. Here are some common dimensions that affect evolvability in modern software architectures:

Technical
: The implementation parts of the architecture: the frameworks, dependent libraries, and the implementation language(s).

Data
: Database schemas, table layouts, optimization planning, etc. The database administrator generally handles this type of architecture.

Security
: Defines security policies, guidelines, and specifies tools to help uncover deficiencies.

Operational/System
: Concerns how the architecture maps to existing physical and/or virtual infrastructure: servers, machine clusters, switches, cloud resources, and so on.

Each of these perspectives forms a *dimension* of the architecture—an intentional partitioning of the parts supporting a particular perspective. Our concept of architectural dimensions encompasses traditional architectural characteristics ("-ilities") plus any other role that contributes to building software. Each of these forms a perspective on architecture that we want to preserve as our problem evolves and the world around us changes.

A variety of partitioning techniques exist for conceptually carving up architectures. For example, the 4 + 1 architecture View Model, which focuses on different perspectives from different roles and was incorporated into the IEEE definition of software architecture, splits the ecosystem into *logical*, *development*, *process*, and *physical* views. In the well-known book Software Systems Architecture, the authors posit a catalog of viewpoints on software architecture, spanning a larger set of roles. Similarly, Simon Brown's C4 notation partitions concerns for aid in conceptual organization. In this text, in contrast, we don't attempt to create a taxonomy of dimensions but rather recognize the ones extant in existing projects. Pragmatically, regardless of which category a particular important concern falls into, the architect must still protect that dimension. Different projects have differing concerns, leading to unique sets of dimensions for a given project. Any of these techniques provide useful insight, particularly in new projects, but existing projects must deal with the realities of what exists.

When architects think in terms of architectural dimensions, it provides a mechanism by which they can analyze the evolvability of different architectures by assessing how each important dimension reacts to change. As systems become more intertwined with competing concerns (scalability, security, distribution, transactions, and so on), architects must expand the dimensions they track on projects. To build an evolvable

system, architects must think about how the system might evolve across all the important dimensions.

The entire architectural scope of a project consists of the software requirements plus the other dimensions. We can use fitness functions to protect those characteristics as the architecture and the ecosystem evolve together through time, as illustrated in Figure 1-3.

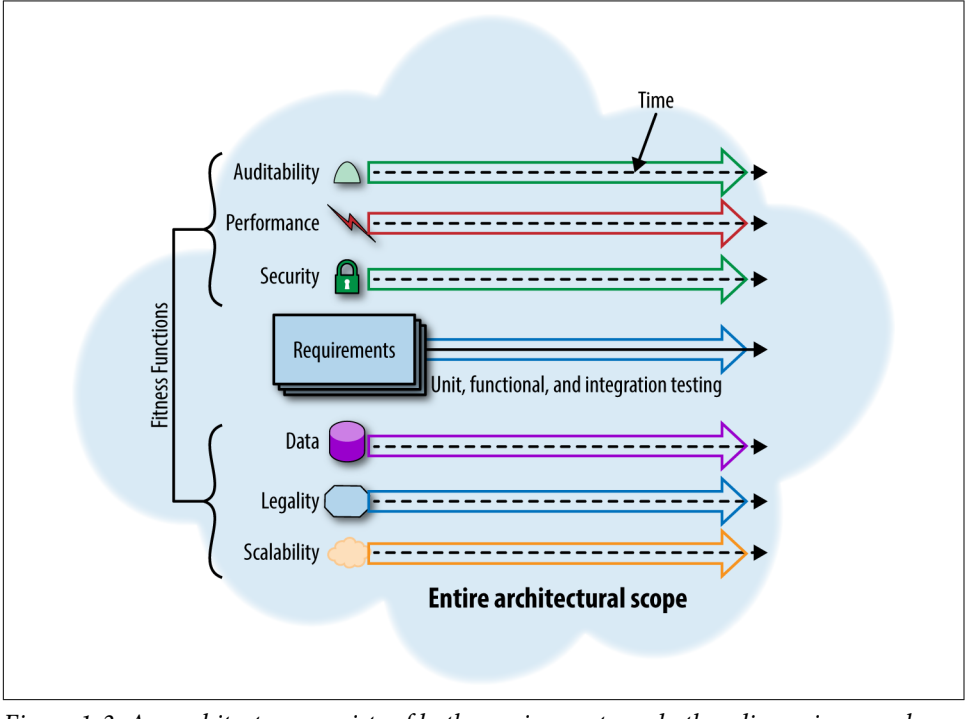


Figure 1-3. An architecture consists of both requirements and other dimensions, each protected by fitness functions

In Figure 1-3, the architects have identified *auditability*, *data*, *security*, *performance*, *legality*, and *scalability* as the additional architectural characteristics important for this application. As the business requirements evolve over time, each of the architectural characteristics utilize fitness functions to protect their integrity as well.

While the authors of this text stress the importance of a holistic view of architecture, we also realize that a large part of evolving architecture concerns technical architecture patterns and related topics like coupling and cohesion. We discuss how technical architecture coupling affects evolvability in Chapter 4 and the impacts of data coupling in Chapter 5.

Coupling applies to more than just structural elements in software projects. Many software companies have recently discovered the impact of team structure on surprising things like architecture. We discuss all aspects of coupling in software, but the team impact comes up so early and often that we need to discuss it here.

Conway's Law

In April 1968, Melvin Conway submitted a paper to Harvard Business Review called, "How Do Committees Invent?". In this paper, Conway introduced the notion that the social structures, particularly the communication paths between people, inevitably influence final product design.

As Conway describes, in the very early stage of the design, a high-level understanding of the system is made to understand how to break down areas of responsibility into different patterns. The way that a group breaks down a problem affects choices that they can make later. He codified what has become known as *Conway's Law*:

> Organizations which design systems … are constrained to produce designs which are copies of the communication structures of these organizations.
>
> —Melvin Conway

As Conway notes, when technologists break down problems into smaller chunks to delegate, we introduce coordination problems. In many organizations, formal communication structures or rigid hierarchy appear to solve this coordination problem but often lead to inflexible solutions. For example, in a layered architecture where the team is separated by technical function (user interface, business logic, and so on), solving common problems that cut vertically across layers increases coordination overhead. People who have worked in startups and then have joined joined large multinational corporations have likely experienced the contrast between the nimble, adaptable culture of the former and the inflexible communication structures of the latter. A good example of Conway's Law in action might be trying to change the contract between two services, which could be difficult if the successful change of a service owned by one team requires the coordinated and agreed-upon effort of another.

In his paper, Conway was effectively warning software architects to pay attention not only to the architecture and design of the software, but also the delegation, assignment, and coordination of the work between teams.

In many organizations teams are divided according to their functional skills. Some common examples include:

Front-end developers
: A team with specialized skills in a particular user interface (UI) technology (e.g., HTML, mobile, desktop).

Back-end developers
: A team with unique skills in building back-end services, sometimes API tiers.

Database developers
: A team with unique skills in building storage and logic services.

In organizations with functional silos, management divides teams to make their Human Resources department happy without much regard to engineering efficiency. Although each team may be good at their part of the design (e.g., building a screen, adding a back-end API or service, or developing a new storage mechanism), to release a new business capability or feature, all three teams must be involved in building the feature. Teams typically optimize for efficiency for their immediate tasks rather than the more abstract, strategic goals of the business, particularly when under schedule pressure. Instead of delivering an end-to-end feature value, teams often focus on delivering components that may or may not work well with each other.

In this organizational split, a feature dependent on all three teams takes longer as each team works on their component at different times. For example, consider the common business change of updating the Catalog page. That change entails the UI, business rules, and database schema changes. If each team works in their own silo, they must coordinate schedules, extending the time required to implement the feature. This is a great example of how team structure can impact architecture and the ability to evolve.

As Conway noted in his paper by noting *every time a delegation is made and somebody's scope of inquiry is narrowed, the class of design alternatives which can be effectively pursued is also narrowed*. Stated another way, it's hard for someone to change something if the thing she wants to change is owned by someone else. Software architects should pay attention to how work is divided and delegated to align architectural goals with team structure.

Many companies who build architectures such as microservices structure their teams around service boundaries rather than siloed technical architecture partitions. In the ThoughtWorks Technology Radar, we call this the Inverse Conway Maneuver. Organization of teams in such a manner is ideal because team structure will impact myriad dimensions of software development and should reflect the problem size and scope. For example, when building a microservices architecture, companies typically structure teams that that resemble the architecture by cutting across functional silos and including team members who cover every angle of the business and technical aspects of the architecture.

Structure teams to look like your target architecture, and it will be easier to achieve it.

Introducing PenultimateWidgets and Their Inverse Conway Moment

Throughout this book, we use the example company, PenultimateWidgets, the Second to Last Widget Dealer, a large online seller of widgets (a variety of little unspecified things). The company is gradually updating much of their IT infrastructure. They have some legacy systems they want to keep around for a while, and new strategic systems that require more iterative approaches. Throughout the chapters, we'll highlight many of the problems and solutions PenultimateWidgets develops to address those needs.

The first observation their architects made concerned the software development teams. The old monolithic applications utilized a layered architecture, separating presentation, business logic, persistence, and operations. Their team mirrors these functions: All the UI developers sit together, developers and database administrators have their own silo, and operations is outsourced to a third party.

When the developers started working on the new architectural elements, a microservices architecture with fine-grained services, the coordination costs skyrocketed. Because the services were built around domains (such as *CustomerCheckout*) rather than technical architecture, making a change to a single domain required a crippling amount of coordination across their silos.

Instead, PenultimateWidgets applied the Inverse Conway Maneuver and built cross-functional teams that matched the purview of the service: each service team consists of service owner, a few developers, a business analyst, a database administrator (DBA), a quality assurance (QA) person, and an operations person.

Team impact shows up in many places throughout the book, with examples of how many consequences it has.

Why Evolutionary?

A common question about evolutionary architecture concerns the name itself: why call it *evolutionary* architecture and not something else? Other possible names include *incremental*, *continual*, *agile*, *reactive*, and *emergent*, to name just a few. But each of these terms misses the mark here. The definition of evolutionary architecture that we state here includes two critical characteristics: incremental and guided.

The terms *continual*, *agile*, and *emergent* all capture the notion of change over time, which is clearly a critical characteristic of an evolutionary architecture, but none of these terms explicitly capture any notion of *how* an architecture changes or what the desired end state architecture might be. While all the terms imply a changing environment, none of them cover what the architecture should look like. The *guided* part of our definition reflects the architecture we want to achieve—our end goal.

We prefer the word *evolutionary* over *adaptable* because we are interested in architectures that undergo fundamental evolutionary change, not ones that have been patched and adapted into increasingly incomprehensible accidental complexity. *Adapting* implies finding some way to make something work regardless of the elegance or longevity of the solution. To build architectures that truly evolve, architects must support genuine change, not jury-rigged solutions. Going back to our biological metaphor, *evolutionary* is about the process of having a system that is fit for purpose and can survive the ever-changing environment in which it operates. Systems may have individual adaptations, but as architects, we should care about the overall evolvable system.

Summary

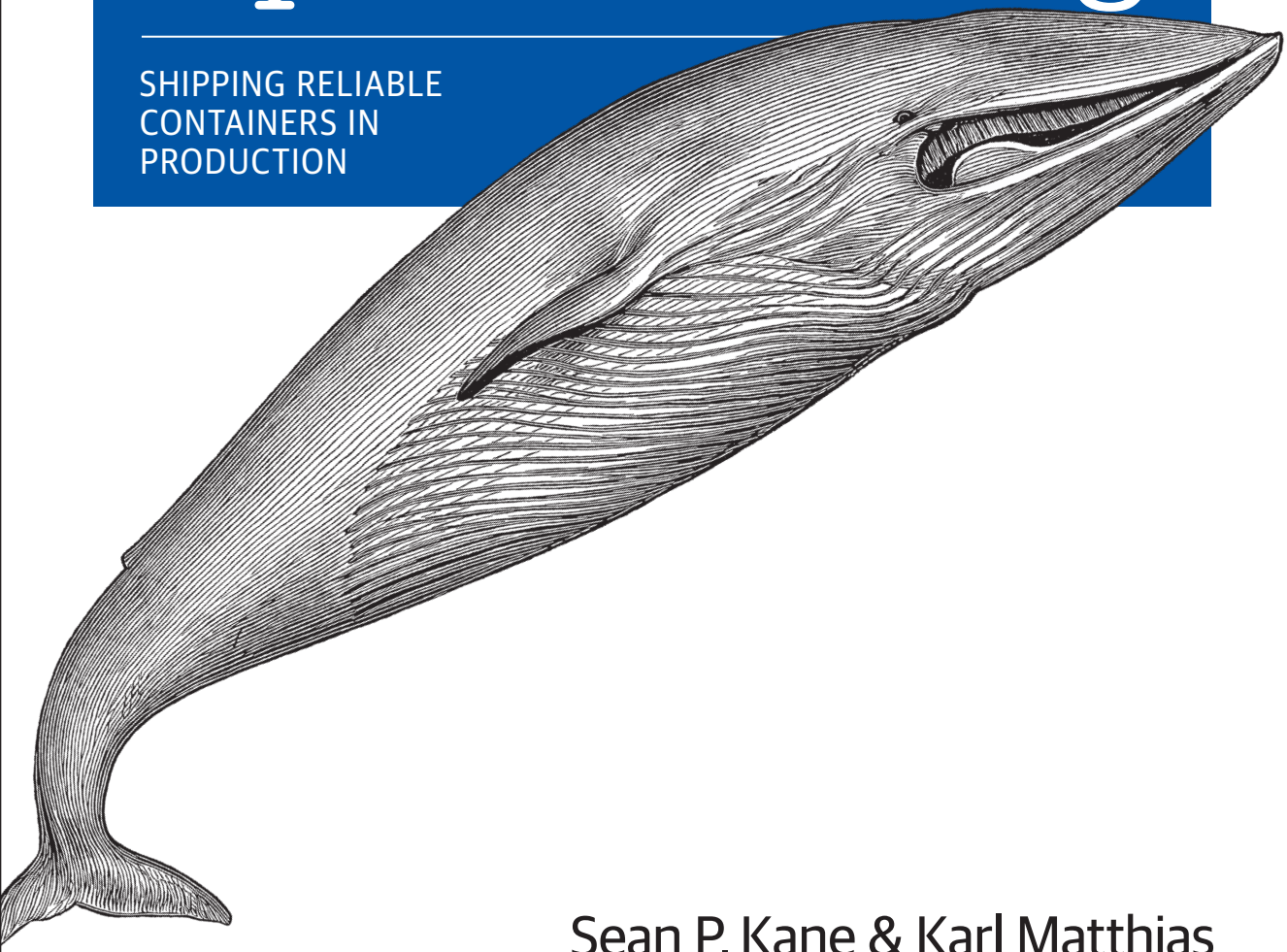
An evolutionary architecture consists of three primary aspects: incremental change, fitness functions, and appropriate coupling. In the remainder of the book, we discuss each of these factors separately, then combine them to address what it takes to build and maintain architectures that support constant change.

O'REILLY®

2nd Edition

Docker
Up & Running

SHIPPING RELIABLE CONTAINERS IN PRODUCTION



Sean P. Kane & Karl Matthias

CHAPTER 2

The Docker Landscape

Before you dive into configuring and installing Docker, a broad survey is in order to explain what Docker is and what it brings to the table. It is a powerful technology, but not a tremendously complicated one at its core. In this chapter, we'll cover the generalities of how Docker works, what makes it powerful, and some of the reasons you might use it. If you're reading this, you probably have your own reasons to use Docker, but it never hurts to augment your understanding before you jump in.

Don't worry—this chapter should not hold you up for too long. In the next chapter, we'll dive right into getting Docker installed and running on your system.

Process Simplification

Because Docker is a piece of software, it may not be obvious that it can also have a big positive impact on company and team processes if it is adopted well. So, let's dig in and see how Docker can simplify both workflows and communication. This usually starts with the deployment story. Traditionally, the cycle of getting an application to production often looks something like the following (illustrated in Figure 2-1):

1. Application developers request resources from operations engineers.
2. Resources are provisioned and handed over to developers.
3. Developers script and tool their deployment.
4. Operations engineers and developers tweak the deployment repeatedly.
5. Additional application dependencies are discovered by developers.
6. Operations engineers work to install the additional requirements.
7. Loop over steps 5 and 6 *N* more times.

8. The application is deployed.

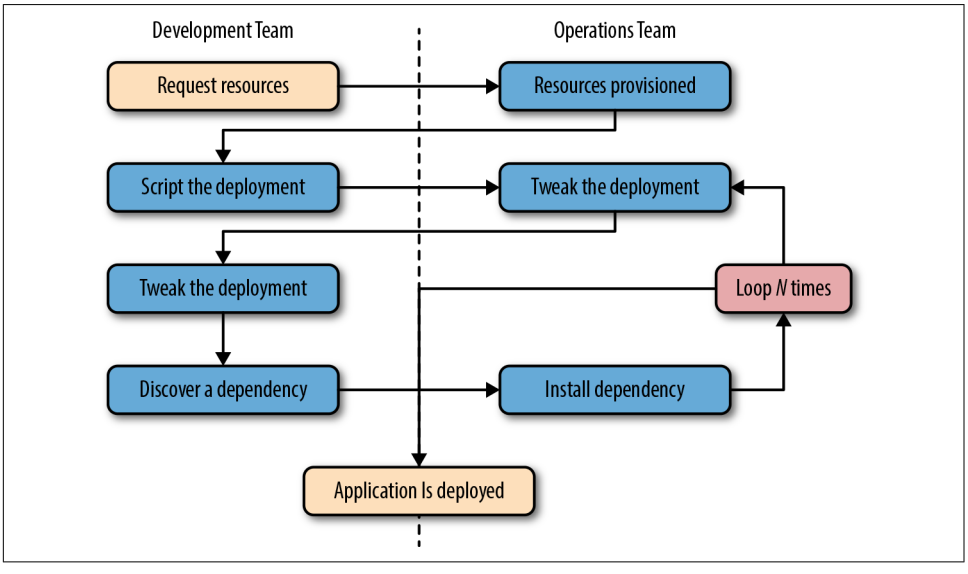


Figure 2-1. A traditional deployment workflow (without Docker)

Our experience has shown that when you are following traditional processes, deploying a brand new application into production can take the better part of a week for a complex new system. That's not very productive, and even though DevOps practices work to alleviate some of the barriers, it often requires a lot of effort and communication between teams of people. This process can be both technically challenging and expensive, but even worse, it can limit the kinds of innovation that development teams will undertake in the future. If deploying software is hard, time-consuming, and dependent on resources from another team, then developers may just build everything into the existing application in order to avoid suffering the new deployment penalty.

Push-to-deploy systems like Heroku have shown developers what the world can look like if you are in control of most of your dependencies as well as your application. Talking with developers about deployment will often turn up discussions of how easy things are on Heroku or similar systems. If you're an operations engineer, you've probably heard complaints about how much slower your internal systems are compared with deploying on Heroku.

Heroku is a whole environment, not just a container engine. While Docker doesn't try to be everything that is included in Heroku, it provides a clean separation of responsibilities and encapsulation of dependencies, which results in a similar boost in productivity. Docker also allows even more fine-grained control than Heroku by putting developers in control of everything, down to the OS distribution on which they ship

their application. Some of the tooling and orchestrators built (e.g., Kubernetes, Swarm, or Mesos) on top of Docker now aim to replicate the simplicity of Heroku. But even though these platforms wrap more around Docker to provide a more capable and complex environment, a simple platform that uses only Docker still provides all of the core process benefits without the added complexity of a larger system.

As a company, Docker adopts an approach of "batteries included but removable." This means that they want their tools to come with everything most people need to get the job done, while still being built from interchangeable parts that can easily be swapped in and out to support custom solutions.

By using an image repository as the hand-off point, Docker allows the responsibility of building the application image to be separated from the deployment and operation of the container. What this means in practice is that development teams can build their application with all of its dependencies, run it in development and test environments, and then just ship the exact same bundle of application and dependencies to production. Because those bundles all look the same from the outside, operations engineers can then build or install standard tooling to deploy and run the applications. The cycle described in Figure 2-1 then looks somewhat like this (illustrated in Figure 2-2):

1. Developers build the Docker image and ship it to the registry.
2. Operations engineers provide configuration details to the container and provision resources.
3. Developers trigger deployment.

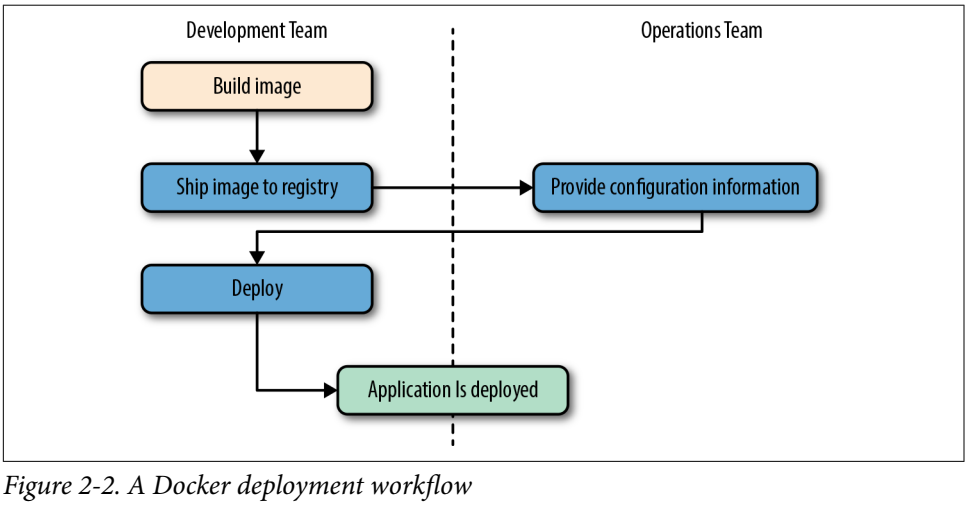


Figure 2-2. A Docker deployment workflow

This is possible because Docker allows all of the dependency issues to be discovered during the development and test cycles. By the time the application is ready for first deployment, that work is done. And it usually doesn't require as many handoffs between the development and operations teams. That's a lot simpler and saves a lot of time. Better yet, it leads to more robust software through testing of the deployment environment before release.

Broad Support and Adoption

Docker is already well supported, with the majority of the large public clouds offering some direct support for it. For example, Docker runs in AWS via multiple products like Elastic Container Service (ECS), Elastic Container Service for Kubernetes (EKS), Fargate, and Elastic Beanstalk. Docker can also be used on Google AppEngine, Google Kubernetes Engine, Red Hat OpenShift, IBM Cloud, Microsoft Azure, Rackspace Cloud, Docker's own Docker Cloud, and many more. At DockerCon 2014, Google's Eric Brewer announced that Google would be supporting Docker as its primary internal container format. Rather than just being good PR for these companies, what this meant for the Docker community was that a lot of money began to back the stability and success of the Docker platform.

Further building its influence, Docker's containers are becoming the lingua franca between cloud providers, offering the potential for "write once, run anywhere" cloud applications. When Docker released their `libswarm` development library at DockerCon 2014, an engineer from Orchard demonstrated deploying a Docker container to a heterogeneous mix of cloud providers at the same time. This kind of orchestration had not been easy before because every cloud provider provided a different API or toolset for managing instances, which were usually the smallest item you could manage with an API. What was only a promise from Docker in 2014 has since become fully mainstream as the largest companies continue to invest in the platform, support, and tooling. With most providers offering some form of Docker orchestration as well as the container runtime itself, Docker is already well supported for nearly any kind of workload in common production environments. If all of your tooling is around Docker, your applications can be deployed in a cloud-agnostic manner, allowing a huge new flexibility not previously possible.

That covers Docker containers and tooling, but what about OS vendor support and adoption? The Docker client runs directly on most major operating systems, and the server can run on Linux or Windows Server. The vast majority of the ecosystem is built around Linux servers, but other platforms are increasingly being supported. The beaten path is and will likely continue to be Linux servers running Linux containers.

It is actually possible to run Windows containers natively (without a VM) on 64-bit versions of Windows Server 2016. However, 64-bit versions of Windows 10 Professional still require Hyper-V to provide the Windows Server kernel that is used for Windows containers. We dive into a little more detail about this in "Windows Containers" on page 110.

In order to support the growing demand for Docker tooling in development environments, Docker has released easy-to-use implementations for macOS and Windows. These appear to run natively but are still sitting on top of a Linux kernel underneath. Docker has traditionally been developed on the Ubuntu Linux distribution, but most Linux distributions and other major operating systems are now supported where possible. RedHat, for example, has gone all-in on containers and all of their platforms have first-class support for Docker. With the near ubiquity of containers in the Linux realm, we now have distributions like Red Hat's CoreOS, which is built entirely on top of Docker containers, either running on Docker itself or under their own `rkt` runtime.

In the first years after Docker's release, a set of competitors and service providers voiced concerns about Docker's proprietary image format. Containers on Linux did not have a standard image format, so Docker, Inc., created their own according to the needs of their business.

Service providers and commercial vendors were particularly reluctant to build platforms subject to the whim of a company with somewhat overlapping interests to their own. Docker as a company faced some public challenges in that period as a result. In order to support wider adoption, Docker, Inc., then helped sponsor the Open Container Initiative (OCI) in June 2015. The first full specification from that effort was released in July 2017, and was based in large part on version 2 of the Docker image format. The OCI is working on the certification process and it will soon be possible for implementations to be OCI certified. In the meantime, there are at least four runtimes claiming to implement the spec: `runc`, which is part of Docker; `railcar`, an alternate implementation by Oracle; Kata Containers (formerly Clear Containers) from Intel, Hyper, and the OpenStack Foundation, which run a mix of containers and virtual machines; and finally, the gVisor runtime from Google, which is implemented entirely in user space. This set is likely to grow, with plans for CoreOS's `rkt` to become OCI compliant, and others are likely to follow.

The space around deploying containers and orchestrating entire systems of containers continues to expand, too. Many of these are open source and available both on premise and as cloud or SaaS offerings from various providers, either in their own clouds or yours. Given the amount of investment pouring into the Docker space, it's likely that Docker will continue to cement itself further into the fabric of the modern internet.

Architecture

Docker is a powerful technology, and that often means something that comes with a high level of complexity. And, under the hood, Docker is fairly complex; however, its fundamental user-facing structure is indeed a simple client/server model. There are several pieces sitting behind the Docker API, including `containerd` and `runc`, but the basic system interaction is a client talking over an API to a server. Underneath this simple exterior, Docker heavily leverages kernel mechanisms such as iptables, virtual bridging, cgroups, namespaces, and various filesystem drivers. We'll talk about some of these in Chapter 11. For now, we'll go over how the client and server work and give a brief introduction to the network layer that sits underneath a Docker container.

Client/Server Model

It's easiest to think of Docker as consisting of two parts: the client and the server/daemon (see Figure 2-3). Optionally there is a third component called the registry, which stores Docker images and their metadata. The server does the ongoing work of building, running, and managing your containers, and you use the client to tell the server what to do. The Docker daemon can run on any number of servers in the infrastructure, and a single client can address any number of servers. Clients drive all of the communication, but Docker servers can talk directly to image registries when told to do so by the client. Clients are responsible for telling servers what to do, and servers focus on hosting containerized applications.

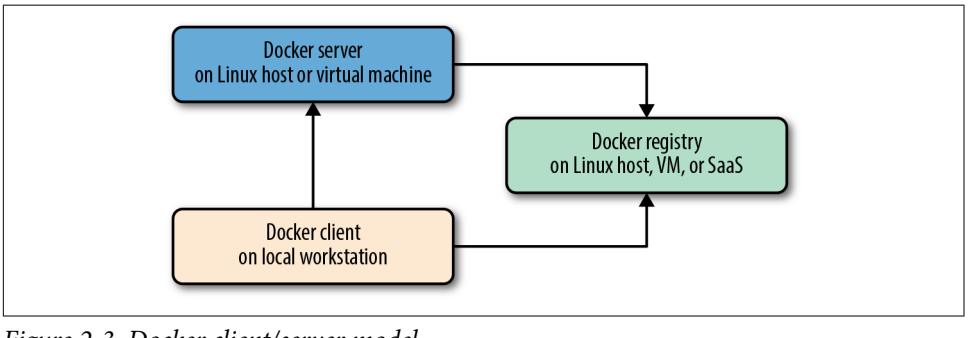


Figure 2-3. Docker client/server model

Docker is a little different in structure from some other client/server software. It has a `docker` client and a `dockerd` server, but rather than being entirely monolithic, the server then orchestrates a few other components behind the scene on behalf of the client, including `docker-proxy`, `runc`, `containerd`, and sometimes `docker-init`. Docker cleanly hides any complexity behind the simple server API, though, so you can just think of it as a client and server for most purposes. Each Docker host will normally have one Docker server running that can manage a number of containers. You can then use the `docker` command-line tool client to talk to the server, either

from the server itself or, if properly secured, from a remote client. We'll talk more about that shortly.

Network Ports and Unix Sockets

The `docker` command-line tool and `dockerd` daemon talk to each other over network sockets. Docker, Inc., has registered two ports with IANA for use by the Docker daemon and client: TCP ports 2375 for unencrypted traffic and 2376 for encrypted SSL connections. Using a different port is easily configurable for scenarios where you need to use different settings. The default setting for the Docker installer is to use only a Unix socket to make sure the system defaults to the most secure installation, but this is also easily configurable. The Unix socket can be located in different paths on different operating systems, so you should check where yours is located (often */var/run/docker.sock*). If you have strong preferences for a different location, you can usually specify this at install time or simply change the server configuration afterward and restart the daemon. If you don't, then the defaults will probably work for you. As with most software, following the defaults will save you a lot of trouble if you don't need to change them.

Robust Tooling

Among the many things that have led to Docker's growing adoption is its simple and powerful tooling. Since its initial release, its capabilities have been expanding ever wider, thanks to efforts from the Docker community at large. The tooling that Docker ships with supports building Docker images, basic deployment to individual Docker daemons, a distributed mode called Swarm mode, and all the functionality needed to actually manage a remote Docker server. Beyond the included Swarm mode, community efforts have focused on managing whole fleets (or clusters) of Docker servers and scheduling and orchestrating container deployments.

One point of confusion for people who search for Docker Swarm information on the internet is that there are two different things sharing that name. There was an older, now deprecated project that was a standalone application, called "Docker Swarm," and there is a newer, built-in Swarm that we refer to here as "Swarm mode." You'll need to carefully look at the search results to identify the relevant ones. Over time this conflict will hopefully become moot as the older product fades into history.

Docker has also launched its own orchestration toolset, including Compose, Machine, and Swarm, which creates a cohesive deployment story for developers. Docker's offerings in the production orchestration space have been largely overshadowed by Google's Kubernetes and the Apache Mesos project in current deploy-

ments. But Docker's orchestration tools remain useful, with Compose being particularly handy for local development.

Because Docker provides both a command-line tool and a remote web API, it is easy to add further tooling in any language. The command-line tool lends itself well to shell scripting, and a lot of power can easily be leveraged with simple wrappers around it. The tool also provides a huge amount of functionality itself, all targeted at an individual Docker daemon.

Docker Command-Line Tool

The command-line tool `docker` is the main interface that most people will have with Docker. This is a Go program that compiles and runs on all common architectures and operating systems. The command-line tool is available as part of the main Docker distribution on various platforms and also compiles directly from the Go source. Some of the things you can do with the Docker command-line tool include, but are not limited to:

- Build a container image.
- Pull images from a registry to a Docker daemon or push them up to a registry from the Docker daemon.
- Start a container on a Docker server either in the foreground or background.
- Retrieve the Docker logs from a remote server.
- Start a command-line shell inside a running container on a remote server.
- Monitor statistics about your container.
- Get a process listing from your container.

You can probably see how these can be composed into a workflow for building, deploying, and observing applications. But the Docker command-line tool is not the only way to interact with Docker, and it's not necessarily the most powerful.

Docker Engine API

Like many other pieces of modern software, the Docker daemon has a remote web application programming interface (API). This is in fact what the Docker command-line tool uses to communicate with the daemon. But because the API is documented and public, it's quite common for external tooling to use the API directly. This enables all manners of tooling, from mapping deployed Docker containers to servers, to automated deployments, to distributed schedulers. While it's very likely that beginners will not initially want to talk directly to the Docker API, it's a great tool to have available. As your organization embraces Docker over time, it's likely that you will increasingly find the API to be a good integration point for this tooling.

Extensive documentation for the API is on the Docker site. As the ecosystem has matured, robust implementations of Docker API libraries have emerged for all popular languages. Docker maintains SDKs for Python and Go, but the best libraries remain those maintained by third parties. We've used the much more extensive third-party Go and Ruby libraries, for example, and have found them to be both robust and rapidly updated as new versions of Docker are released.

Most of the things you can do with the Docker command-line tooling are supported relatively easily via the API. Two notable exceptions are the endpoints that require streaming or terminal access: running remote shells or executing the container in interactive mode. In these cases, it's often easier to use one of these solid client libraries or the command-line tool.

Container Networking

Even though Docker containers are largely made up of processes running on the host system itself, they usually behave quite differently from other processes at the network layer. Docker initially supported a single networking model, but now supports a robust assortment of configurations that handle most application requirements. Most people run their containers in the default configuration, called *bridge mode*. So let's take a look at how it works.

To understand bridge mode, it's easiest to think of each of your Docker containers as behaving like a host on a private network. The Docker server acts as a virtual bridge and the containers are clients behind it. A bridge is just a network device that repeats traffic from one side to another. So you can think of it like a mini–virtual network with each container acting like a host attached to that network.

The actual implementation, as shown in Figure 2-4, is that each container has its own virtual Ethernet interface connected to the Docker bridge and its own IP address allocated to the virtual interface. Docker lets you bind and expose individual or groups of ports on the host to the container so that the outside world can reach your container on those ports. That traffic passes over a proxy that is also part of the Docker daemon before getting to the container.

Docker allocates the private subnet from an unused RFC 1918 private subnet block. It detects which network blocks are unused on the host and allocates one of those to the virtual network. That is bridged to the host's local network through an interface on the server called `docker0`. This means that all of the containers are on a network together and can talk to each other directly. But to get to the host or the outside world, they go over the `docker0` virtual bridge interface. As we mentioned, inbound traffic goes over the proxy. This proxy is fairly high performance but can be limiting if you run high-throughput applications in containers.

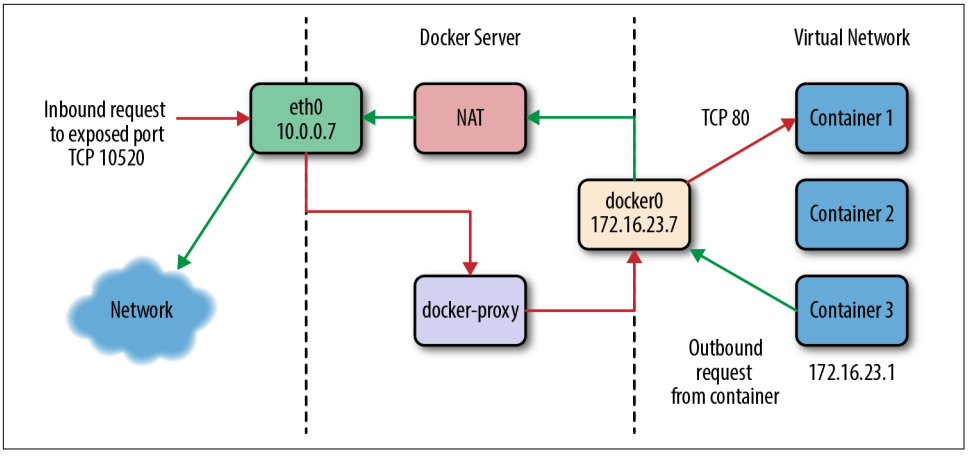


Figure 2-4. The network on a typical Docker server

There is a dizzying array of ways in which you can configure Docker's network layer, from allocating your own network blocks to configuring your own custom bridge interface. People often run with the default mechanisms, but there are times when something more complex or specific to your application is required. You can find much more detail about Docker networking in the documentation, and we will cover more details in the Chapter 11.

When developing your Docker workflow, you should definitely get started with the default networking approach. You might later find that you don't want or need this default virtual network. Networking is configurable per container, and you can switch off the whole virtual network layer entirely for a container using the `--net=host` switch to `docker run`. When running in that mode, Docker containers use the host's own network devices and address and no virtual interfaces or bridge are provisioned. Note that host networking has security implications you might need to consider. Other network topologies are possible and discussed in Chapter 11.

Getting the Most from Docker

Like most tools, Docker has a number of great use cases, and others that aren't so good. You can, for example, open a glass jar with a hammer. But that has its downsides. Understanding how to best use the tool, or even simply determining if it's the right tool, can get you on the correct path much more quickly.

To begin with, Docker's architecture aims it squarely at applications that are either stateless or where the state is externalized into data stores like databases or caches. Those are the easiest to containerize. Docker enforces some good development prin-

ciples for this class of application, and we'll talk later about how that's powerful. But this means doing things like putting a database engine inside Docker is a bit like swimming against the current. It's not that you can't do it, or even that you shouldn't do it; it's just that this is not the most obvious use case for Docker, so if it's the one you start with, you may find yourself disappointed early on. Databases that run well in Docker are often now deployed this way, but this is not the simple path. Some good applications for beginning with Docker include web frontends, backend APIs, and short-running tasks like maintenance scripts that might normally be handled by cron.

If you focus first on building an understanding of running stateless or externalized-state applications inside containers, you will have a foundation on which to start considering other use cases. We strongly recommend starting with stateless applications and learning from that experience before tackling other use cases. Note that the community is working hard on how to better support stateful applications in Docker, and there are likely to be many developments in this area.

Containers Are Not Virtual Machines

A good way to start shaping your understanding of how to leverage Docker is to think of containers not as virtual machines but as very lightweight wrappers around a single Unix process. During actual implementation, that process might spawn others, but on the other hand, one statically compiled binary could be all that's inside your container (see "Outside Dependencies" on page 203 for more information). Containers are also ephemeral: they may come and go much more readily than a traditional virtual machine.

Virtual machines are by design a stand-in for real hardware that you might throw in a rack and leave there for a few years. Because a real server is what they're abstracting, virtual machines are often long-lived in nature. Even in the cloud where companies often spin virtual machines up and down on demand, they usually have a running lifespan of days or more. On the other hand, a particular container might exist for months, or it may be created, run a task for a minute, and then be destroyed. All of that is OK, but it's a fundamentally different approach than the one virtual machines are typically used for.

To help drive this differentiation home, if you run Docker on a Mac or Windows system you are leveraging a Linux virtual machine to run `dockerd`, the Docker server. However, on Linux `dockerd` can be run natively and therefore there is no need for a virtual machine to be run anywhere on the system (see Figure 2-5).

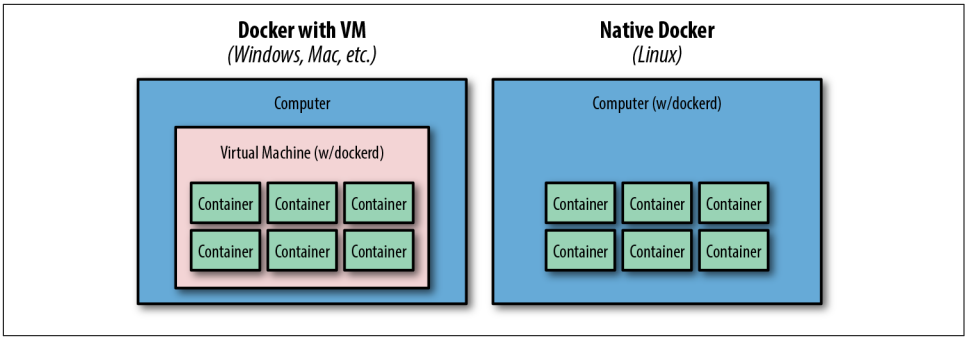


Figure 2-5. Typical Docker installations

Limited Isolation

Containers are isolated from each other, but that isolation is probably more limited than you might expect. While you can put limits on their resources, the default container configuration just has them all sharing CPU and memory on the host system, much as you would expect from colocated Unix processes. This means that unless you constrain them, containers can compete for resources on your production machines. That is sometimes what you want, but it impacts your design decisions. Limits on CPU and memory use are encouraged through Docker, but in most cases they are not the default like they would be from a virtual machine.

It's often the case that many containers share one or more common filesystem layers. That's one of the more powerful design decisions in Docker, but it also means that if you update a shared image, you may want to recreate a number of containers that still use the older one.

Containerized processes are just processes on the Docker server itself. They are running on the same exact instance of the Linux kernel as the host operating system. They even show up in the `ps` output on the Docker server. That is utterly different from a hypervisor, where the depth of process isolation usually includes running an entirely separate instance of the operating system kernel for each virtual machine.

This light containment can lead to the tempting option of exposing more resources from the host, such as shared filesystems to allow the storage of state. But you should think hard before further exposing resources from the host into the container unless they are used exclusively by the container. We'll talk about security of containers later, but generally you might consider helping to enforce isolation further by applying SELinux (Security-Enhanced Linux) or AppArmor policies rather than compromising the existing barriers.

By default, many containers use UID 0 to launch processes. Because the container is *contained*, this seems safe, but in reality it isn't. Because everything is running on the same kernel, many types of security vulnerabilities or simple misconfiguration can give the container's `root` user unauthorized access to the host's system resources, files, and processes. Refer to "Security" on page 264 for a discussion of how to mitigate this.

Containers Are Lightweight

We'll get more into the details of how this works later, but creating a container takes very little space. A quick test reveals that a newly created container from an existing image takes a whopping 12 kilobytes of disk space. That's pretty lightweight. On the other hand, a new virtual machine created from a golden image might require hundreds or thousands of megabytes. The new container is so small because it is just a reference to a layered filesystem image and some metadata about the configuration. There is no copy of the data allocated to the container. Containers are just processes on the existing system, so there may not be a need to copy any data for the exclusive use of the container.

The lightness of containers means that you can use them for situations where creating another virtual machine would be too heavyweight or where you need something to be truly ephemeral. You probably wouldn't, for instance, spin up an entire virtual machine to run a `curl` command to a website from a remote location, but you might spin up a new container for this purpose.

Toward an Immutable Infrastructure

By deploying most of your applications within containers, you can start simplifying your configuration management story by moving toward an immutable infrastructure, where components are replaced entirely rather than being changed in place. The idea of an immutable infrastructure has gained popularity in response to how difficult it is, in reality, to maintain a truly idempotent configuration management codebase. As your configuration management codebase grows, it can become as unwieldy and unmaintainable as large, monolithic legacy applications.

With Docker it is possible to deploy a very lightweight Docker server that needs almost no configuration management, or in many cases, none at all. You handle all of your application management simply by deploying and redeploying containers to the server. When the server needs an important update to something like the Docker daemon or the Linux kernel, you can simply bring up a new server with the changes, deploy your containers there, and then decommission or reinstall the old server.

Container-based Linux distributions like RedHat's CoreOS are designed around this principle. But rather than requiring you to decommission the instance, CoreOS can

entirely update itself and switch to the updated OS. Your configuration and workload largely remain in your containers and you don't have to configure the OS very much at all.

Because of this clean separation between deployment and configuration of your servers, many container-based production systems are now using tools such as HashiCorp's Packer to build cloud virtual server images and then leveraging Docker to nearly or entirely avoid configuration management systems.

Stateless Applications

A good example of the kind of application that containerizes well is a web application that keeps its state in a database. Stateless applications are normally designed to immediately answer a single self-contained request, and have no need to track information between requests from one or more clients. You might also run something like ephemeral memcache instances in containers. If you think about your web application, though, it probably has local state that you rely on, like configuration files. That might not seem like a lot of state, but it means that you've limited the reusability of your container and made it more challenging to deploy into different environments, without maintaining configuration data in your codebase.

In many cases, the process of containerizing your application means that you move configuration state into environment variables that can be passed to your application from the container. Rather than baking the configuration into the container, you apply the configuration to the container at deployment time. This allows you to easily do things like use the same container to run in either production or staging environments. In most companies, those environments would require many different configuration settings, from the names of databases to the hostnames of other service dependencies.

With containers, you might also find that you are always decreasing the size of your containerized application as you optimize it down to the bare essentials required to run. We have found that thinking of anything that you need to run in a distributed way as a container can lead to some interesting design decisions. If, for example, you have a service that collects some data, processes it, and returns the result, you might configure containers on many servers to run the job and then aggregate the response on another container.

Externalizing State

If Docker works best for stateless applications, how do you best store state when you need to? Configuration is best passed by environment variables, for example. Docker supports environment variables natively, and they are stored in the metadata that makes up a container configuration. This means that restarting the container will ensure that the same configuration is passed to your application each time. It also

makes the configuration of the container easily observable while it's running, which can make debugging a lot easier.

Databases are often where scaled applications store state, and nothing in Docker interferes with doing that for containerized applications. Applications that need to store files, however, face some challenges. Storing things to the container's filesystem will not perform well, will be extremely limited by space, and will not preserve state across a container lifecycle. If you redeploy a stateful service without utilizing storage external to the container, you will lose all of that state. Applications that need to store filesystem state should be considered carefully before you put them into Docker. If you decide that you can benefit from Docker in these cases, it's best to design a solution where the state can be stored in a centralized location that could be accessed regardless of which host a container runs on. In certain cases, this might mean using a service like Amazon S3, EBS volumes, HDFS, OpenStack Swift, a local block store, or even mounting EBS volumes or iSCSI disks inside the container. Docker volume plug-ins provide some options here and are briefly discussed in Chapter 11.

Although it is possible to externalize state on an attached filesystem, it is not generally encouraged by the community, and should be considered an advanced use case. It is strongly recommended that you start with applications that don't need persistent state. There are multiple reasons why this is typically discouraged, but in almost all cases it is because it introduces dependencies between the container and the host that interfere with using Docker as a truly dynamic, horizontally scalable application delivery service. If your container relies on an attached filesystem, it can only be deployed to the system that contains this filesystem. Remote volumes that can be dynamically attached are a good solution, but also an advanced use case.

The Docker Workflow

Like many tools, Docker strongly encourages a particular workflow. It's a very enabling workflow that maps well to how many companies are organized, but it's probably a little different than what you or your team are doing now. Having adapted our own organizations' workflow to the Docker approach, we can confidently say that this change is a benefit that touches many teams in the organization. If the workflow is implemented well, it can help realize the promise of reduced communication overhead between teams.

Revision Control

The first thing that Docker gives you out of the box is two forms of revision control. One is used to track the filesystem layers that comprise images, and the other is a tagging systems for images.

Filesystem layers

Docker containers are made up of stacked filesystem layers, each identified by a unique hash, where each new set of changes made during the build process is laid on top of the previous changes. That's great, because it means that when you do a new build, you only have to rebuild the layers that follow the change you're deploying. This saves time and bandwidth because containers are shipped around as layers and you don't have to ship layers that a server already has stored. If you've done deployments with many classic deployment tools, you know that you can end up shipping hundreds of megabytes of the same data to a server over and over at each deployment. That's slow, and worse, you can't really be sure exactly what changed between deployments. Because of the layering effect, and because Docker containers include all of the application dependencies, with Docker you can be quite sure where the changes happened.

To simplify this a bit, remember that a Docker image contains everything required to run your application. If you change one line of code, you certainly don't want to waste time rebuilding every dependency your code requires into a new image. Instead, Docker will use as many base layers as it can so that only the layers affected by the code change are rebuilt.

Image tags

The second kind of revision control offered by Docker is one that makes it easy to answer an important question: what was the previous version of the application that was deployed? That's not always easy to answer. There are a lot of solutions for non-Dockerized applications, from Git tags for each release, to deployment logs, to tagged builds for deployment, and many more. If you're coordinating your deployment with Capistrano, for example, it will handle this for you by keeping a set number of previous releases on the server and then using symlinks to make one of them the current release.

But what you find in any scaled production environment is that each application has a unique way of handling deployment revisions. Many of them do the same thing, but some may be different. Worse, in heterogeneous language environments, the deployment tools are often entirely different between applications and very little is shared. So the question of "What was the previous version?" can have many answers depending on whom you ask and about which application. Docker has a built-in mechanism for handling this: it provides image tagging at deployment time. You can leave multi-

ple revisions of your application on the server and just tag them at release. This is not rocket science, and it's not functionality that is hard to find in other deployment tooling, as we mention. But it can easily be made standard across all of your applications, and everyone can have the same expectations about how things will be tagged for all applications. This makes communication easier between teams and it makes tooling much simpler because there is one source of truth for build versions.

In many examples on the internet and in this book, you will see people use the `latest` tag. This is useful when you're getting started and when you're writing examples, as it will always grab the most recent version of a image. But since this is a floating tag, it is a really bad idea to use `latest` in most production workflows, as your dependencies can get updated out from under you, and it is impossible to roll back to `latest` because the old version is no longer the one tagged `latest`. It also makes it hard to verify if the same image is running on different servers. The rule of thumb is: don't use the `latest` tag in production. It's not even a good idea to use the `latest` tag from upstream images, for the same reasons.

Building

Building applications is a black art in many organizations, where a few people know all the levers to pull and knobs to turn in order to spit out a well-formed, shippable artifact. Part of the heavy cost of getting a new application deployed is getting the build right. Docker doesn't solve all the problems, but it does provide a standardized tool configuration and toolset for builds. That makes it a lot easier for people to learn to build your applications, and to get new builds up and running.

The Docker command-line tool contains a `build` flag that will consume a *Dockerfile* and produce a Docker image. Each command in a *Dockerfile* generates a new layer in the image, so it's easy to reason about what the build is going to do by looking at the *Dockerfile* itself. The great part of all of this standardization is that any engineer who has worked with a *Dockerfile* can dive right in and modify the build of any other application. Because the Docker image is a standardized artifact, all of the tooling behind the build will be the same regardless of the language being used, the OS distribution it's based on, or the number of layers needed. The *Dockerfile* is checked into a revision control system, which also means tracking changes to the build is simplified. Modern multistage Docker builds also allow you to define the build environment separately from the final artifact image. This provides huge configurability for your build environment just like you'd have for a production container.

Most Docker builds are a single invocation of the `docker build` command and generate a single artifact, the container image. Because it's usually the case that most of the logic about the build is wholly contained in the *Dockerfile*, it's easy to create stan-

dard build jobs for any team to use in build systems like Jenkins. As a further standardization of the build process, many companies—eBay, for example—have standardized Docker containers to do the image builds from a *Dockerfile*. SaaS build offerings like Travis CI and Codeship also have first-class support for Docker builds.

Testing

While Docker itself does not include a built-in framework for testing, the way containers are built lends some advantages to testing with Docker containers.

Testing a production application can take many forms, from unit testing to full integration testing in a semi-live environment. Docker facilitates better testing by guaranteeing that the artifact that passed testing will be the one that ships to production. This can be guaranteed because we can either use the Docker SHA for the container, or a custom tag to make sure we're consistently shipping the same version of the application.

Since, by design, containers include all of their dependencies, tests run on containers are very reliable. If a unit test framework says tests were successful against a container image, you can be sure that you will not experience a problem with the versioning of an underlying library at deployment time, for example. That's not easy with most other technologies, and even Java WAR (Web application ARchive) files, for example, don't include testing of the application server itself. That same Java application deployed in a Docker container will generally also include an application server like Tomcat, and the whole stack can be smoke-tested before shipping to production.

A secondary benefit of shipping applications in Docker containers is that in places where there are multiple applications that talk to each other remotely via something like an API, developers of one application can easily develop against a version of the other service that is currently tagged for the environment they require, like production or staging. Developers on each team don't have to be experts in how the other service works or is deployed just to do development on their own application. If you expand this to a service-oriented architecture with innumerable microservices, Docker containers can be a real lifeline to developers or QA engineers who need to wade into the swamp of inter-microservice API calls.

A common practice in organizations that run Docker containers in production is for automated integration tests to pull down a versioned set of Docker containers for different services, matching the current deployed versions. The new service can then be integration-tested against the very same versions it will be deployed alongside. Doing this in a heterogeneous language environment would previously have required a lot of custom tooling, but it becomes reasonably simple to implement because of the standardization provided by Docker containers.

Packaging

Docker produces what for all intents and purposes is a single artifact from each build. No matter which language your application is written in or which distribution of Linux you run it on, you get a multilayered Docker image as the result of your build. And it is all built and handled by the Docker tooling. That's the shipping container metaphor that Docker is named for: a single, transportable unit that universal tooling can handle, regardless of what it contains. Like the container port or multimodal shipping hub, your Docker tooling will only ever have to deal with one kind of package: the Docker image. That's powerful, because it's a huge facilitator of tooling reuse between applications, and it means that someone else's off-the-shelf tools will work with your build images.

Applications that traditionally take a lot of custom configuration to deploy onto a new host or development system become totally portable with Docker. Once a container is built, it can easily be deployed on any system with a running Docker server.

Deploying

Deployments are handled by so many kinds of tools in different shops that it would be impossible to list them here. Some of these tools include shell scripting, Capistrano, Fabric, Ansible, and in-house custom tooling. In our experience with multiteam organizations, there are usually one or two people on each team who know the magical incantation to get deployments to work. When something goes wrong, the team is dependent on them to get it running again. As you probably expect by now, Docker makes most of that a nonissue. The built-in tooling supports a simple, one-line deployment strategy to get a build onto a host and up and running. The standard Docker client handles deploying only to a single host at a time, but there are a large array of tools available that make it easy to deploy into a cluster of Docker hosts. Because of the standardization Docker provides, your build can be deployed into any of these systems, with low complexity on the part of the development teams.

The Docker Ecosystem

Over the years, a wide community has formed around Docker, driven by both developers and system administrators. Like the DevOps movement, this has facilitated better tools by applying code to operations problems. Where there are gaps in the tooling provided by Docker, other companies and individuals have stepped up to the plate. Many of these tools are also open source. That means they are expandable and can be modified by any other company to fit their needs.

Docker is a commercial company that has contributed much of the core Docker source code to the open source community. Companies are strongly encouraged to join the community and contribute back to the open source efforts. If you are looking for supported versions of the core Docker tools, you can find out more about its offerings on the Docker website.

Orchestration

The first important category of tools that adds functionality to the core Docker distribution contains orchestration and mass deployment tools. Mass deployment tools like New Relic's Centurion, Spotify's Helios, and the Ansible Docker tooling still work largely like traditional deployment tools but leverage the container as the distribution artifact. They take a fairly simple, easy-to-implement approach. You get a lot of the benefits of Docker without much complexity.

Fully automatic schedulers like Apache Mesos—when combined with a scheduler like Singularity, Aurora, Marathon, or Google's Kubernetes—are more powerful options that take nearly complete control of a pool of hosts on your behalf. Other commercial entries are widely available, such as HashiCorp's Nomad, CoreOS's Tectonic, Mesosphere's DCOS, and Rancher.[1] The ecosystems of both free and commercial options continue to grow rapidly.

Atomic hosts

One additional idea that you can leverage to enhance your Docker experience is atomic hosts. Traditionally, servers and virtual machines are systems that an organization will carefully assemble, configure, and maintain to provide a wide variety of functionality that supports a broad range of usage patterns. Updates must often be applied via nonatomic operations, and there are many ways in which host configurations can diverge and introduce unexpected behavior into the system. Most running systems are patched and updated in place in today's world. Conversely, in the world of software deployments, most people deploy an entire copy of their application, rather than trying to apply patches to a running system. Part of the appeal of containers is that they help make applications even more atomic than traditional deployment models.

What if you could extend that core container pattern all the way down into the operating system? Instead of relying on configuration management to try to update, patch, and coalesce changes to your OS components, what if you could simply pull down a new, thin OS image and reboot the server? And then if something breaks, easily roll back to the exact image you were previously using?

1 Some of these commercial offerings have free editions of their platforms.

This is one of the core ideas behind Linux-based atomic host distributions, like Red Hat's CoreOS and Red Hat's original Project Atomic. Not only should you be able to easily tear down and redeploy your applications, but the same philosophy should apply for the whole software stack. This pattern helps provide very high levels of consistency and resilience to the whole stack.

Some of the typical characteristics of an atomic host are a minimal footprint, a design focused on supporting Linux containers and Docker, and atomic OS updates and rollbacks that can easily be controlled via multihost orchestration tools on both bare-metal and common virtualization platforms.

In Chapter 3, we will discuss how you can easily use atomic hosts in your development process. If you are also using atomic hosts as deployment targets, this process creates a previously unheard of amount of software stack symmetry between your development and production environments.

Additional tools

Docker is not just a standalone solution. It has a massive feature set, but there is always a case where someone needs more than it can deliver. There is a wide ecosystem of tools to either improve or augment Docker's functionality. Some good production tools leverage the Docker API, like Prometheus for monitoring, or Ansible or Spotify's Helios for orchestration. Others leverage Docker's plug-in architecture. Plug-ins are executable programs that conform to a specification for receiving and returning data to Docker. Some examples include Rancher's Convoy plug-in for managing persistent volumes on Amazon EBS volumes or over NFS mounts, Weaveworks' Weave Net network overlay, and Microsoft's Azure File Service plug-in.

There are many more good tools that either talk to the API or run as plug-ins. Many plug-ins have sprung up to make life with Docker easier on the various cloud providers. These really help with seamless integration between Docker and the cloud. As the community continues to innovate, the ecosystem continues to grow. There are new solutions and tools available in this space on an ongoing basis. If you find you are struggling with something in your environment, look to the ecosystem!

Wrap-Up

There you have it: a quick tour through Docker. We'll return to this discussion later on with a slightly deeper dive into the architecture of Docker, more examples of how to use the community tooling, and an exploration of some of the thinking behind designing robust container platforms. But you're probably itching to try it all out, so in the next chapter we'll get Docker installed and running.

O'REILLY®

Seeking SRE

CONVERSATIONS ABOUT RUNNING PRODUCTION SYSTEMS AT SCALE

Curated and edited by
David N. Blank-Edelman

CHAPTER 3

So, You Want to Build an SRE Team?

Luke Stone, Google

This chapter is for leaders in organizations that are not practicing Site Reliability Engineering (SRE). It's for the IT managers who are facing disruption from the cloud. It's for operations directors who are dealing with more and more complexity every day. It's for CTOs who are building new technical capabilities. You might be thinking about building an SRE team. Is it a good fit for your organization? I want to help you decide.

Since 2014, I've worked in Google Cloud Platform, and I've met at least a hundred leaders in this situation. My first job at Google was before SRE existed, and today I work with cloud customers who are applying SRE in their organizations. I've seen SRE teams grow from seeds in many environments. I've also seen teams that struggled to find energy and nutrition.

So, let's assume that you want to build an SRE team. Why? I've heard some themes that come up over and over. These made-up quotes illustrate the themes:

- "My friends think SRE is cool and it would look nifty on my résumé."
- "I got yelled at the last time we had an outage."
- "My organization wants more predictable reliability and is willing to pay for it."

They are cheeky distillations of more nuanced situations, but they might resonate with you. Each one points to several opportunities and pitfalls. Understanding them will help you to figure out how SRE would fit into your organization. Let's take a look at the real meaning of each one.

Choose SRE for the Right Reasons

> "My friends think SRE is cool and it would look nifty on my résumé."

The first point is about avoiding misconceptions: your organization must understand SRE at a deeper level.

If SRE seems cool to you, congratulations on your excellent taste! It is pretty cool to take something as theoretical as an error budget and apply it to a real-world problem…and to see results. It's science! It's engineering!

However, in the long run, SRE is not going to thrive in your organization based purely on its current popularity. Every outage will be a test of the team's ability to deliver real value. The bad news is that it takes a lot of work to build trust around a team that is highly visible when things go disastrously wrong. The good news is that SRE is well designed, detailed, and tested in the real world. It can be the backbone of your plan to achieve the reliability that you need.

So, you can't count on the coolness to take you very far. A well-functioning SRE team has a culture that develops over time, and it's going to be a long-term, high-effort project to start a team in your organization. We're talking months to years. If this is not your idea of fun, you can still get some quick results by adopting a few SRE practices. For example, you can practice consistently writing postmortems and reviewing them with your team. This can be useful, but don't expect it to have the same impact as adopting the full set of SRE principles.

Similarly, don't expect to have impact from your SRE team if it is not practicing the SRE approach to production engineering all day, every day. It's tempting to slap the SRE label on an existing team, change a few job titles, and claim that you've taken the first steps. You can probably guess that this is unlikely to succeed because it doesn't actually change anything about the systems themselves. An SRE team is fundamentally a team of software engineers who are pointed at the reliability problem, so if your organization doesn't sustain software engineers, it's not right for SRE. If you have software engineers but can't focus their energy on an SRE approach to production reliability, you'll have another flavor of "SRE in name only."

An effective SRE team requires a certain environment. There must be support from the ground up, also from the sides and above. You'll need to get buy-in from the SRE team members themselves, their peers (development teams and other SRE teams), and management. The team will not thrive in isolation. It needs to be integrated into your company's technical core. Ease of communication matters, so consider physical colocation with development teams or investing in high-quality video conferencing. SRE needs to be at the table for high-level decisions about reliability needs and levels of investment in people and redundant infrastructure.

The members of the team must be dedicated and trustworthy to one another. They must be skilled at automation and technical troubleshooting. They must possess the generosity necessary to continually build one another up because SRE is really a team effort and there is a saying: "No heroes." Avoid teams that have been scraped together from far-flung individuals. It might be better to repurpose a well-functioning team to bootstrap your SRE team.

SRE teams interact with other teams, particularly development teams, so these need to be bought in, too. For example, it's hard to create a blameless postmortem culture if developers or managers are pointing fingers. When paired with an SRE team, a development team should put something of value into the partnership. Here are some examples:

- Fund the team
- Agree to take some on-call shifts
- Give SRE a mechanism to prioritize SRE needs
- Dedicate the best software engineers to the SRE team

Here's a recipe for disaster: when the dev team expects the SRE team to "just fix it." SRE should be empowered to ensure that the software can be operated reliably, and the dev team should be interested in collaborating toward that goal. The developers should be expected to actively participate in production operations—for example, by pairing with SREs or taking a small slice of the SRE team's operational duties. This will make them aware of the value of the SRE team and also gives some firsthand data about what it's like to operate the system. It's also very important for business continuity in case the SRE team needs to "hand back the pager" and stop supporting the system. If the developers cannot or will not take responsibility for some portion of the production engineering tasks, SRE is not going to work.

SRE thrives in an environment of open, effective communication. It will be very difficult if the SRE team is excluded from the conversations about the future of the product or service. This can be unintentional friction, for example if the SRE team is in a remote location, works in a different time zone, or uses another language (spoken or programming language, that is). It's even worse if the SRE team is local but excluded from planning or social events.

But wait, I thought SRE was obviously cool! I mean, look at SREcon attendance growing every year, and the SRE book's a best-seller. At Google, SREs wear awesome bomber jackets and they have a logo with a dragon on it. Super cool!

If you want to make SRE successful in your organization, it definitely helps to make it cool. That doesn't mean it needs to be elitist or exclusive. There will be a tendency for other teams to reject SRE if they don't understand it, so it's very important to get all technical leaders to introduce the concepts and the people into their teams. You

might be willing to try a rotation program between dev and SRE, or a policy whereby people can freely transfer between dev and SRE teams. It's about building personal relationships and treating SRE as equals.

At a higher level, the executive leadership must be willing to back up the SRE team when it is acting according to its principles, even when the world appears to be on fire. Leadership must defer to SREs who are managing a live incident. They must be willing to sign off on Service-Level Objectives (SLOs), meaning that they are taking responsibility for defending some amount of downtime as acceptable. They must be willing to enforce error budgets, meaning enforcing consequences like feature freezes when budgets are exceeded.

Orienting to a Data-Driven Approach

> "I got yelled at the last time we had an outage."

This point is about recognizing that an organization is compatible with SRE only if it's driven by facts and data.

My grandfather was not an SRE. He was a truck driver and an entrepreneur. At one point in the 1950s he invested in a novel gadget: an alarm clock that turns off when the user screams. Very effective for getting everyone up in the morning. It's the only machine I've ever seen that does the right thing when you yell at it.

If your organization reacts to engineering problems by applying emotion, it's not ready for SRE. People will need to keep cool heads in order to get to the bottom of increasingly difficult reliability problems. When people feel threatened, they are motivated to hide information, and SRE requires an open and collaborative atmosphere. For example, some teams give awards, not punishment, to people who take responsibility for causing an outage, *if* they follow up with an appropriate incident response and postmortem.

Be alert for signs of maturity like focus on long-term benefits, incremental improvement, and acknowledgment of existing problems. If people can face problems, a blameless postmortem culture can take root. If they want to measure the pace of improvement, SLOs and error budgets will work. If they are willing to invest, automation will thrive and toil melts away.

A good sign is the use of objective metrics to drive business discussions. People agree ahead of time on the desired, measurable outcomes, and then periodic reviews check the numbers and adjust as necessary. If you have seen a scenario in which someone's review was full of bad news, and the group's reaction was to offer help, the culture might be right for SRE.

Another good sign of maturity is when the organization can reward "landings" (e.g., automation in production providing measurable savings) instead of "launches" (like

a new, shiny tool). Do you reward significant clean-ups and grunge work that improves maintainability and operational effectiveness?

SRE relies on a fact-based approach, like hard science. It can take time to get some difficult facts, like the root cause of an outage or an analysis of when the current capacity will be inadequate. It's very important that these facts get the respect they deserve. Look for people who are patient about gathering the facts. Look for people who question assumptions and demand hard evidence. Avoid people who don't have the focus or the interest to get to the bottom of a complex technical situation.

People across your organization will need to put their trust in the SRE process as their chosen tool for managing reliability. They will need to stick to the process and live with the reliability goals that they originally agreed to. An SLO and an error budget are like a contract with performance targets and consequences. Sticking to this contract in the face of an incident requires a cool head. It needs to be OK for SRE to say, "errors increased, but stayed within SLO, so this is not an emergency."

Commitment to SRE

> "My organization wants more predictable reliability and is willing to pay for it."

This brings us to the last point: your organization must be willing to invest the attention and resources required to sustain an SRE team.

SRE can be expensive. You'll want to be sure about what you're getting from it. A sustainable SRE team needs enough people to cover on-call duties without getting burned out, and enough people to have cycles for automation, tooling, and other production engineering projects. They will also need enough people to respond to the service's myriad operational needs that come up. If the team is understaffed, even by a little, the most talented people will leave and you won't achieve your goal. There is a minimum team size necessary to carry SRE responsibilities for a system. This doesn't make sense for smaller systems, so some things will be out of scope for the SRE team. They are not expected to solve everyone's problems.

Many people think that SRE is about continually increasing reliability, forever, asymptotically approaching 100%. That could be your goal, but at some point, you'll get diminishing returns on your resources. SRE is really about managing your reliability so that it doesn't get in the way of other business goals. The organization must be able to accept reliability under 100%—because no systems are perfect. Some amount of downtime must be anticipated, accepted, and even embraced. Reluctance to face this reality is the single most significant barrier to adopting SRE.

What goals would be more important than reliability? When technology is a differentiator to the business. Where innovation is important. In these cases, you might want a product development team going as fast as possible—and it would be worthwhile to

have an SRE team providing some reliability guardrails. For example, SRE could provide a deployment system with canary servers and easy rollbacks.

SRE can fit into your organization if you have enough discipline to trust the process. At first, it seems like SRE establishes another set of constraints. But, actually, people gain freedom when they adopt practices like SLOs and blameless postmortems. An SLO is a codification of clear expectations. It allows local optimizations and empowers people to reason about the reliability impacts of their decisions. A blameless postmortem is explicit permission to point out things that went wrong. It allows you to begin to address the root cause of your reliability issues. These are powerful tools. Like all good tools, they require training and maintenance. And, of course, they should be used according to their design and best purpose.

Making a Decision About SRE

I hope this chapter has given you some perspective on whether SRE is a good fit for your organization. The steps for actually starting the team can be covered in another chapter, or perhaps another book! If you're still not sure whether SRE is right for you, here are some ideas for gathering more information:

- Find some influential people in your organization (leaders and old-timers, for example) and have them read the first few chapters of the original SRE book. Do the ideas resonate? Does SRE seem realistic?
- Brainstorm some SLOs for your service. Test them with the people who are responsible for the user experience. Play "what if" with a few scenarios, like unplanned outages of various durations, planned maintenance for a feature, and degraded service.
- Try to think of a mature, important system with which to start. Are there clear reliability metrics? Is it clear who would decide on the SLOs and error budget?
- Imagine that you've built an SRE team. How do you tell if it's working well? More reliability with fewer people? At greater release velocity? Something that's measurable.

You are now well armed with questions to help you evaluate the possibility of SRE in your organization. Be sure to include your colleagues and stakeholders while you are gathering data and deliberating. This decision is too big to be taken by one person. Working together, you can make a wise decision on starting your SRE team. Good luck!

Luke Stone joined Google in 2002 as the first technical support engineer for AdSense. He witnessed SRE's impact on Google from the beginning and joined Google SRE as a founding member of the Customer Reliability Engineering team. Before Google, Luke was a system administrator and developer in academic and nonprofit organizations, and studied computer science at Stanford.

O'REILLY®

Monolith to Microservices

Evolutionary Patterns to Transform Your Monolith

Early Release

RAW & UNEDITED

Sam Newman

CHAPTER 1
Splitting the Monolith

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as he or she writes—so you can take advantage of these technologies long before the official release of these titles. The following will be Chapter 2 in the final release of the book.

In an earlier chapter, we explored how to think about migration to a microservice architecture. More specifically we explored if it was even a good idea, and if it was, then how should you go about it in terms of rolling out your new architecture and making sure you're going in the right direction.

We've discussed what a good service looks like, and why smaller servers may be better for us. We also previously discussed the importance of being able to evolve the design of our systems. But how do we handle the fact that we may already have a large number of code bases lying about that don't follow these patterns? How do we go about decomposing these monolithic applications without having to embark on a big-bang rewrite?

The monolith grows over time. It acquires new functionality and lines of code at an alarming rate. Before long it becomes a big, scary giant presence in our organization that people are scared to touch or change. But all is not lost! With the right tools at our disposal, we can slay this beast.

Throughout the rest of this chapter, we'll explore a variety of different migration patterns and tips that can help you adopt a microservice architecture. We'll look at patterns that will work for black-box vendor software, legacy systems, or monoliths that you plan to continue to maintain and evolve. For incremental rollout to work though, we have to ensure that we can continue to work with, and make use of, the existing monolithic software.

Incremental Migration Is Key

Remember that we want to make our migration incremental. We want to migrate over to a microservice architecture in small steps, allowing us to learn from the process and change our minds if needed.

To Change The Monolith, Or Not?

One of the first things you're going to have to consider as part of your migration is whether or not you plan (or are able) to change the existing monolith.

If you have the ability to change the existing system, this will give you the most flexibility in terms of the various patterns at your disposal. In some situations however, there will be a hard constraint in place, denying you this opportunity. The existing system may be a vendor product for which you don't have the source code. It may also be written in a technology that you no longer have the skills for.

There may also be softer drivers that might divert you away from changing the existing system. It's possible that the current monolith is in such a bad state that the cost of change is too hight - as a result, you want to cut your losses and start again (although as I detailed earlier, I worry that people reach this conclusion far too easily). Another possibility is that the monolith itself is being worked on by many other people, and you're worried about getting in their way. Some patterns, like the Branch By Abstraction pattern we'll explore shortly, can mitigate these issues, but you may still judge that the impact to others is too great.

In one memorable situation I was working with some colleagues to help scale a computationally heavy system. The underlying calculations were performed by a C library we were given. Our job was to collect the various inputs, pass them into the library, and retrieve and store the results. The library itself was riddled with problems. Memory leaks and horrendously inefficient API design being just two of the major causes of problems. We asked for many months for the source code for the library so we could fix these issues, but we were rebuffed.

Many years later, I caught up with the project sponsor, and asked why they hadn't let us change the underlying library. It was at that point the sponsor finally admitted - they'd lost the source code but were too embarrassed to tell us! Don't let this happen to you.

So hopefully we're in a position where we can work with, and change, the existing monolithic codebase. But if we can't, does this mean we're stuck? Quite the contrary, there are a number of patterns that can help us here. We'll be covering some of those shortly.

Cut, Copy, Or Re-implement?

Even if you have access to the existing code in the monolith, when you start migrating functionality to your new microservices, it's not always clear cut as to what to do with the existing code. Should we move the code as is, or re-implement the functionality?

If the existing monolithic codebase is sufficiently well factored you may be able to save significant time by moving the code itself. The key thing here is to understand that we want to *copy* the code from the monolith, and at this stage at least we don't want to remove this functionality from the monolith itself. Why? Because leaving the functionality in the monolith for a period of time gives you more options. It can give us a rollback point, or perhaps the opportunity to run both implementations in parallel. Further down the line, once you're happy that the migration has been successful, you can remove the functionality from the monolith.

I've observed that often the biggest barrier to migrating code directly is that existing code bases are traditionally not organized around business domain concepts, with technical categorizations being more prominent (think of all the `Model`, `View`. `Controller` package names you've seen for example). When you're trying to move business domain functionality this can be difficult - the existing codebase doesn't match that categorization, so even finding the code you're trying to move can be problematic!

If you do go down the route of reorganising your existing monolith along business domain boundaries, I can thoroughly recommend *Working Effectively with Legacy Code* (Prentice-Hall) by Michael Feathers[1]. In his book, Michael defines the concept of a *seam*—that is, a portion of the code that can be treated in isolation and worked on without impacting the rest of the codebase. He features a number of techniques to work safely within seams as a way of helping clean up code bases.

Michael's concepts of seams are very closely analogous to bounded contexts from Domain-Driven Design - a mechanism often used to describe a microservice boundary. So while Working Effectively with Legacy Code may not refer directly to Domain-Driven Design concepts, you can use the techniques in Michael's book to organism your code along these principles.

My general inclination is always to attempt to salvage the existing codebase first, before resorting to just re-implementing functionality, and the advice I gave in my previous book Building Microservices[2] was along these lines. However I have to accept that in practice, I find very few teams take the approach of refactoring their

1 *Working Effectively with Legacy Code* (Prentice-Hall) by Michael Feathers

2 Building Microservices, Sam Newman, O'Reilly Media 2015

monolith as precursor to moving to microservices. Instead, what seems more common is that once teams have identified the responsibilities of the newly created microservice, they instead do a new clean-room implementation of that functionality.

But aren't we in danger of repeating the problems associated with big bag rewrites if we start reimplementing our functionality? They key is to ensure you're only rewriting small pieces of functionality at a time, and shipping this reworked functionality to our customers regularly. If the work to reimplement the behavior of the service is a few days or weeks, it's probably fine. If the timelines start looking more like several months, I'd be re-examining my approach.

Migration Patterns

There are a large number of different techniques I have seen used as part of a microservice migration. For the rest of the chapter, we'll explore these patterns, looking at where they may be useful, and how they can be implemented. Remember, as with all patterns, these aren't universally "good" ideas - for each one I've attempted to give enough information to help you understand if they make sense in your context.

Make sure you understand the pros and cons of each of these patterns. They are not universally the "right" way to do things.

We'll start with looking at a number of techniques to allow you to migrate and integrate with the monolith; we will deal primarily with where the application code lives. To start with though, we'll look at one of the most useful and commonly used techniques, the strangler application.

Pattern: Strangler Application

A technique which has seen frequent use when doing system rewrites is called the Strangler Application. Martin Fowler first captured this pattern, inspired by a certain type of fig which seeds itself in the upper branches of trees. The fig then descends towards the ground to take root, gradually enveloping the original tree. The existing tree becomes initially a support structure for the new fig, and if taken to the final stages you may see the original tree die and rot away, leaving only the new, now self-supporting fig in it's place.

In the context of software, the parallel here is to have our new system initially be supported by, and wrapping, the existing system. The idea is that the old and the new can co-exist, giving the new system time to grow and potentially entirely replace the old system. The key benefit to this pattern, as we'll see shortly, is that it supports our goal

of allowing for incremental migration to a new system. Moreover it gives us the ability to pause and even stop the migration altogether, while still taking advantage of the new system delivered so far.

As we'll see shortly, when we implement this idea for our software, we strive to not only take incremental steps towards our new application architecture, but we will also ensure that each step is easily reversible, reducing the risk of each incremental step.

How It Works

While the stranger application has been commonly used to migrate from one monolithic system with another, we will look to migrate from a monolith to a series of microservices. This may involve actually copying the code from the monolith (if possible), or else re-implementing the functionality in question. In addition, if the functionality in question requires the persistence of state then consideration needs to be given in terms of how that state can be migrated to the new service, and potentially back again. We'll explore aspects related to data later in Chapter 2.

Implementing a strangler pattern relies on three steps, as outlined in Figure 1-1. Firstly, identify parts of the existing system that you wish to migrate. You'll need to use judgement as to which parts of the system to tackle first, using a sort of tradeoff activity. You then need to implement this functionality in your new microservice. With your new implementation ready, you need to be able to reroute calls from the monolith over to your shinny new microservice.

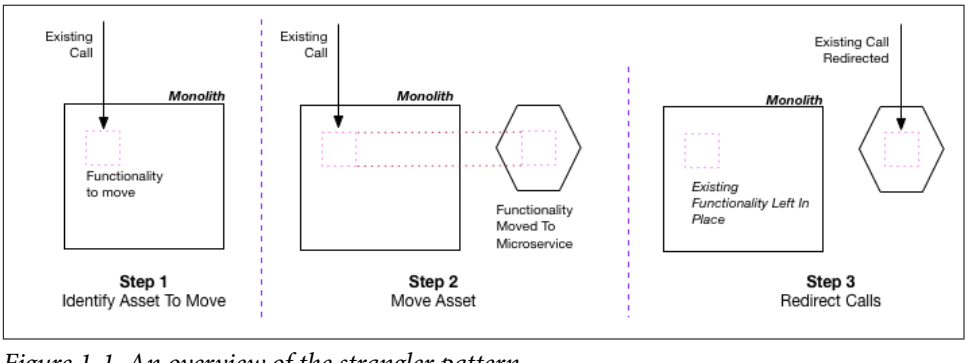


Figure 1-1. An overview of the strangler pattern

It's worth noting that until the call to the moved functionality is redirected, that the new functionality isn't technically live - even if it is deployed into a production environment. This means you could take your time getting that functionality right, working on implementing this functionality over a period of time. You could push these changes into a production environment, safe in the knowledge that it isn't yet being used. allowing us to get happy with the deployment and management aspects of your new service. I am a strong advocate of frequent check-ins and frequent releases of

software. As we'll explore later on, until the new service is live, we could freely deploy this service into our production system,,

A key point of this stranger application approach is not just that we can incrementally migrate new functionality to the new system, but that we can also rollback this change very easily if required. Remember, we all make mistakes - so we want techniques that not only allow us to make mistakes as cheaply as possible (hence lots of small steps), but which also allow us to fix our mistakes quickly.

Where To Use It

The use of a strangler pattern allows you to move functionality over to your new services architecture without having to touch or make any changes to your existing system. This has benefits when the existing monolith itself may be being worked on by other people, as this can help reduce contention. It's also very useful when the monolith is in effect a black-box system - such as a COTS product.

Occasionally, you can extract an entire end-to-end slice of functionality in one piece, as we see in Figure 1-2. This simplifies the extraction greatly, aside from concerns around data, which we'll look at later on in this chapter.

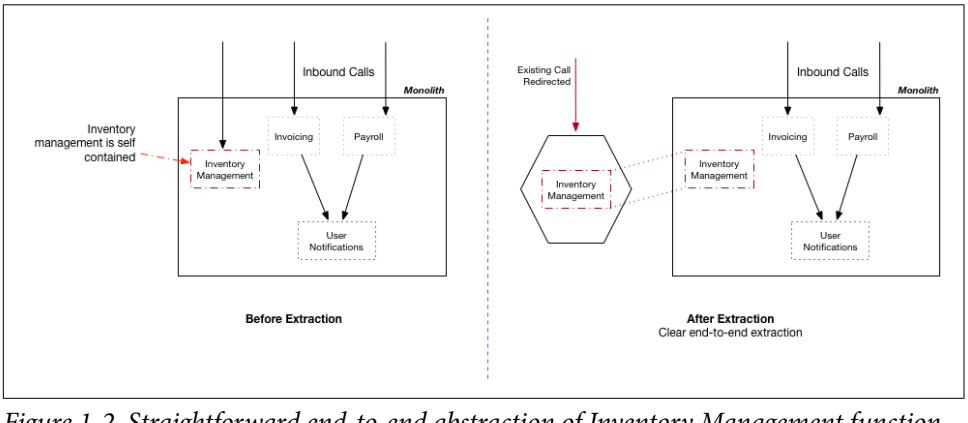


Figure 1-2. Straightforward end-to-end abstraction of Inventory Management functionality

In order to perform a clean end-to-end extraction like this, you might be inclined to extract larger groups of functionality to simplify this process. This can result in a tricky balancing act - by extracting larger slices of functionality you are taking on more work, but may simplify some of your integration challenges.

If you do want to take a smaller bite, you may have to consider more "shallow" extractions like we see in Figure 1-3. Here we are extracting `Payroll` functionality, despite the fact it makes use of other functionality that remains inside the monolith - in this example the ability to send `User Notifications`.

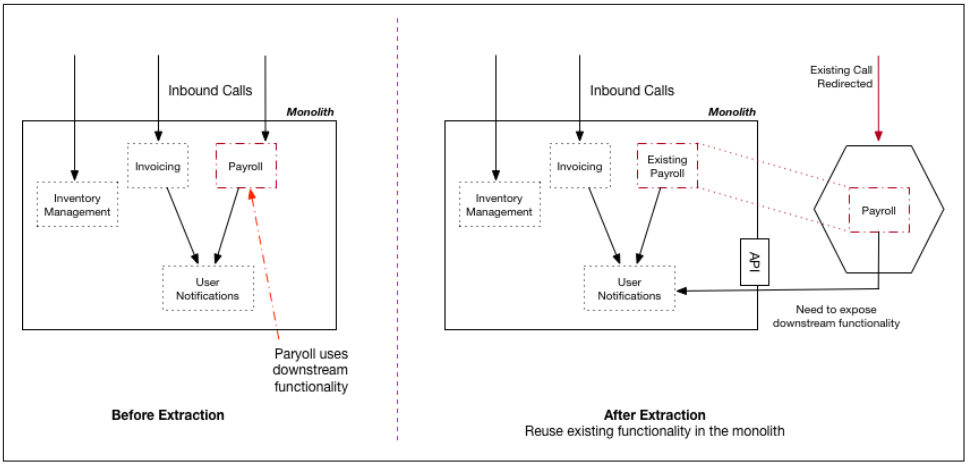


Figure 1-3. Extraction of functionality that still needs to use the monolith

Rather than also reimplementing the `User Notification` functionality, we instead expose this functionality to our new microservice by exposing it from the monolith - something that obviously would require changes to the monolith itself.

For the strangler to work though, you need to be able to clearly map the inbound call to the functionality you care about to the asset that you want to move. For example, in Figure 1-4, we'd ideally like to move out the ability to send `User Notifications` to our customers into a new service. However notifications are fired as a result of multiple inbound calls to the existing monolith - therefore we can't clearly redirect the calls from outside the system itself. Instead we'd need to look at a technique like "Pattern: Branch By Abstraction" on page 32, which we'll cover later on.

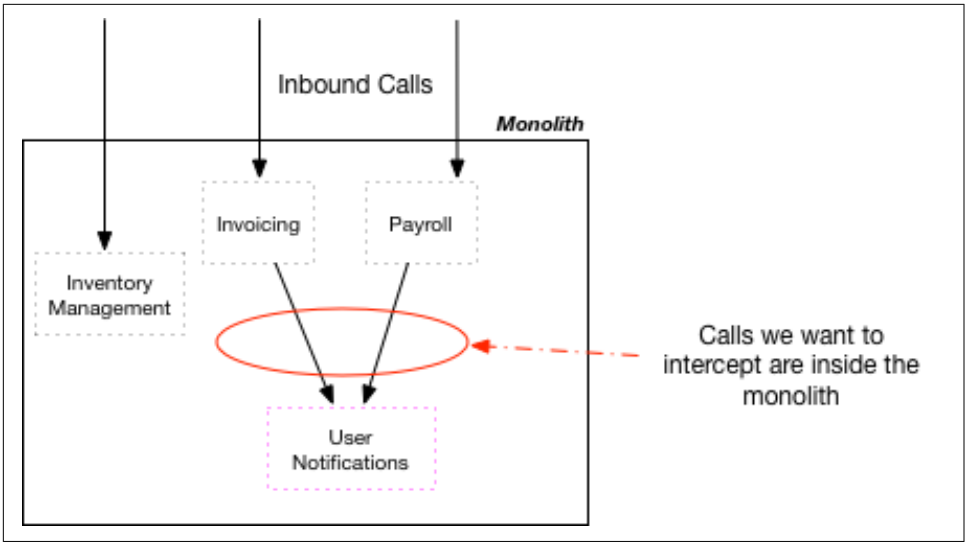


Figure 1-4. The strangler pattern doesn't work too well the functionality to be moved is deeper inside the existing system.

You will also need to consider the nature of the calls being made into the existing system. As we explore shortly, a protocol such as HTTP is very amenable to redirection. HTTP itself has the concepts of transparent redirection build in, and proxies can be used to clearly understand the nature of an inbound request, and divert it accordingly. Other types of protocols may be less amenable to redirection, for example some RPC protocols. The more work you have to do in the proxy layer to understand and potentially transform the inbound call, the less viable this option becomes.

Despite these restrictions, the strangler application has proven itself time and again to be a very useful migration technique. Given the light touch, and easy approach to handle incremental change, it's often my first port of call when exploring how to migrate a system.

Example: HTTP Proxy

HTTP has some very interesting capabilities, among them is the fact that it is very easy to intercept and redirect in a way which can be made transparent to the calling system. This means that an existing monolith with a HTTP interface is very amenable to migration through use of a strangler pattern.

In Figure 1-5, we see an existing monolithic system which exposes a HTTP interface. This application may be headless, or the HTTP interface may in fact be being called by an upstream UI. Either way the goal is the same, to insert a HTTP proxy between the upstream calls and the downstream monolith.

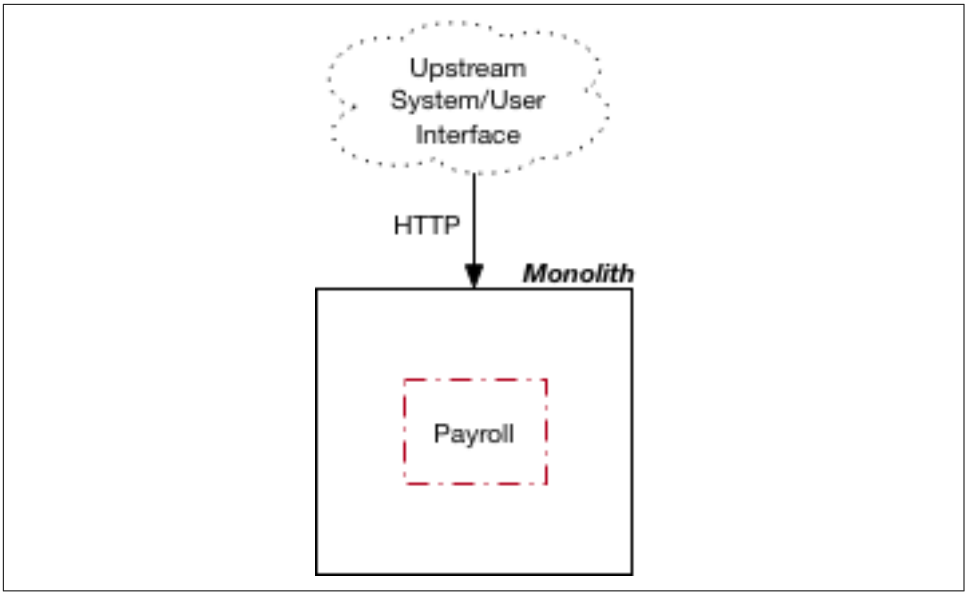


Figure 1-5. A simple overview of a HTTP-driven monolith prior to a strangler being implemented

Step 1: Insert Proxy

Unless you already have an appropriate HTTP proxy in place which you can reuse, I would suggest getting one in place *first*, as seen in Figure 1-6. In this first step, the proxy will just allow any calls to pass through without change.

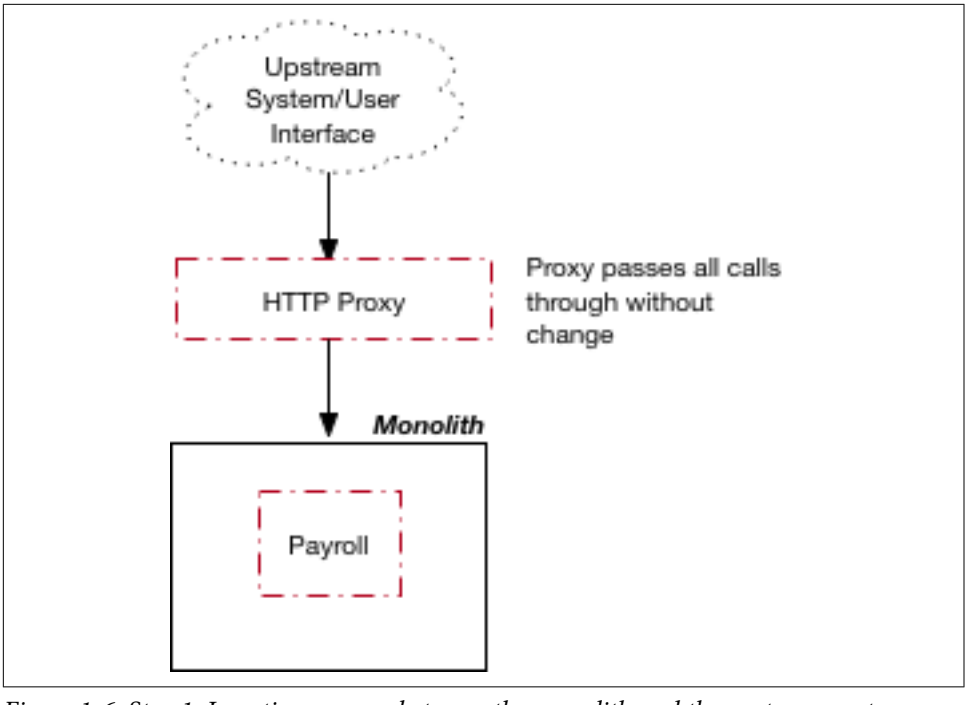


Figure 1-6. Step 1: Inserting a proxy between the monolith and the upstream system

This step will allow you to assess the impact of inserting an additional network hop between the upstream calls and the downstream monolith, setup any required monitoring of your new component, and basically, sit with it a while. From a latency point of view, we will be adding a network hop and a process in the processing path of all calls. With a decent proxy and network you'd expect a minimal impact on latency (perhaps in the order of a few milliseconds), but if this turns out not to be the case you have a chance to stop and investigate the issue before you go any further.

If you already have an existing proxy in place in front of your monolith, you can skip this step - although do make sure you understand how this proxy can be reconfigured to redirect calls later on. I'd suggest at the very least carry out some spikes to check the redirection will work as intended before assuming that this can be done later on. It would be a nasty surprise to discover that this is impossible just before you plan to send your new service live!

Step 2: Migrate Functionality

With our proxy in place, next you can start extracting your new microservice, as we see in Figure 1-7.

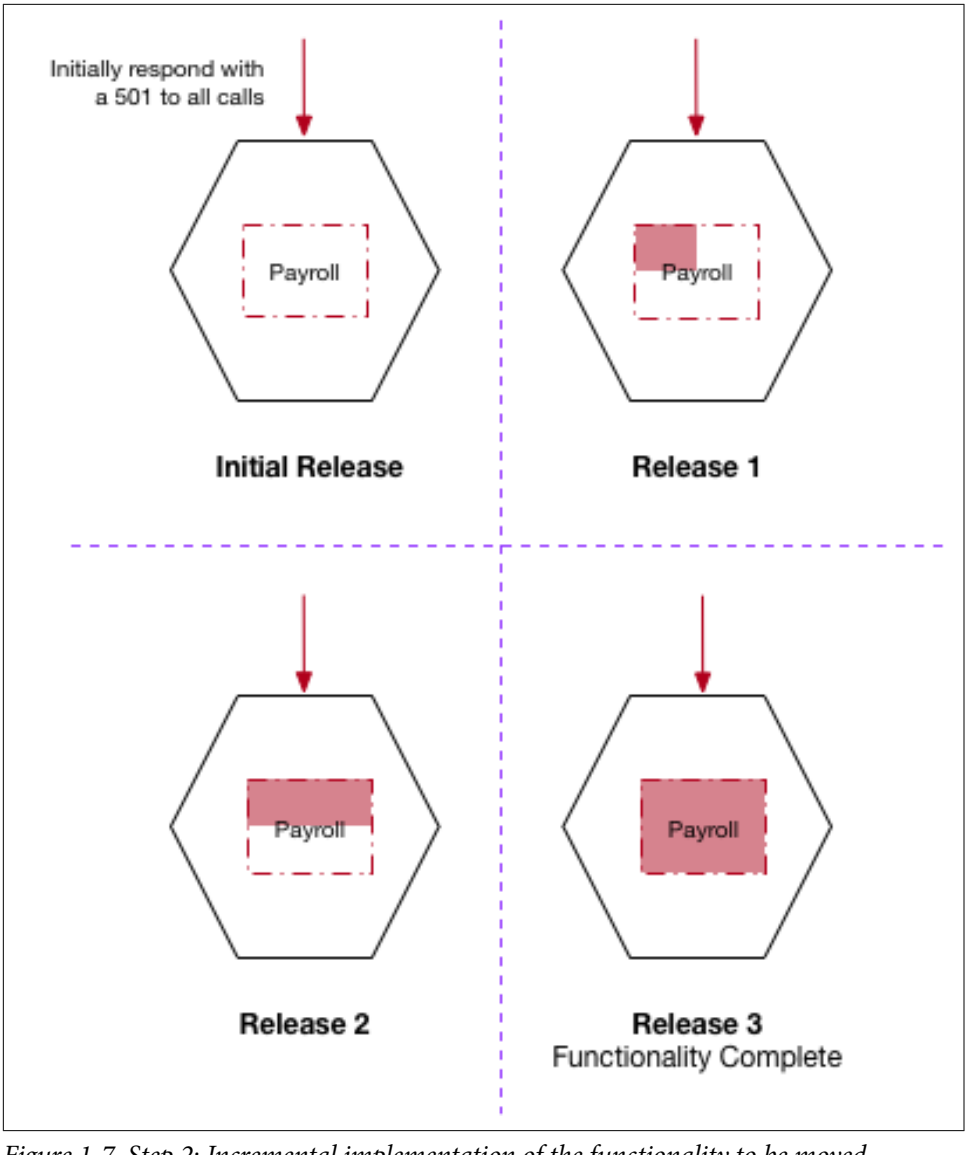


Figure 1-7. Step 2: Incremental implementation of the functionality to be moved

This step itself can be broken down into multiple stages. First, get a basic service up and running without any of the functionality being implemented. Your service will need to accept the calls made to the matching functionality, but at this stage you could just return a `501 Not Implemented`. Even at this step I'd get this service deployed into the production environment. This allows you to get comfortable with the production deployment process, and test the service in situ. At this point your

new service isn't *released*, as you haven't redirected the existing upstream calls yet. Effectively we are separating the step of deployment of the software from release of the software, a common release technique which we'll revisit later on.

Step 3: Redirect Calls

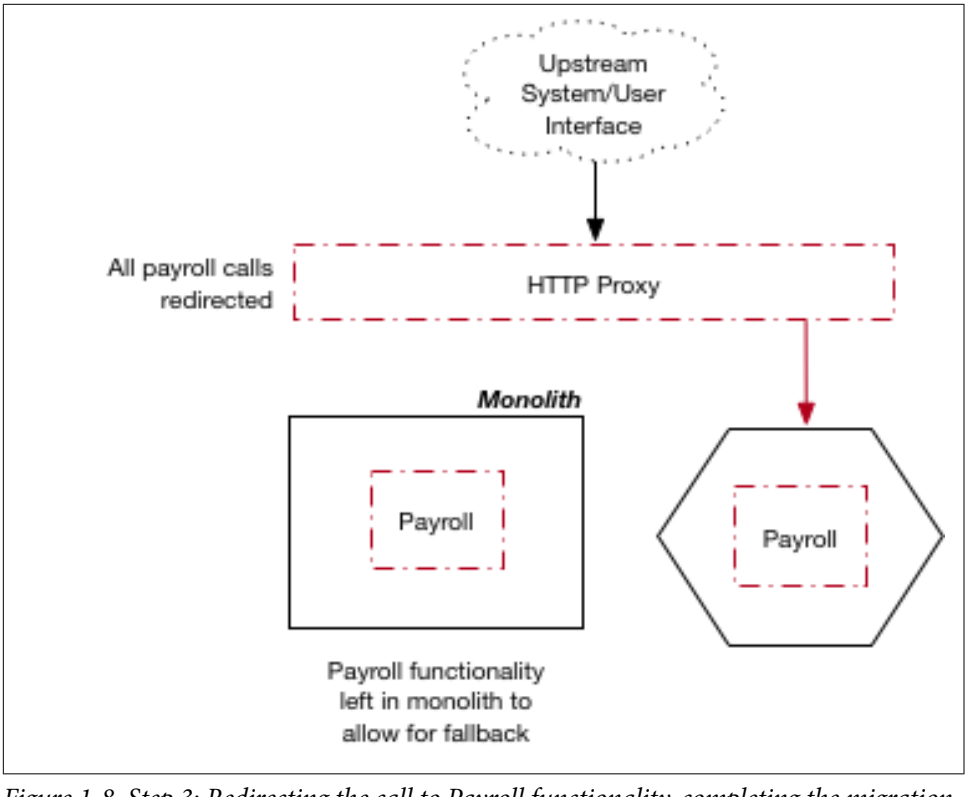


Figure 1-8. Step 3: Redirecting the call to Payroll functionality, completing the migration

It's only once you've completed movement of all the functionality that that you reconfigure the proxy to redirect the call, as we see in Figure 1-8. If this fails for whatever reason, then you can switch the redirection back - for most proxies this is a very quick and easy process, giving you a fast rollback.

Data?

So far, we haven't talked about data. In Figure 1-8 above, what happens if our newly migrated `Payroll` service needs access to data that is currently held in the monolith's database? We'll explore options for this more fully in Chapter 2.

Proxy Options

How you implement the proxy is in part going to be down to the protocol used by the monolith. If the existing monolith uses HTTP, then we're off to a good start. HTTP is such a widely supported protocol that you have a wealth of options out there for managing the redirection. I would probably opt for a dedicated proxy like `nginx` which has been created with exactly these sorts of use cases in mind, and can support a multitude of redirection mechanisms which are tried and tested and likely to perform fairly well.

Some redirections will be simpler than others. Consider redirection around URI paths, perhaps as would be exhibited making use of REST resources. In Figure 1-9 we move the entire `Invoice` resource over to our new service, and this is easy to parse from the URI path.

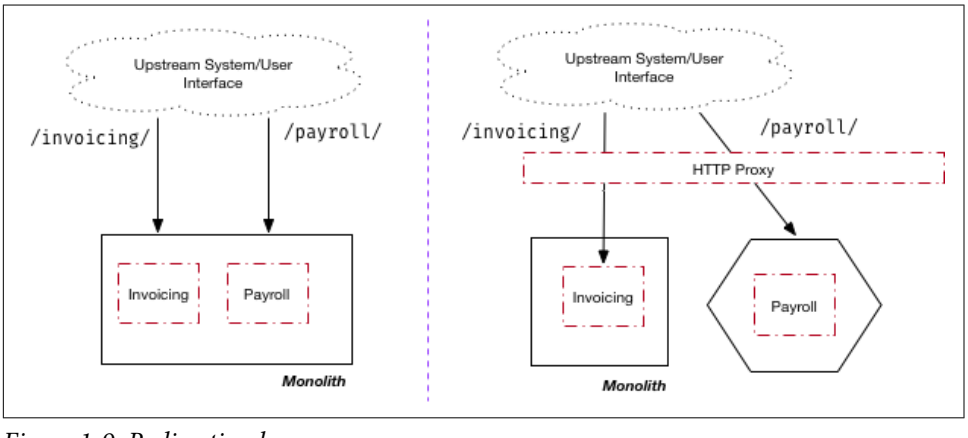


Figure 1-9. Redirection by resources

If however the existing system buries information about the nature of the functionality being called somewhere in the request body, perhaps in a form parameter, our redirection rule will need to be able to switch on a parameter in the POST - something which is possible, but more complicated. It is certainly worth checking the proxy options available to you to make sure they are able to handle this if you find yourself in this situation.

If the nature of interception and redirection is more complex, or in situations where the monolith is using a less well supported protocol, you might be tempted to code something yourself. I would just say be very cautious about this approach. I've written a couple of network proxies by hand before (one in Java, the other in Python), and whilst it may say more about my coding ability than anything else, in both situations the proxies were incredibly inefficient, adding significant lag into the system. Nowadays if I needed more custom behavior, I'd be more likely to consider adding

custom behavior to a dedicated proxy - for example `nginx` allows you to use code written in `lua` to add custom behavior.

Incremental Rollout

As you can see in Figure 1-10, this technique allows for architectural changes to be made via a series of small steps, each of which can be done alongside other work on the system.

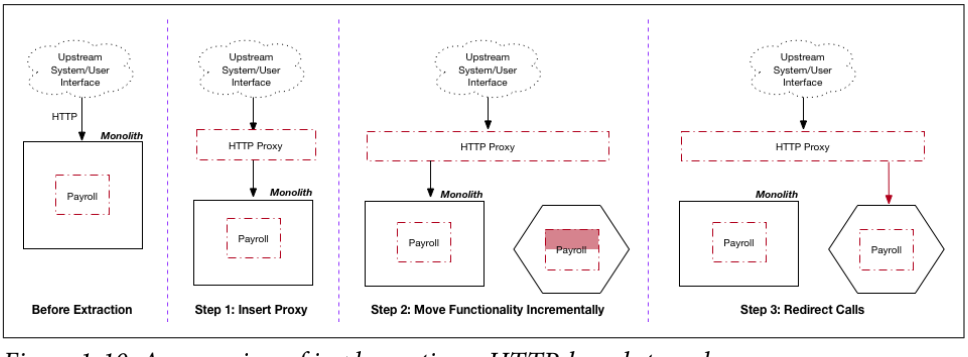


Figure 1-10. An overview of implementing a HTTP-based strangler

No big bang, stop the line re-platforming required. This makes it much easier to break this work down into stages that can be delivered alongside other delivery work. Rather than breaking your backlog down into "feature" and "technical" stories, fold all this work together - get good at making incremental changes to your architecture while still delivering new features!

Changing Protocols

You could also use your proxy to transform the protocol. For example you may currently expose a SOAP-based HTTP interface, but your new microservice is going to support a gRPC interface instead. You could then configure the proxy to transform requests and responses accordingly, as see in Figure 1-11.

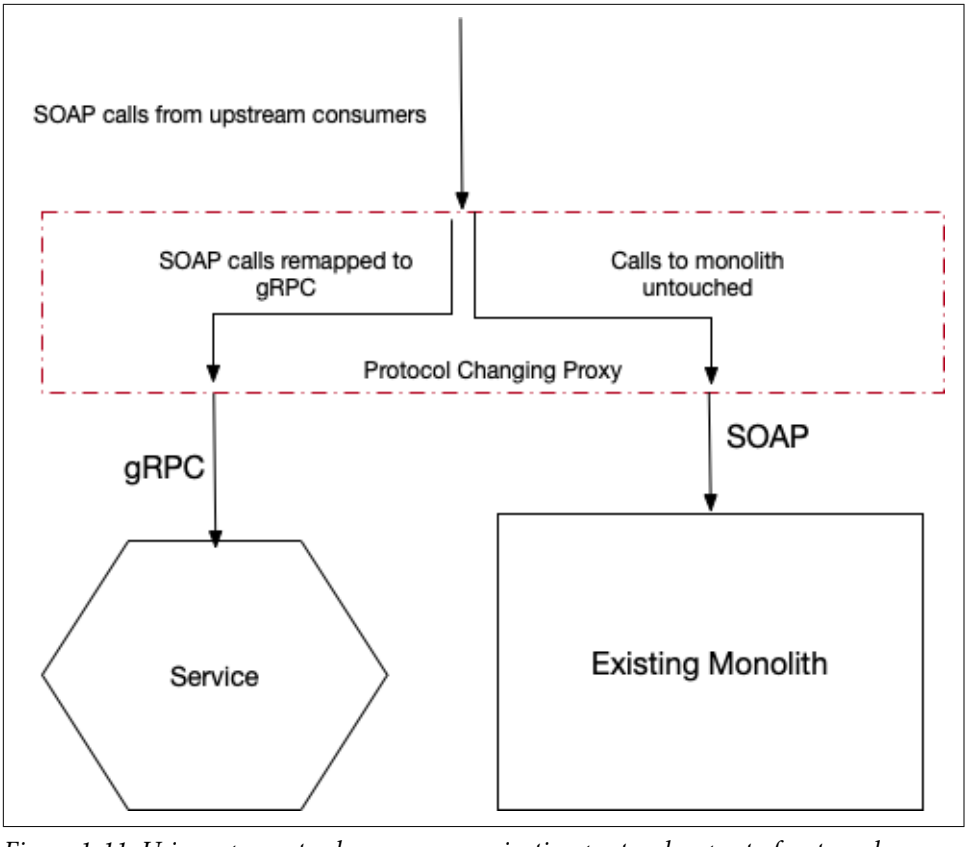


Figure 1-11. Using a proxy to change communication protocol as part of a strangler migration

I do have concerns about this approach, primarily due to the complexity and logic that starts to build up in the proxy itself. For a single service this doesn't look too bad, but if you start transforming the protocol for multiple services, the work being done in the proxy builds up and up. We're typically optimizing for independent deployability of our services, but if we have a shared proxy layer that multiple teams need to edit, this can slow down the process of making and deploying changes. We need to be careful that we aren't just adding a new source of contention.

If you want to migrate the protocol being used, I'd much rather push the mapping into the service itself - with the service supporting both your old communication protocol and the new protocol. Inside the service, calls to our old protocol could just get remapped internally to the new communication protocol, as we see in Figure 1-12.

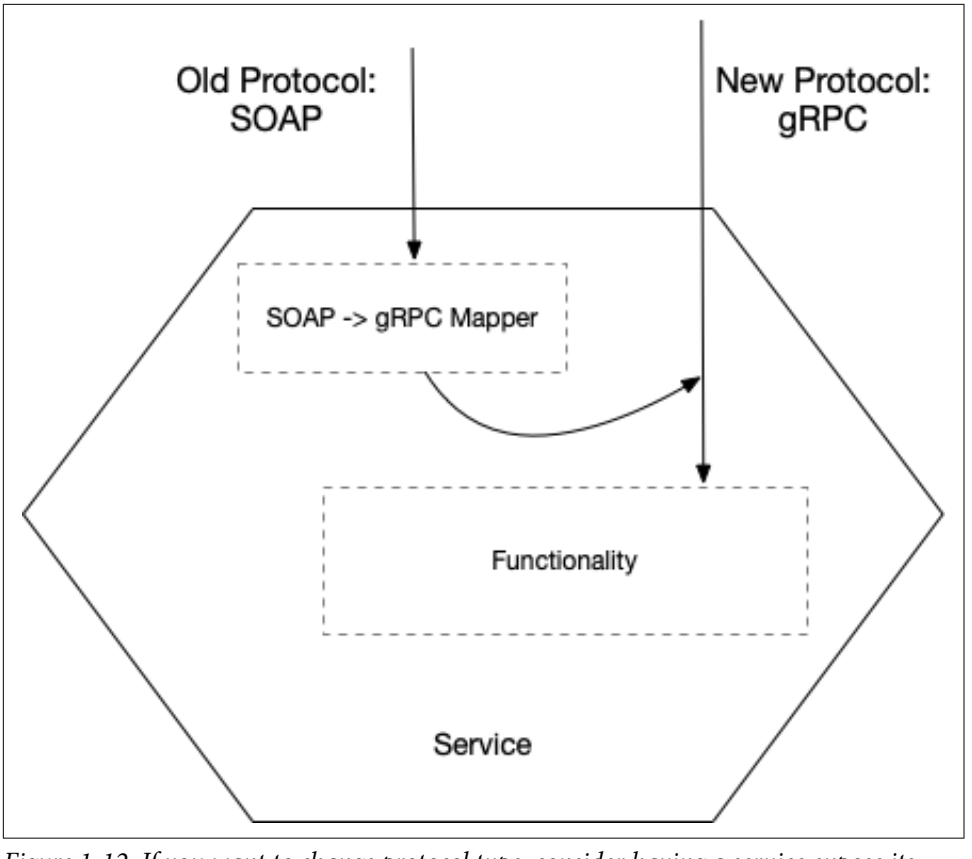


Figure 1-12. If you want to change protocol type, consider having a service expose its capabilities over multiple types of protocol

By pushing service-specific request and response mapping inside the service, this keeps the proxy layer simple and much more generic. Additionally, by a service providing both types of endpoints, you give yourself time to migrate upstream consumers before potentially retiring the old API.

Example: FTP

Although I've spoken at length regarding the use of the strangler pattern for HTTP-based systems, there is nothing to stop you intercepting and redirecting other forms of communication protocol. A European Real Estate company used a variation of this pattern to change how customers uploaded new real estate listings.

The company's customers uploaded listings via FTP, with an existing monolithic system handling the uploaded files. The company were keen to move over to microservices, and also wanted to start to support a new upload mechanism which rather than

supporting batch FTP upload, instead was going to use a REST API which matched a soon to be ratified standard.

The real estate company didn't want to have to change things from the customer point of view - they wanted to make any changes seamless. This means that FTP still needed to be the mechanism by which customers interacted with the system, at least for the moment. In the end, they intercepted FTP uploads (by detecting changes in the FTP server log), and directed newly uploaded files to an adapter that converted the uploaded files into requests to the new REST API, as shown in Figure 1-13

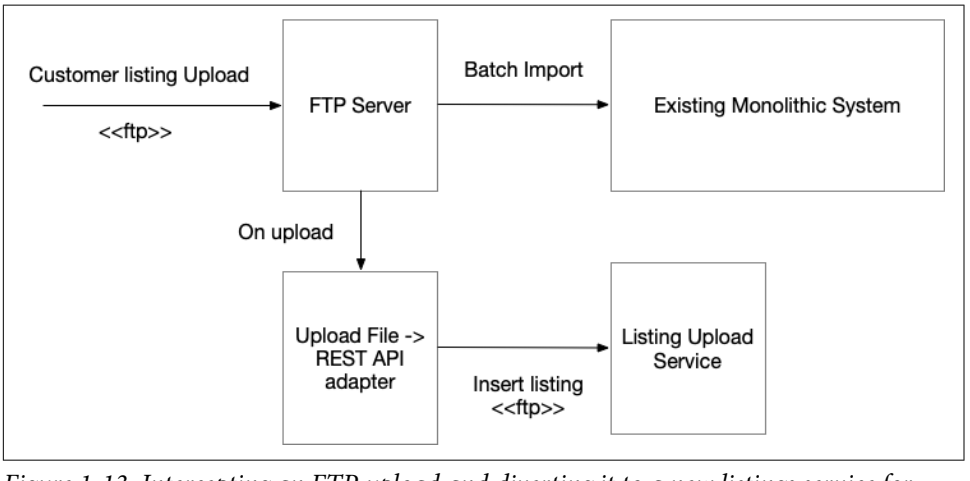


Figure 1-13. Intercepting an FTP upload and diverting it to a new listings service for XXX

From a customer point of view, the upload process itself didn't change. The benefit came from the fact that the new service which handled the upload was able to publish the new data much more quickly, helping customers get their adverts live much faster. Later on, there is a plan to directly expose the new REST API to customers. Interestingly, during this period both listing upload mechanisms were enabled. This allowed the team to make sure the two upload mechanisms were working appropriately. This is a great example of a pattern we'll explore later in "Pattern: Parallel Running" on page 50.

Example: Message Interception

So far we've looked at intercepting synchronous calls, but what if your monolith is driven from some other form of protocol, perhaps receiving messages via a message broker? The fundamental pattern here is the same - we need a method to intercept the calls, and to redirect them to our new microservice. The main difference here is the nature of the protocol itself.

Content-based routing

In Figure 1-14, our monolith receives a number of messages, a subset of which we need to intercept.

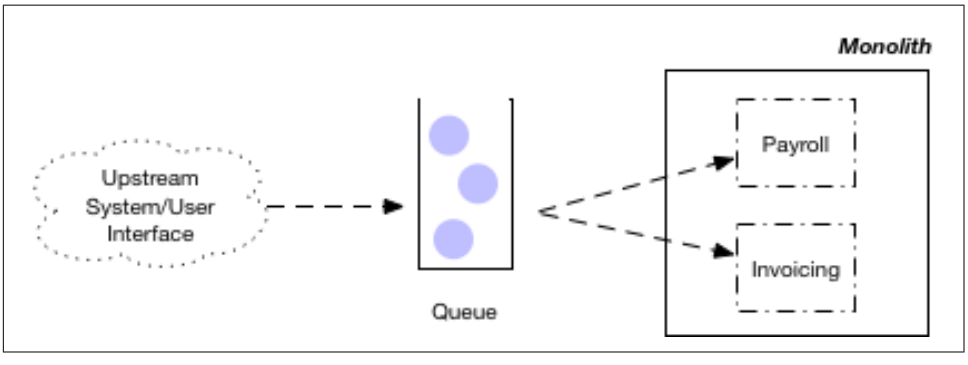


Figure 1-14. A monolith receiving calls via a queue

A simple approach would be to intercept **all** messages intended for the downstream monolith, and filter the messages on to the appropriate location, as outlined in Figure 1-15. This is basically an implementation of the Content-based Router pattern, as described in *Enterprise Integration Patterns* (Addison-Wesley)[3].

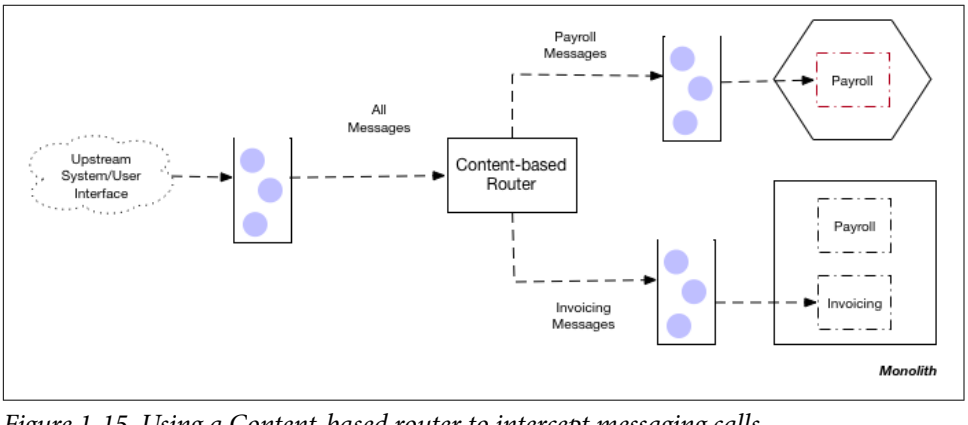


Figure 1-15. Using a Content-based router to intercept messaging calls

This technique allows us to leave the monolith untouched, but we're placing another queue on our request path, which could add additional latency and is another thing we need to manage. The other concern is how many "smarts" are we placing into our

3 See Page 230 of Enterprise Integration Patterns by Bobby Woolf and Gregor Hohpe.

messaging layer. In Building Microservices[4] I spoke about the challenges caused by systems making use of too many smarts in the networks between your services, as this can make systems harder to understand and harder to change. Instead I urged you to embrace the mantra of "Smart endpoints, dumb pipes", something which I still push for. It's arguable here that the content-based router is us implementing a "smart pipe" - adding additional complexity in terms of how calls are routed between our systems. In some situations this is a very useful technique - but it's up to find a happy balance.

Selective Consumption

An alternative would be to change the monolith and have it ignore messages sent which should be received by our new service, as we see in Figure 1-16. Here, we have both our new service and our monolith share the same queue, and locally they use some sort of pattern matching process to listen to the messages they care about. This filtering process is very common, and can be implemented using something like a `Message Selector` in JMS or using equivalent technology on other platforms.

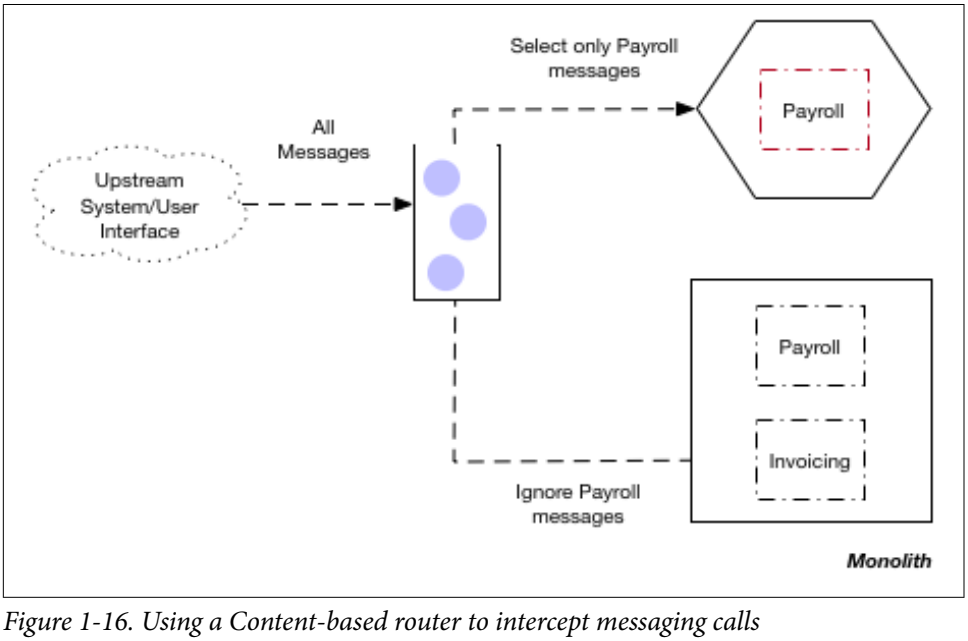


Figure 1-16. Using a Content-based router to intercept messaging calls

This filtering approach reduces the need to create an additional queue, but has a couple of challenges. Firstly, your underlying messaging technology may or may not let you share a single queue subscription like this (although this is a very common fea-

4 See Chapter 4 Building Microservices, Sam Newman, O'Reilly Media 2015

ture, so I would be surprised if this was the case). Secondly, when you want to redirect the calls, it requires two changes to be fairly well coordinated. I need to stop my monolith from reading the calls meant for the new service, and then have the service pick them up. Likewise, reverting the call interception requires two changes to roll back.

The more different types of consumers you have for the same queue, and the more complex the filtering rules, the more problematic things can become. It can be easy to imagine a situation where two different consumers both start receiving the same message due to overlapping rules, or even the opposite - where some messages are being ignored altogether. For this reason I would likely only consider using selective consumption with a small number of consumers &/or with a very simple set of filtering rules. A content-based routing approach is likely to make more sense as the number of different types of consumers increase, although beware of the potential downsides cited above, especially falling into the "smart pipes" problem.

The added complication with either this solution or with Content-Based Routing is that if we are using an asynchronous request-response style of communication, we'll need to make sure we can route the request back to the client, hopefully without them realising anything has changed. There are a number of other options for call routing in message-driven systems, many of which can help you implement strangler pattern migrations. I can thoroughly recommend *Enterprise Integration Patterns* (Addison-Wesley) as a great resource here.

Other Protocols

As I hope you can understand from this example there are lots of different ways to intercept calls into your existing monolith, even if you use different types of protocols. What if your monolith is driven by a batch file upload? Intercept the batch file, extract the calls that you want to intercept, and remove them from the file before you forward it on. True, some mechanisms make this process more complicated, and it's much easier if using something like HTTP, but with some creative thinking the strangler pattern can be used in a surprising number of situations.

Other Examples Of The Strangler Pattern

The Strangler Pattern is highly useful in any context where you are looking to incrementally re-platform an existing system, and it's use isn't limited to teams implementing microservice architectures. The pattern has been in use for a long time before Martin Fowler wrote it up in 2004. At my previous employer, ThoughtWorks, we often used it to help rebuild monolithic applications. Paul Hammant has authored a collated a non-exhaustive list of projects where we used this pattern over on his blog. They include a trading company's blotter, an airline booking application, a rail ticketing system and a classified ad portal.

Pattern: UI Composition

With the techniques we've considered so far, we've primarily pushed the work of incremental migration to the server - but the user interface presents us with some very useful opportunities to splice together functionality served in part from an existing monolith or new microservice architecture.

Many years ago I was involved in helping move the Guardian online from their existing content management system over to a new, custom Java-based platform. This was going to coincide with the rollout of a whole new look and feel for the online newspaper to tie in with the relaunch of the print edition. As we wanted to embrace an incremental approach, the cut-over from the existing CMS to the new website served from the brand new was phased in parts, targeting specific verticals (travel, news, culture etc). Even within those verticals, we also looked for opportunities to break down the migration into smaller chunks.

We ended up using two different compositional techniques that ended up being very useful. From speaking to other companies it's become clear to me over the past several years that variations on these techniques are a significant part of how many organizations adopt microservice architectures.

Example: Page Composition

With the Guardian, although we started by rolling out a single widget (which we'll discuss shortly), the plan had always been to mostly use a page-based migration in order to allow a brand new look and feel to go live. This was done on a vertical-by-vertical basis, with Travel being the first we sent live. Visitors to the Guardian website during this transition time would have been presented with a very different look and feel when they went to the new parts of the site. Great pains were also taken to ensure that all old page links were redirected to the new locations (where URLs had changed).

When the Guardian made another change in technology, moving away from the Java monolith some years later, they again used a similar technique of migrating a vertical at a time. At this point they made use of the Fastly Content Delivery Network (CDN) to implement the new routing rules, effectively using the CDN much like you might use an in-house proxy[5].

REA Group in Australia, which provides online listings for real estate, have different teams are responsible for commercial or residential listings, owning those whole

5 It was nice to hear from Graham Tackley at the Guardian that the "new" system I initially helped implemented lasted almost ten years before being entirely replaced with the current architecture. As a reader of the Guardian online I reflected that I never really noticed anything changing during this period!

channels. In such a situation, a page based composition approach makes sense, as a team can own the whole end-to-end experience. REA actually employ subtly different branding for the different channels, which means that page-based decomposition make even more sense as you can deliver quite different experiences to different customer groups.

Example: Widget Composition

At the Guardian, the travel vertical was the first one identified to be migrated to the new platform. The rationale was partly that it had some interesting challenges around categorization, but also that it wasn't the most high-profile part of the site. Basically, we were looking to get something live, learn from that experience, but also make sure that if something did go wrong then it wouldn't affect the prime parts of the site.

Rather than go live with the whole travel part of the website, replete with in-depth reportage of glamorous destinations all over the world, we wanted a much more low-key release to test out the system. Instead, we deployed a single widget displaying the top ten travel destinations, defined using the new system. This widget was spliced into the old Guardian travel pages, as show in Figure 1-17. In our case we made use of a technique called Edge-side Includes (ESI), using Apache. With ESI, you define a template in your webpage, and a web server splices in this content.

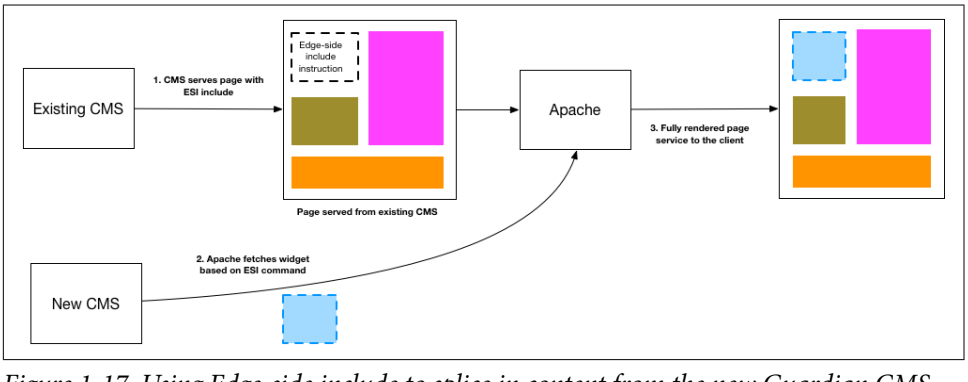


Figure 1-17. Using Edge-side include to splice in content from the new Guardian CMS

Nowadays, splicing in a widget using a technique like ESI is less common. This is largely as browser-based technology has become more sophisticated, allowing for much more composition to be done in the browser itself (or in the native app - more on that later). This means for widget-based web UIs, the browser itself is often making multiple calls to load various different widgets using a multitude of different techniques. This has the further benefit that if one widget fails to load - perhaps because the backing service is unavailable - the other widgets can still be rendered allowing for only a partial, rather than total degradation of service.

Although in the end we mostly used Page-based composition at the Guardian, many other companies make heavy use of widget-based composition with supporting back-end services. Orbitz (now part of Expeida) for example created dedicated services just to serve up a single widget[6]. Prior to their move to microservices, the Orbitz website was already broken up into separate UI "modules" (in Orbitz nomeclature). These modules could represent a search form, booking form, map etc. These UI modules were initially servied directly from the Content Orchestration Service, as we see in Figure 1-18.

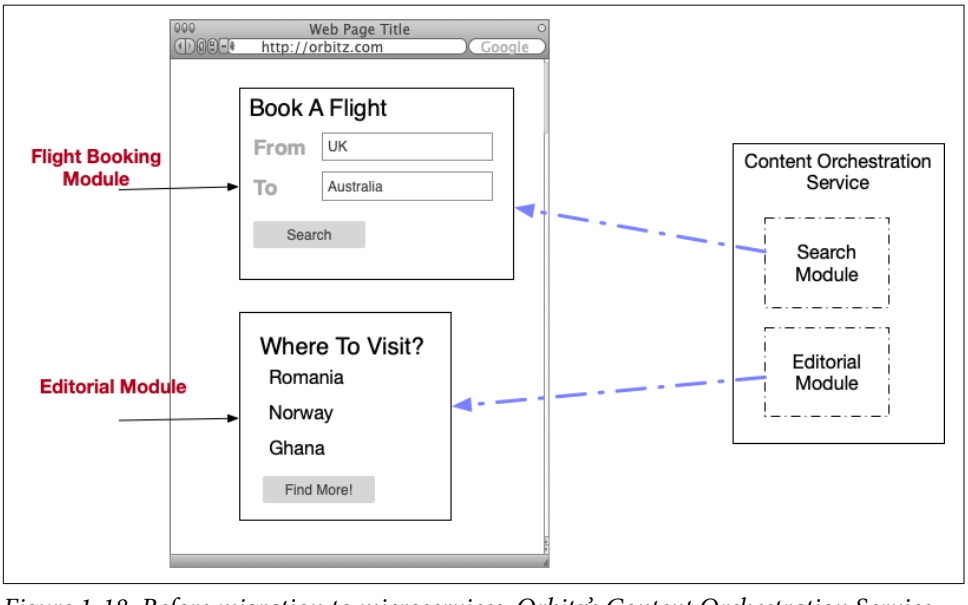


Figure 1-18. Before migration to microservices, Orbitz's Content Orchestration Service served up all modules

The Content Orcherstration Service was in effect a large monolith. Teams that owned these modules all had to co-ordinate changes being made inside the monolith, causing significant delays in rolling out changes. As part of a drive towards faster release cycles, when Orbitz decided to try microservices they focused their decomposition along these module lines - starting with the editorial module. The Content Orchestration Service was changed to delegate responsibility for transitioned modules to downstream services, as we see in Figure 1-19.

6 GOTO 2016 • Enabling Microservices at Orbitz • Steve Hoffman & Rick Fast *https://www.youtube.com/watch?v=gY7JSjWDFJc*

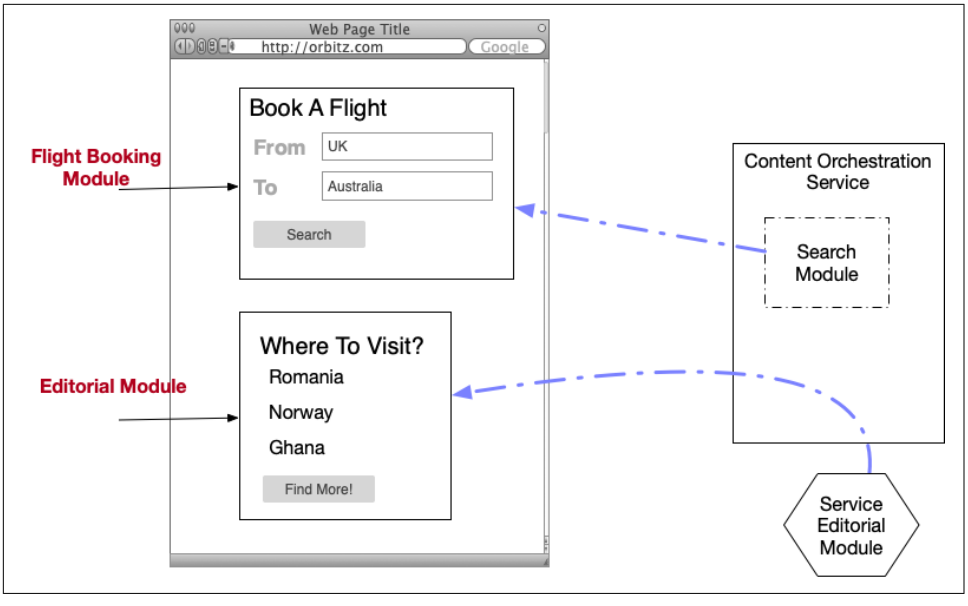


Figure 1-19. Modules were migrated one at a time, with the Content Orchestration Service delegating to the new backing services

The fact that the UI was already decomposed visually along these lines made this work easier to do in an incremental fashion. This transition was further helped as the separate modules already had clean lines of ownership, making it easier to perform migrations without interfering with other teams.

It's worth noting that not all user interfaces suit decomposition into clean widgets - but when they can, it makes the work of incremental migration to microservices much easier.

And Mobile Applications

While we've spoken primarily about web-based UIs, some of these techniques can work well for mobile based clients too. Both Android and iOS for example provide the ability to componentize parts of their user interfaces, making it easier for these parts of the UI to be worked on in isolation, or recombined in different ways.

One of the challenges with deploying changes with native mobile applications is that both the Apple App store and Android Play store require applications to be submitted and vetted prior to new versions being made available. While the times for applications to be signed off by the app stores have in general reduced significantly over the last several years, this still adds time before new releases of software can be deployed. The app itself is also at this point a monolith - if you want to change one single part of a native mobile application, the whole application still needs to be

deployed. You also have to consider the fact that users have to download the new app to see the new features - something you typically don't have to deal with when using a web-based application as changes are seamlessly delivered to the users' browser.

Many organisations have dealt with this by allowing them to make dynamic changes to an existing native mobile application without having to resort to deploying a new version of a native application. By making changes on the server-side, changes can be seen immediately on client devices. This can be achieved very simply using things like embedded web views, although some companies use more sophisticated techniques.

Spotify's UI across all platforms is heavily component-oriented, including their iOS and Android applications - pretty much everything you see is a separate component from a simple text header, to album artwork, or a playlist[7]. Some of these modules are in turn backed by one or more microservices. The configuration and layout of these UI components is defined in a declarative fashion on the serverside - Spotify engineers are able to change the views that users see and roll that change very quickly, without needing to submit new versions of their application to the app store. This allows them to much more rapidly experiment and try out new features.

Example: Micro Frontends

As bandwidth and the capability of web browsers have improved, so has the sophistication of the code running in browsers improved. Many web-based user interfaces now make use of some form of Single Page Application framework, which does away with the concept of an application consisting of different webpages, instead giving you more powerful user interface were everything runs in a single pane - effectively in-browser user experiences that previously were only available to those of us working with "thick" UI SDKs like Java's Swing.

By delivering an entire interface in a single page, we obviously can't consider page-based composition, so have to consider some form of widget-based composition. Attempts have been made to codify common widget formats for the web - most recently the Web Components specification is attempting to define a standard component model supported across browsers. It has taken a long while though for this standard to gain any traction, with browser support amongst other things being a considerable stumbling block.

People making use of Single Page App frameworks like Vue, Angular or React haven't sat around though waiting for Web Components to solve their problems. Instead many people have tried to tackle the problem of how to modularize UIs built with

7 *https://www.youtube.com/watch?v=vuCfKjOwZdU* UMT2016 - John Sundell - Building component-driven UIs at Spotify

SDKs that were initially designed to own the whole browser pane - this has lead to the push towards what some people have called Micro Frontends.

At first glance, Micro Frontends really are just about breaking down a user interface into different components that can be worked on independently. In that, they are nothing new - component-oriented software pre-dates by birth by several years! What is more interesting is that people are working out how to make web browsers, SPA SDKs and componentization work together. How exactly do you create a single UI out of bits of Vue and React without having their dependencies clash, but still allow them to potentially share information?

Covering this topic in depth is out of scope for this book, partly because the exact way you make this work will vary based on the each SPA frameworks being used. But if you find yourself with a single page application that you want to break apart, you're not alone, and there are many people out there sharing different techniques and libraries to make this work.

Where To Use It

UI composition as a technique to allow for re-platforming systems is highly effective, as it allows for whole vertical slices of functionality to be migrated. For it to work though, you need to have the ability to change the existing user interface to allow for new functionality to be safely inserted. We'll cover different compositional techniques later on in the book, but it's worth noting that which techniques you can use will often depend on the nature of the technology used to implement the user interface. A good old fashioned website makes UI composition very easy, whereas single page app technology does add some complexity, and an often bewildering array of implementation approaches!

Pattern: Branch By Abstraction

For the very useful strangler pattern to work, we need to be able to intercept calls at the perimeter of our monolith. However, what happens if the functionality we want to extract is deeper inside our existing system? Coming back to a previous example, consider the desire to extract the notification functionality, as seen in Figure 1-4.

In order to perform this extraction, we will need to make changes to the existing system. These changes could be significant, and disruptive to other developers working on the codebase at the same time. We have competing tensions here. On the one hand we want to make our changes in incremental steps. On the other-hand, we want to reduce the disruption to other people working on other areas of the codebase - this will naturally drive us towards wanting to complete the work quickly.

Often, when reworking parts of an existing codebase, people will do that work on a separate source code branch. This allows the changes to be made without disrupting

the work of other developers. The challenge is that once the change in the branch has been completed, these changes have to be merged back, which can often cause significant challenges. The longer the branch exists, the bigger these problems are. I won't go into detail now as to the problems associated with long-lived source code branches, other than to say it runs contrary to the principles of Continuous Integration. I could also throw in that data gathered from the State Of DevOps Report shows that embracing trunk-based development (where changes are made directly on mainline and branches are avoided) and use short-lived branches contributes to higher performance of IT teams[8]. Let's just say that I am not a fan of long-lived branches, and I'm not alone.

So, we want to be able to make changes to our codebase in an incremental fashion, but also keep disruption to a minimum for developers working on other parts of our codebase. There is another pattern we can use that allows us to incrementally make changes to our monolith without resorting to source code branching. The Branch By Abstraction pattern instead relies on making changes to the existing codebase to allow the implementations to safely co-exist alongside each other, in the same version of code, without causing too much disruption.

How It Works

Branch by abstraction consists of five steps:

1. Create an abstraction for the functionality to be replaced
2. Change clients of the existing functionality to use the new abstraction
3. Create a new implementation of the abstraction with the reworked functionality. In our case this new implementation will call out to our new microservice
4. Switch over the abstraction to use our new implementation
5. Clean up the abstraction and remove the old implementation

Let's take a look at these steps with respect to moving our `Notification` functionality out into a service, as detailed in Figure 1-4.

Step 1: Create Abstraction

The first task is to create an abstraction that represents the interactions between the code to be changed, and the callers of that code, as we see in Figure 1-20. If the existing `Notification` functionality is reasonably well factored, this could be as simple as applying an `Extract Interface` refactoring in our IDE. If not, you may need to extract a seam, as mentioned above.

8 See the 2017 State Of DevOps Report: *https://puppet.com/resources/whitepaper/2017-state-of-devops-report*

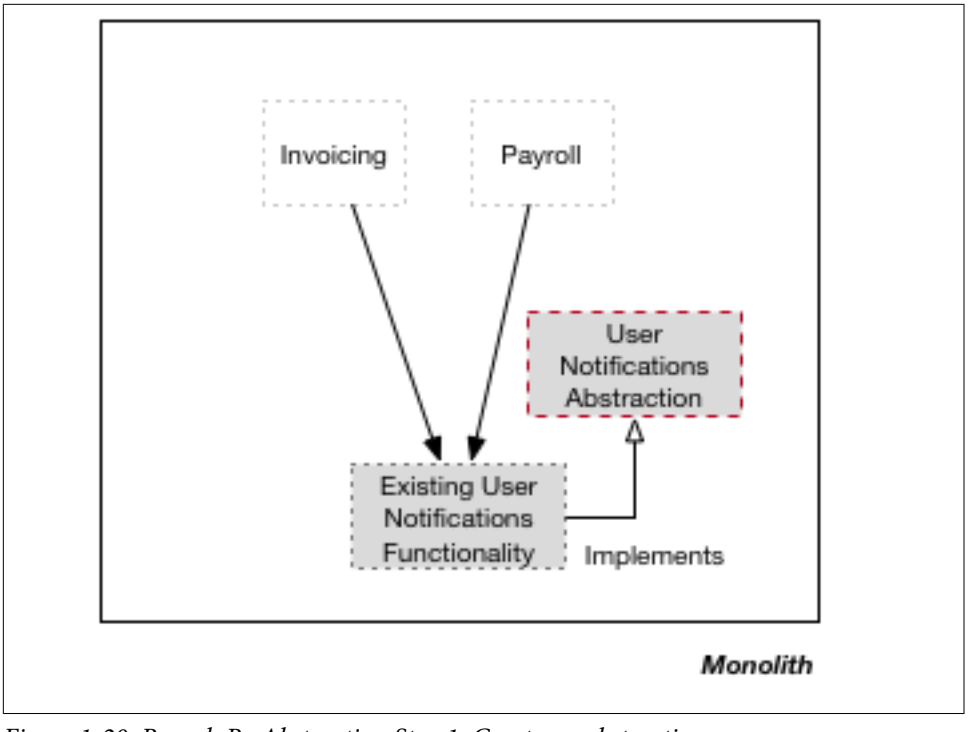


Figure 1-20. Branch By Abstraction Step 1: Create an abstraction

Step 2: Use Abstraction

With our abstraction created, we now need to refactor the existing clients of the `Notification` functionality to use this new abstraction point as we see in Figure 1-21. It's possible that an `Extract Interface` refactoring could have done this for us automatically - but in my experience it's more common that this will need to be an incremental process, involving manually tracking inbound calls to the functionality in question. The nice thing here is that these changes are small and incremental - they're easy to make in small steps without making too much impact on the existing code. At this point, there should be no functional change in system behavior.

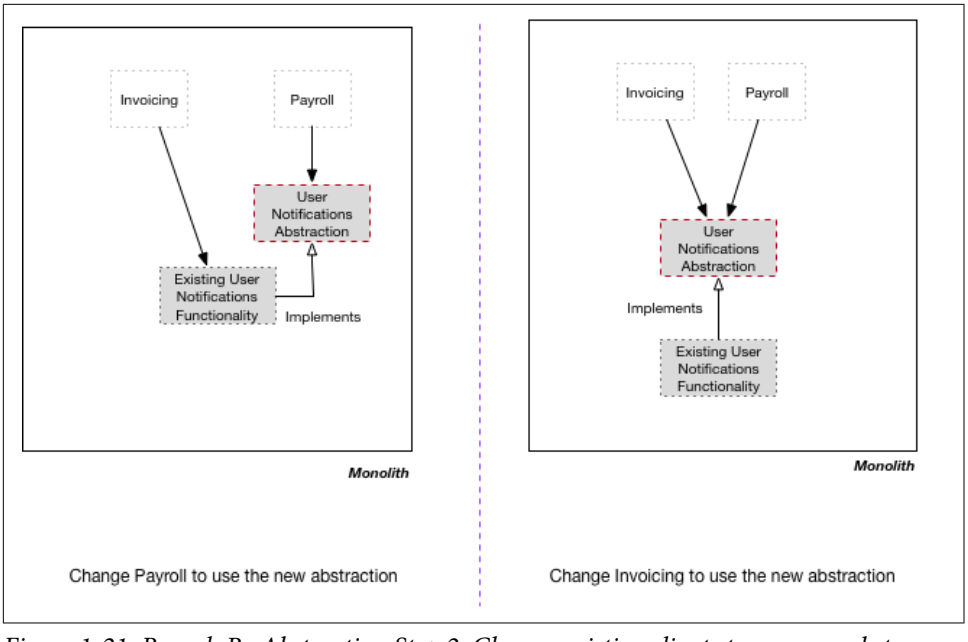


Figure 1-21. Branch By Abstraction Step 2: Change existing clients to use new abstraction

Step 3: Create new implementation

With our new abstraction in place, we can now start work on our new service calling implementation. Inside the monolith, our implementation of the `Notifications` functionality will mostly just be a client calling out to the external service as in Figure 1-22 - the bulk of the functionality will be in the service itself.

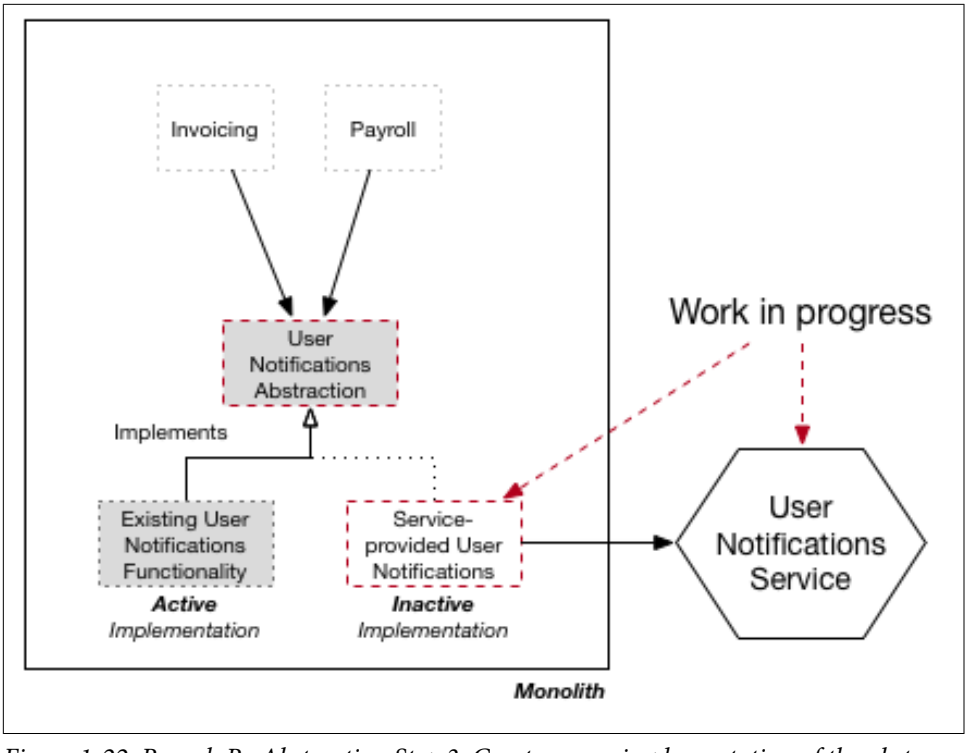


Figure 1-22. Branch By Abstraction Step 3: Create a new implementation of the abstraction

The key thing to understand at this point is that although we have two implementations of the abstraction in the codebase at the same time, it's only one implementation that is currently active in the system. Until we're happy that our new service calling implementation is ready to send live, it is in effect dormant. While we work to implement all the equivalent functionality in our new service, our new implementation of the abstraction could return `Not Implemented` errors. This doesn't stop us writing tests for the functionality we have written of course, and this is in fact one of the benefits of getting this work integrated as early as possible.

During this process, we can also deploy our work-in-progress `User Notification Service` into production, just as we did with the strangler pattern. The fact that it isn't finished is fine - at this point as our implementation of the `Notifications` abstraction isn't live, the service isn't actually being called. But we can deploy it, test it in situ, verify the parts of the functionality we have implemented are working correctly.

This phase could last a significant amount of time. Jez Humble details[9] the use of the Branch By Abstraction pattern to migrate the database persistence layer used in the Continuous Delivery application GoCD (at the time called Cruise). The switch from using `iBatis` to `Hibernate` lasted several months - during which, the application was still being shipped to clients on a twice weekly basis.

Step 4: Switch implementation

Once we are happy that our new implementation is working correctly, we switch our abstraction point so that our new implementation is active, and the old functionality is no longer being used, as seen in Figure 1-23.

9 *https://continuousdelivery.com/2011/05/make-large-scale-changes-incrementally-with-branch-by-abstraction/*

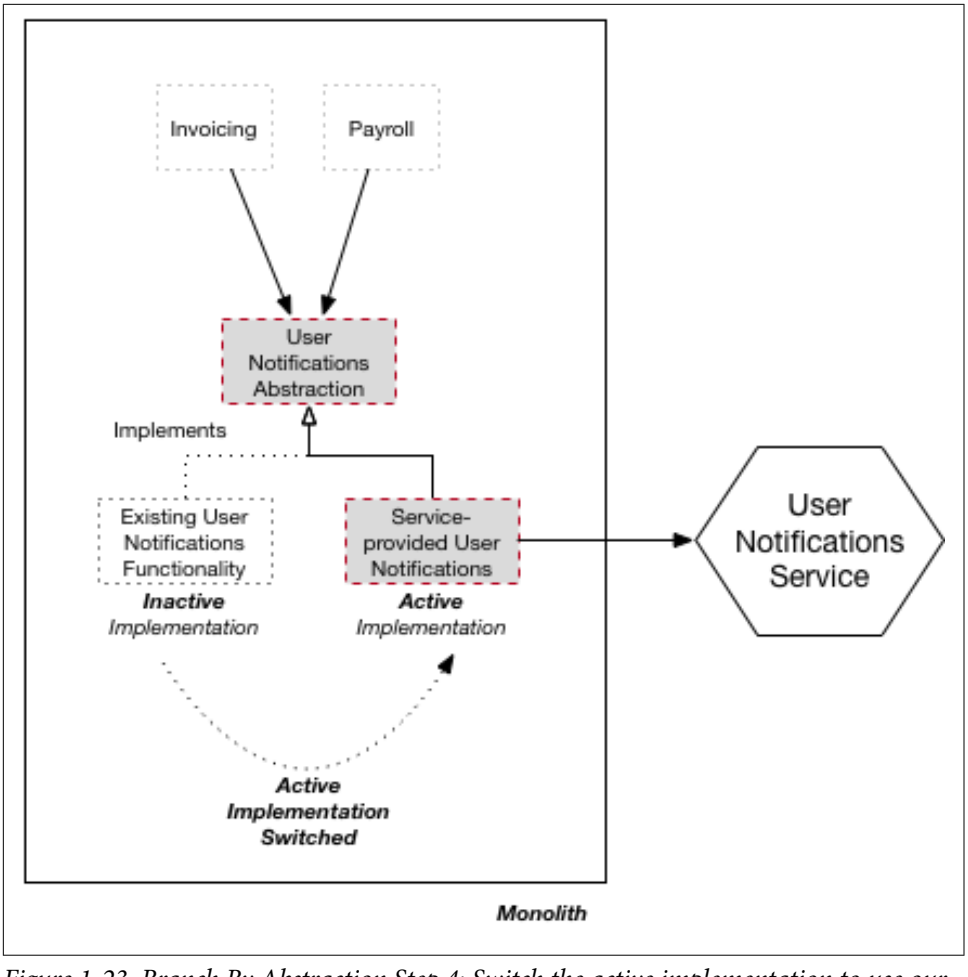


Figure 1-23. Branch By Abstraction Step 4: Switch the active implementation to use our new microservice

Ideally, as with the Strangler Pattern above, we'd want to use a switching mechanism that can be toggled easily. This allows us to quickly switch back to the old functionality if we found a problem with it. A common solution to this would be to use Feature Toggles. In Figure 1-24 we see toggles being implemented using a configuration file, allowing us to change the implementation being used without having to change code. If you want to know more about Feature Toggles and how to implement them, then Pete Hodgson has an excellent writeup[10].

10 *https://martinfowler.com/articles/feature-toggles.html*

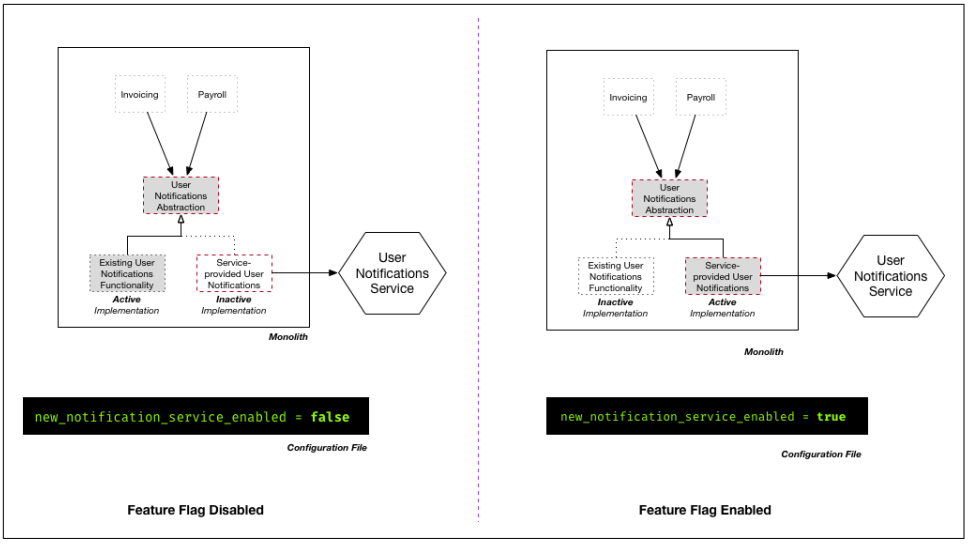


Figure 1-24. Branch By Abstraction Step 4: Using feature toggles to switch between different implementations

At this stage, we have two implementations of the same abstraction, which we *hope* should be functionality equivalent. We can use tests to verify equivalency, but we also have the option here to use *both* implementations in production to provide additional verification. This idea is explored further in "Pattern: Parallel Running" on page 50.

Step 5: Cleanup

With our new microservice now providing all notifications for our users, we can turn our attention to cleaning up after ourselves. At this point, our old `User Notifications` functionality is no longer being used, so an obvious step would be to remove it, as shown in Figure 1-25. We are starting to shrink the monolith!

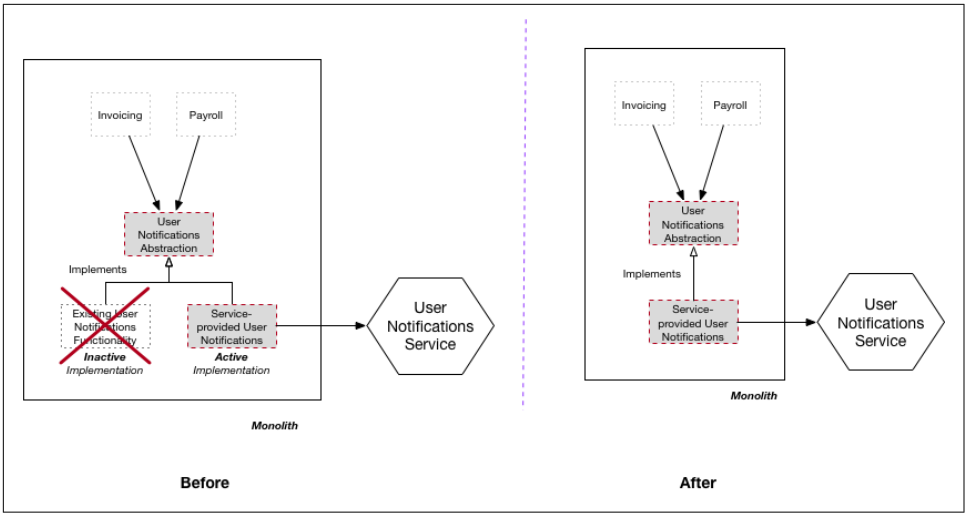


Figure 1-25. Branch By Abstraction Step 5: Remove the old implementation

When removing the old implementation, it would also make sense to remove any feature flag switching we may have implemented. One of the real problems associated with the use of feature flags is leaving old ones lying around - don't do that! Remove flags you don't need any more to keep things simple.

Finally, with the old implementation gone, we have the option of removing the abstraction point itself, as in Figure 1-26. It's possible however that the creation of the abstraction may have improved the codebase to the point where you'd rather keep it in place. If it's something as simple as an Interface, retaining it will have minimal impact on the existing codebase.

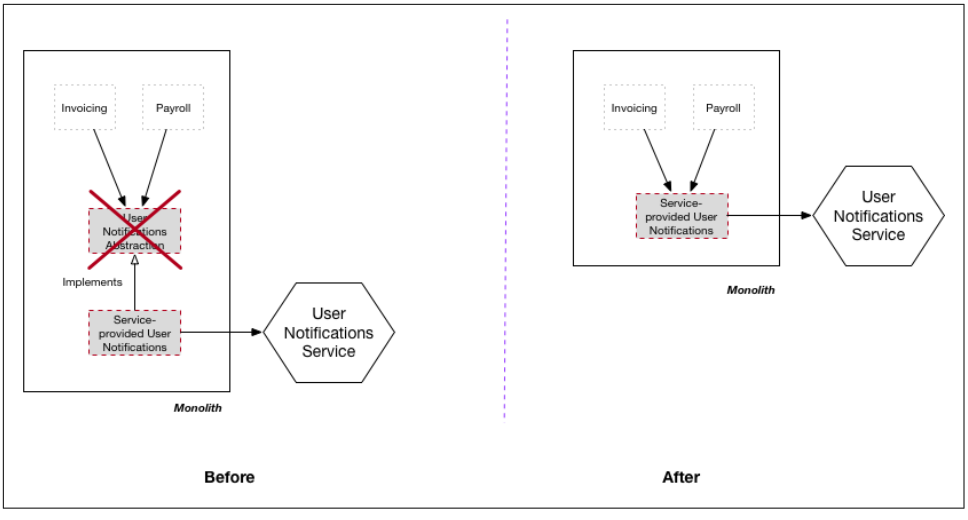


Figure 1-26. Branch By Abstraction Step 5: (Optional) Remove the abstraction point

As A Fallback Mechanism

The ability for us to switch back to our previous implementation if our new service isn't behaving is useful, but is there a way we could do that automatically? Steve Smith details[11] a variation of the branch by abstraction pattern called Verify Branch By Abstraction that also implements a live verification step - we can see an example of this in Figure 1-27. The idea is that if calls to the new implementation fail, then the old implementation could be used instead.

11 *https://www.continuousdeliveryconsulting.com/blog/application-pattern-verify-branch-by-abstraction/*

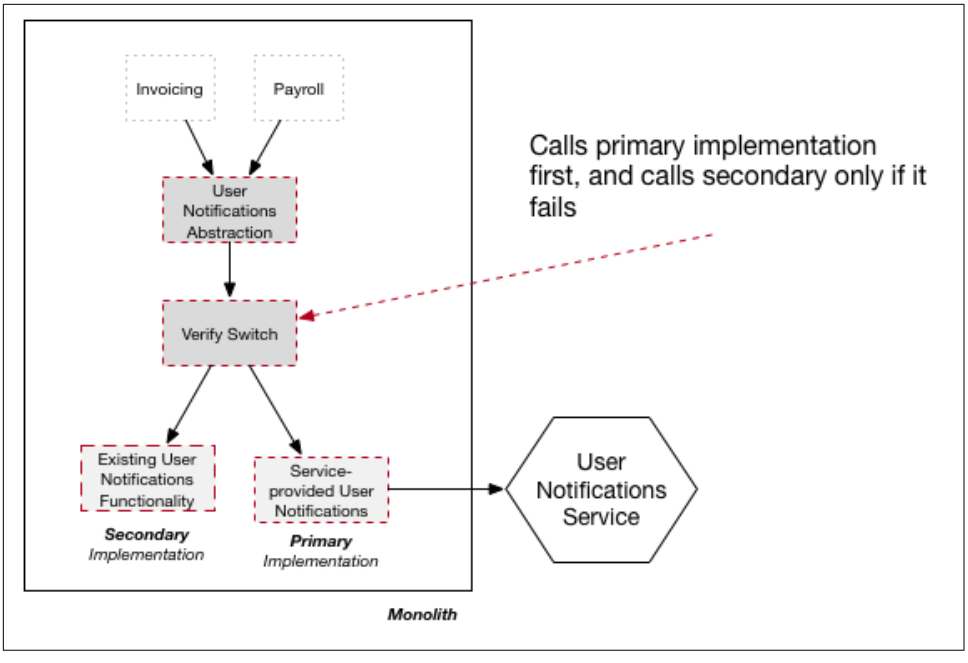


Figure 1-27. Verify Branch By Abstraction

This clearly adds some complexity, not only in terms of code but also in terms of reasoning about the system. Effectively, both implementations might be active at any given point in time, which can make understanding system behavior more difficult. If the two implementation are in anyway stateful, then we also have data consistency to consider. Although data consistency is a challenge in any situation where we are switching between implementations, The Verify Branch By Abstraction pattern allows for us to switch back and forth between implementations more dynamically, which makes managing data consistency much more complex as it greatly limits some of the options you can consider in this space. We'll explore issues around data in Chapter 2.

Nonetheless, this could work very well for situations where the alternative implementations are stateless, and where the risk associated with the change justifies the extra work.

Where To Use It

Branch By Abstraction is a fairly general purpose pattern, useful in any situation where changes to the existing codebase are likely going to take time to carry out, but where you want to avoid disrupting your colleagues too much. In my opinion it is a better option than the use of long-lived code branches in nearly all circumstances.

For migration to a microservice architecture, I'd nearly always look to use a strangler pattern first, as it's simpler in may regards. However there are some situations, as with `Notifications` here, where that just isn't possible.

This pattern also assumes that you can change the code of the existing system. If you can't, for whatever reason, you may need to look at other options, some of which we explore in the rest of this chapter.

Pattern: Decorating Collaborator

What happens if you want to trigger some behavior based on something happening inside the monolith, but you are unable to change the monolith itself? The Decorating Collaborator pattern can help greatly here. The widely known `Decorator` pattern allows you to attach new functionality to something without the underlying thing knowing anything about it - we are going to use a decorator to make it appear that our monolith is making calls to our services direct, even though we haven't actually changed the underlying monolith.

Rather than intercepting these calls before they reach the monolith, instead we allow the call to go ahead as normal. Then, based on the result of this call, we can call out to our external microservices through whatever collaboration mechanism we choose.

Let's explore this idea in detail with an example from MusicCorp.

Example: Loyalty Program

MusicCorp is all about our customers! We want to add the ability for them to learn points based on orders being placed, but our current order placement functionality is complex enough that we'd rather not change it right now. So the order placement functionality will stay in the existing monolith, but we will use a proxy to intercept these calls, and based on the outcome decide how many points to deliver, as shown in Figure 1-28.

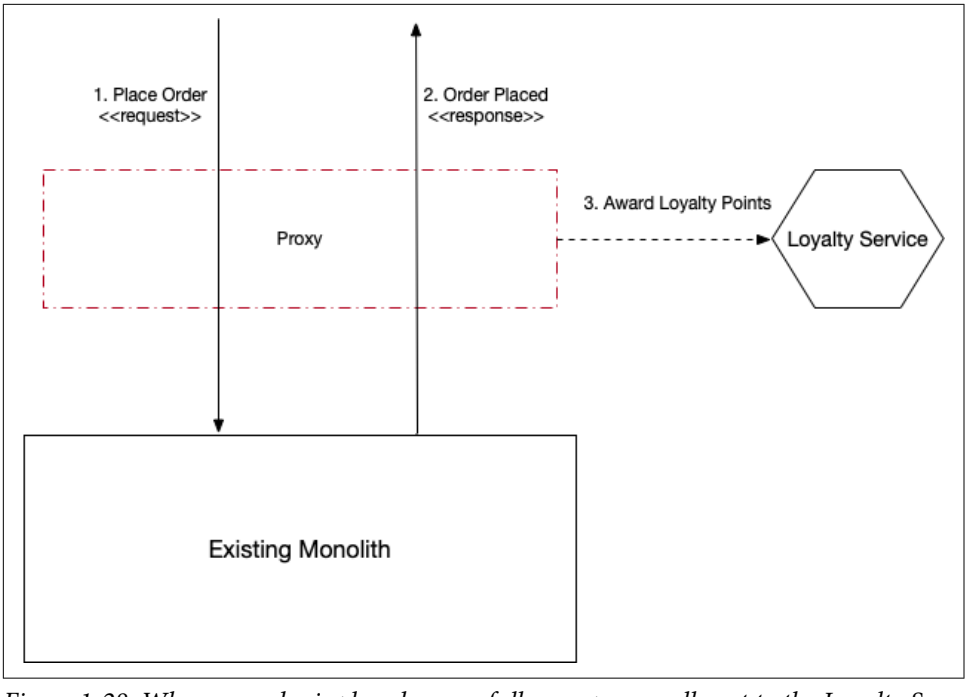


Figure 1-28. When an order is placed successfully, our proxy calls out to the Loyalty Service to add points for the customer

With the strangler pattern, the proxy was fairly simplistic. Now our proxy here is having do embody quite a few more "smarts". It needs to make it's own calls out to the new microservice and tunnel responses back to the customer. As before, keep an eye on complexity that sits in the proxy. The more code you start adding here the more it ends up becoming a microservice in it's own right, albeit a technical one, with all the challenges we've discussed previously.

Another potential challenge here is that we need enough information from the inbound request to be able to make the call to the microservice. For example, if we want to reward points based on the value of the order, but the value of the order isn't clear from either the Place Order request or response, we may need to look up additional information - perhaps calling back into the monolith to extract the required information as in Figure 1-29.

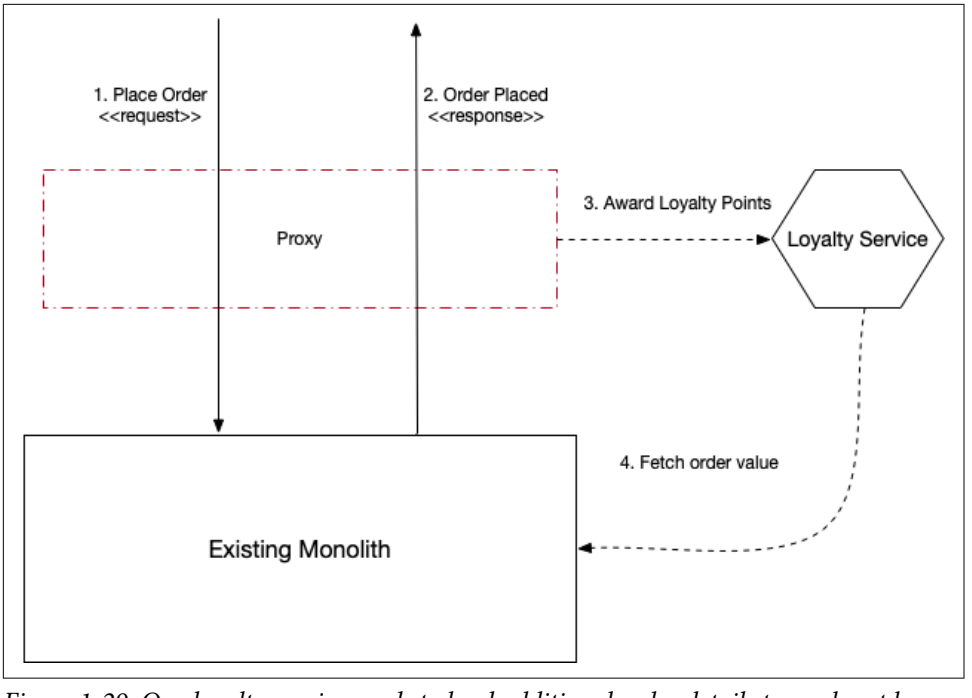


Figure 1-29. Our loyalty service needs to load additional order details to work out how many points to award

Given that this call could generate additional load, and arguably introduces a circular dependency, it might be better to change the monolith to provide the required information when order placement completes. This though would either require changing the code of the monolith, or else perhaps using a more invasive technique like Change-Data Capture.

Where To Use It

When kept simple, it is a more elegant and less coupled approach than Change-Data Capture. This pattern works best where the required information can be extracted from the inbound request, or the response back from the monolith. Where more information is required for the right calls to be made to your new service, the more complex and tangled this implementation ends up being. My gut feel is that if the request and response to and from the monolith don't contain the information you need, then think very carefully before using this pattern.

Pattern: Change-Data Capture

With Change-Data Capture, rather than trying to intercept and act on calls made into the monolith, instead we react to changes made in a data store. For Change Data Capture to work, the underlying capture system has to be coupled to the monolith's data store. That's really an unavoidable challenge with this pattern.

Example: Issuing Loyalty Cards

We want to integrate some functionality to print out loyalty cards for our users when they sign up. At present, a loyalty account is created when the a customer is enrolled. As we can see in Figure 1-30, when enrollment returns from the monolith, we only know that the customer has been successfully enrolled. For us to print a card, we need more details about the customer. This makes inserting this behavior upstream, perhaps using a Decorating Collaborator, more difficult - at the point where the call returns, we'd need to make additional queries to the monolith to extract the other information we need, and that information may or may not be exposed via an API.

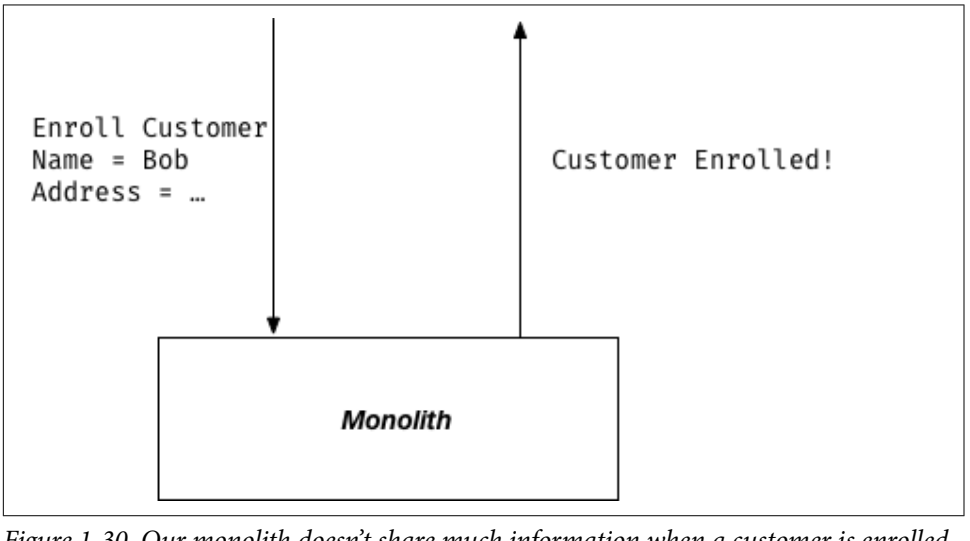


Figure 1-30. Our monolith doesn't share much information when a customer is enrolled

Instead, we decide to use Change Data Capture. We detect any insertions into the `LoyaltyAccount` table, and on insertion, we make a call to our new `Loyalty Card Printing` service, as we see in Figure 1-31. In this particular situation, we decide to fire a `Loyalty Account Created` event - our printing process works best in batch, so this allows us to build up a list of printing to be done in our message broker.

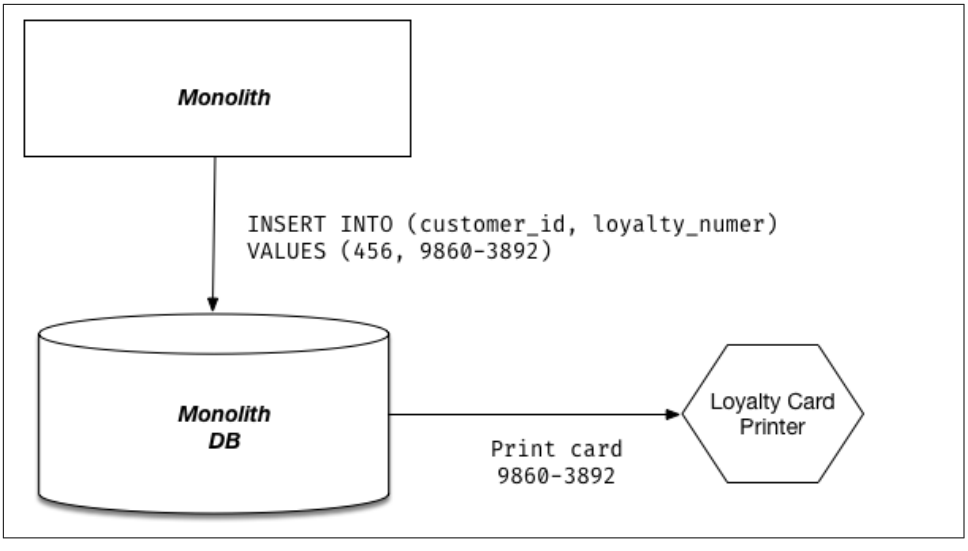


Figure 1-31. How change data capture could be used to call our new printing service

Implementing Change Data Capture

There are a number of techniques we could use to implement change data capture, all of which have different trade-offs in terms of complexity, reliability, and timeliness. Let's take a look at a couple of options.

Database Triggers

Most relational databases allow you to trigger custom behavior when data is changed. Exactly how these triggers are defined, and what they can trigger varies, but all modern relational databases support these in one way or another. In Figure 1-32 we can see an example where our service is called whenever an `INSERT` is made into the `LoyaltyAccount` table.

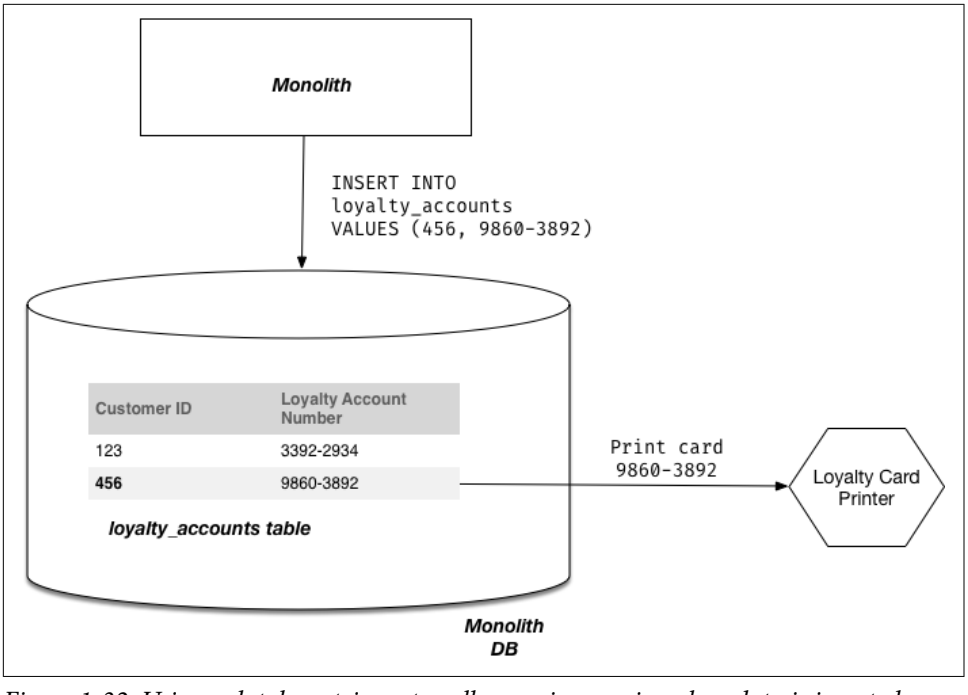


Figure 1-32. Using a database trigger to call our microservice when data is inserted

Triggers need to be installed into the database itself, just like any other stored procedure. There may also be limitations as to what these triggers can do, although at least with Oracle they are quite happy for you to call Web Services or custom Java code.

On first glance, this can seem like quite a simple thing to do. No need to have any other software running, no need to introduce any new technology. However like stored procedures, database triggers can be a very slippery slope.

A friend of mine, Randy Shoup, once said something along the lines of "Having one or two database triggers isn't terrible. Building a whole system off them is a terrible idea". And this is often the problem associated with database triggers. The more of them you have the harder it can be to understand how your system actually works. A large part of this is around the tooling and change management of database triggers - use too many and your application can become some baroque edifice.

So if you're going to use them, use them **very** sparingly.

Transaction Log Pollers

Inside most databases, certainly all mainstream transactional database, there exists a transaction log. This is normally a file, into which is written a record of all the

changes that have been made. For change data capture, the most sophisticated tooling tends to make use of this transaction log.

These systems run as a separate process, and their only interaction with the existing database is via this transaction log, as we see in Figure 1-33. It's worth noting here that only committed transactions will show up in the transaction log (which is sort of the point).

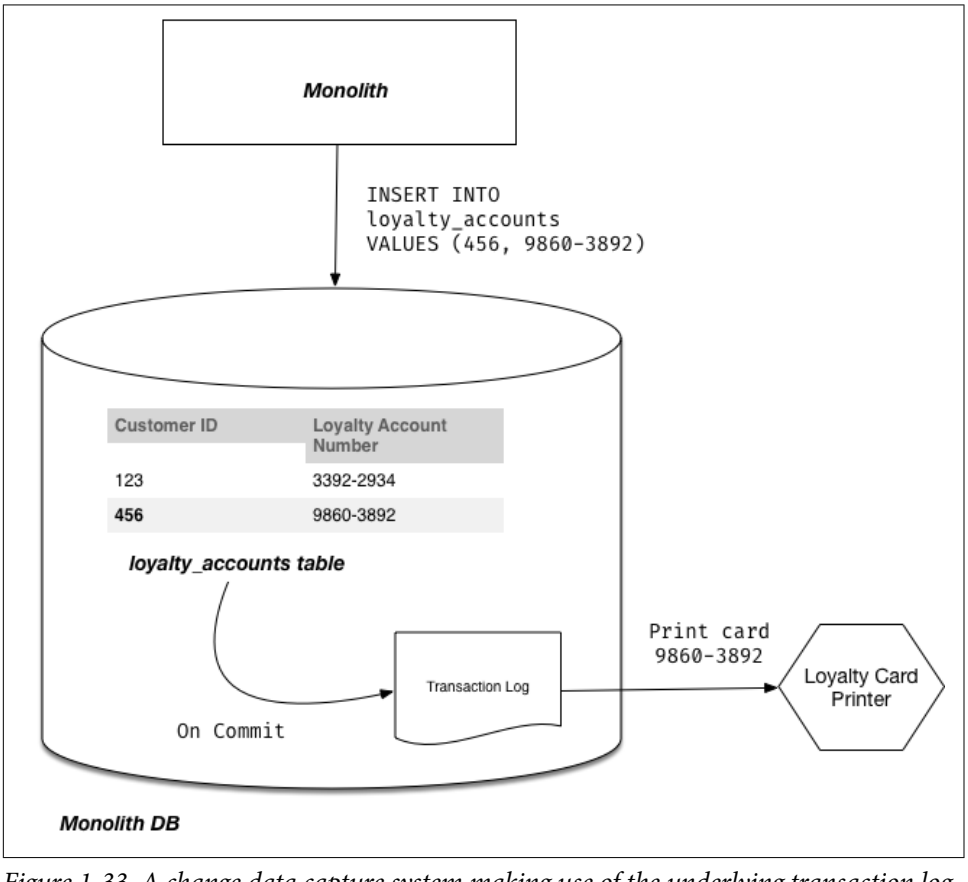


Figure 1-33. A change data capture system making use of the underlying transaction log

These tools will require an understanding of the underlying transaction log format, and this typically varies across different types of databases. As such, exactly what tools you'll have available here will depend on what database you use. There are a huge array of tools in this space, although many of them are used to support data replication. There are also a number of solutions designed to map changes to the transaction log to messages to be placed onto a message broker - this could be very useful if your microservice is asynchronous in nature.

Restrictions aside, in many ways this is the neatest solution for implementing change data capture. The transaction log itself only shows changes to the underlying data, so you aren't worried about working out what has changed. The tooling runs outside the database itself, and can run off a replica of the transaction log, so you have fewer concerns regarding coupling or contention.

Where To Use It

Change-data capture is a very useful general purpose pattern, especially if you need to replicate data (something we'll explore more in Chapter 2). In the case of microservice migration, the sweet spot is where you need to reach to a change in data, but are unable to intercept this either at the perimeter of the system using a strangler or decorator, and cannot change the underlying codebase. In general, I try and keep the use of this pattern to a minimum due to the challenges around some of the implementations of this pattern - database triggers have their downsides, and the full blown change data capture tools that work off transaction logs can add significant complexity to your solution.

Pattern: Parallel Running

There is only so much testing you can do of your new implementation before you deploy it. You'll do your best to ensure that your pre-release verification of your new microservice is done in as production-like a way as possible as part of a normal testing process, but we all understand that it isn't always possible to think of every scenario that could occur in a production setting. But there are other techniques available to us.

Both "Pattern: Strangler Application" on page 10 and "Pattern: Branch By Abstraction" on page 32 allow us to co-exist old and new implementations of the same functionality in production at the same time. Typically, both of these techniques allow us to execute either the old implementation in the monolith, or the new microservice-based solution. To mitigate the risk of switching over to the new service based implementation these techniques allow us to quickly to switch back to the previous implementation.

When using a Parallel Run, rather than calling either the old or the new implementation, instead we call *both*, allowing us to compare the results to ensure they are equivalent. Despite calling both implementations, only one is considered the source of truth at any given time. Typically, the old implementation is considered the source of truth until the ongoing verification reveals that we can trust our new implementation.

This pattern has been used in different form for decades, although typically it is used to run two systems in parallel. I'd argue this pattern can be just as useful within a single system, when comparing two different implementations of the same functionality.

This technique can be used to verify not just that our new implementation is giving the same answers as the existing implementation, but that it is also operating within acceptable non-functional parameters. For example, is our new service responding quickly enough? Are we seeing too may timeouts etc?

Example: Comparing Credit Derivative Pricing

Many years ago, I was involved in a project to change the platform being use to perform calculations on a type of financial product, called Credit Derivatives. To simplify things greatly, credit Derivatives allow you to track the underlying health of some asset - for example, a company. These products were offered to customers by the bank, allowing them (amongst other things[12]) to offset their own risk. For example, if you did a lot of business with MusicCorp but were worried about them going bust (as if!), you could buy a credit derivatives that would pay out if that happened. I pay some money for the product, but effectively I'm taking out insurance.

Before issuing a trade, the bank I was working at needed to make sure it was a good deal for the bank. Would we make money on this trade? Once issued, market conditions would also change. So they also needed to assess the value of current trades to make sure they weren't vulnerable to huge losses as market conditions changed[13].

Due to the amounts of money involved, and the fact that some people's bonuses were based in part on the value of the trades that had been made, there was a great degree of concern over us greatly reworking the existing system. We took the decision to run the two sets of calculations side by side, and carry out daily comparisons of the results. The pricing events were triggered via events, which were easily to duplicate such that both systems carried out the calculations, as we see in Figure 1-34.

12 It's worth calling out here that while hedging risk is something that is done with Credit Derivatives, they are used *far* more for speculation

13 Turned out we were *terrible* at this as an industry. I can recommend The Big Short by Martin Lewis as an excellent overview of the part that credit derivatives played in the Global Financial Collapse. I often look back at the very small part I played in this industry with a great deal of regret. It turns out not knowing what you're doing and doing it anyway can have some pretty disastrous implications.

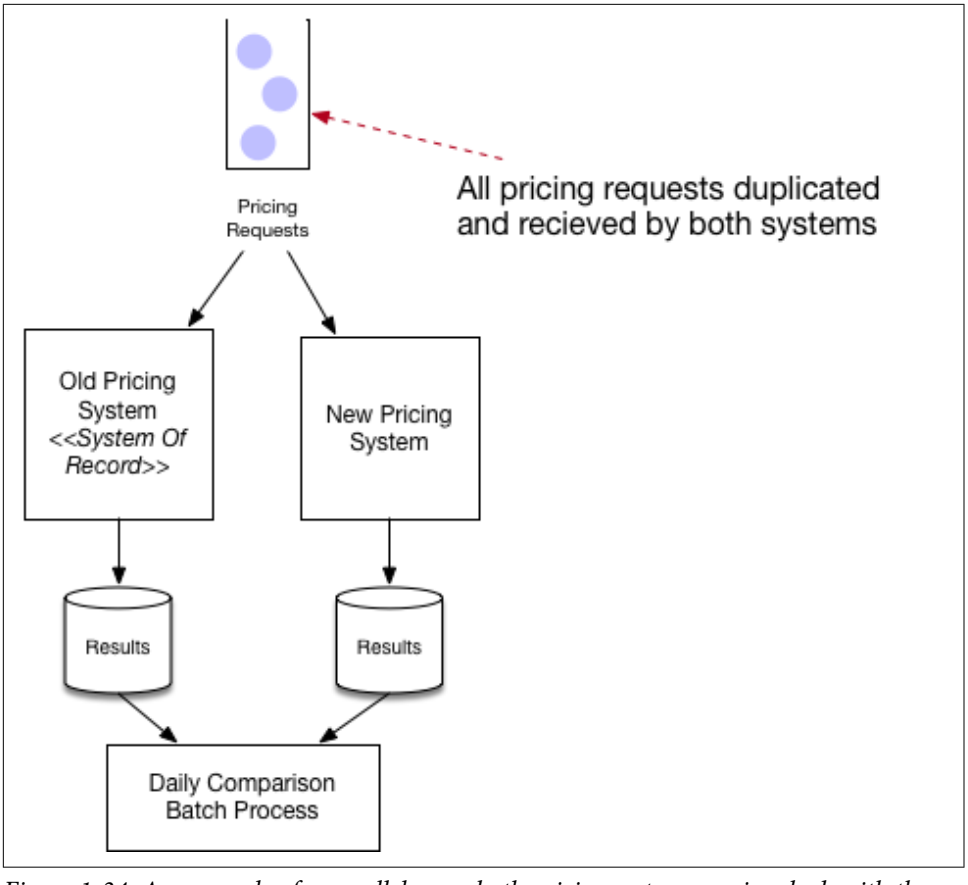


Figure 1-34. An example of a parallel run - both pricing systems are invoked, with the results compared offline

Each morning, we'd run a batch reconciliation of the results, and would then need to account for any variations in results. We actually wrote a program to perform the reconciliation - we presented the results in an Excel spreadsheet making it easy to discuss the variations with the experts at the bank.

It turned out we did have a few issues that we had to fix, but we also found a larger number of discrepancies caused by bugs in the existing system. This meant that some of the different results were actually correct, but we had to show our working (made much easier due to surfacing the results in Excel). I remember having to sit down with analysts and explain why our results were the correct by working things out from first principles.

Eventually, after a month, we switched over to using our system as the source of truth for the calculations, and some time later we retired the old system (we kept it around

for a few more months in case we needed to carry out any auditing of calculations done on the old system).

Example: Real Estate Listing

As we discussed earlier in "Example: FTP" on page 22, the Real Estate company ran bot both their listing import systems in parallel, with the new microservice which handled list imports being compared against the existing monolith. A single FTP upload by a customer would cause both systems to be triggered. Once they had confirmed that the new microservice was behaving in a equivalent fashion, the FTP import was disabled in the old monolith.

N-Version Programming

It could be argued that a variation of Parallel Run exists in certain saftey critical control systems, such as fly-by-wire aircraft. Rather than relying on mechanical controls, airliners increasingly rely on digital control systems. When a pilot uses the controls, rather than this pulling cables to control the rudder, instead fly-by-wire aircraft this sends inputs to control systems that decide how much to turn the rudder by. These control systems have to interpret the signals they are being sent and carry out the appropriate action.

Obviously, a bug in these control systems could be very dangerous. To offset the impact of defects, for some situations multiple different implementations of the same functionality are used side by side. Signals are sent to all implementation of the same sub-system, who then send their response. These results are compared and the "correct" one selected, normally by looking for a quorum amongst the participants. This is a technique known as N-version programming[14].

The end goal with this approach is not to replace any of the implementations, unlike the other patterns we have looked at in this chapter. Instead, the alternative implementations will continue to exist alongside each other, with the alternative implementations hopefully reducing the impact of a bug in any one given subsystem.

Verification Techniques

With a parallel run, we want here to compare functional equivalence of the two implementations. If we take the example of the Credit Derivative pricer from above, we can treat both versions as functions - given the same inputs, we expect the same outputs. But we also can (and should) validate the non-functional aspects too. Calls

14 Algirdas Avizienis, Liming Chen "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation", published in Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995

made across network boundaries can introduce significant latency and can be the cause of lost requests due to timeouts, partitions and the like. So our verification process should also extend to making sure that the calls to the new microservice complete in a timely manner, with an acceptable failure rate.

Where to use it

Implementing Parallel Run is rarely a trivial affair, and is typically reserved for those cases where the functionality being changed is considered to be high risk. For example we'll examine an example of this pattern being used for medical records in Chapter 2. I'd certainly be fairly selective about where I used this pattern - the work to implement this needs to be traded off against the benefits you gain. I've only used this pattern myself once or twice, but in those situations it has been hugely useful.

It's worth calling out that a Parallel Run is different to what is traditionally called Canary Releasing. A canary release involves directing some subset of the calls to the new implementation, with the majority of calls still being served by the old system. The idea being that if the new system has a problem, then only a subset of requests are impacted. With Parallel Run we call *both* implementations. Both techniques can be used to verify if our new functionality is working correctly, and reduce the impact if this turns out not to be the case, but Parallel Run can provide a more thorough level of comparison between implementations. The tradeoff here is complexity of implementations - Canary Releasing is much more straightforward to implement in most situations.

Obviously, with both implementations being called, you have to consider what this means. Is doing the same thing twice going to lead to sub-optimal side effects? Using a Parallel Run as part of our migration of `User Notifications` for example would likely be a bad idea - as we could end up with sending all notifications to our users twice!

Summary

As we've seen, there are a vast array of techniques that allow for the incremental decomposition of existing code bases, and can help you ease yourself into the world of microservices. In my experience, most folks end up using a mix of approaches - it's rare that one single technique will handle every situation. Hopefully what I've been able to do so far is give you a variety of approaches and enough information to work out which techniques may work best for you.

We have though glossed over one of the bigger challenges in migrating to a microservice architecture - data. We can't put it off any longer! In our next chapter, Chapter 2, we'll explore how to migrate data and break apart the databases.

O'REILLY®

Introduction to AWS IaaS Solutions

Deploying and Managing Amazon Web Services

Eric Wright

Preface

Welcome to the *Introduction to AWS IaaS* solutions guide. The goal of this guide is to introduce systems administrators, systems architects, and newcomers to Amazon Web Services (AWS) to some powerful core offerings on the AWS platform.

You will learn common terms, design patterns, and some specific examples of how to deploy Infrastructure as a Service (IaaS) solutions for compute, network, and storage to AWS using the AWS command-line interface (CLI) and the AWS web console. By the end, you will be able to launch and manage AWS solutions, including compute instances and storage, as well as understand the implications and requirements for security and access management for your IaaS resources on AWS.

Additional resources are provided throughout the guide for you to further explore some of the services and technical examples. Resources, code samples, and additional reading links for this guide are available online.

Thanks go out to the entire AWS technical community, the O'Reilly team, and my family for the help and guidance in creating this guide.

— Eric Wright (@DiscoPosse),
November 2018

CHAPTER 1
Introduction to AWS

Today's systems administrators need to acquire and strengthen their skills on public cloud platforms. Amazon Web Services (AWS) began as an infrastructure to run the Amazon.com website and, as of this writing, has since grown to be the largest public cloud provider.

AWS provides a wide variety of service offerings from Infrastructure as a Service (IaaS) through to the application and Platform as a Service (PaaS) and nearly everything in between, which offers alternatives to running infrastructure on-premises.

AWS services are available on demand throughout the world, which makes it a compelling place to run infrastructure and applications. You might already have some familiarity with AWS, which is fine; this guide is geared toward folks who are early in their AWS journey or those looking to solidify their understanding of AWS IaaS solutions for compute, block storage, and networking.

AWS Command-Line Interface Installation

You will be using the AWS command-line interface (CLI) along with the AWS console for the examples in this guide. You can find CLI installation instructions online.

In this chapter, we begin our journey by exploring the AWS public cloud platform with a focus on the IaaS features. We cover general architectural features of the AWS cloud including geographic regions and availability zones. This will give you a comprehensive

understanding of the basics needed to deploy your IaaS workloads on AWS.

A full glossary of AWS terms is available in the additional resources online.

Regions

AWS infrastructure is comprised of many services available in many areas of the world known as *Regions*. These Regions provide geographic availability with close proximity for low-latency access. AWS also provides the GovCloud region, which is a specialty region for government agencies and provides additional compliance and security requirements.

Each Region is located within a country's boundary to ensure protection by any regulatory requirement for geo-locality of workloads, data, and services. Some Regions might also require special access such as Asia Pacific (Osaka) due to country-specific regulations.

Edge connectivity is provided globally, which also gives service-focused access to features like the Content Delivery Network (CDN), Domain Name System (DNS) using Route 53, Identity and Access Management (IAM), and others. This ensures that you and your customers have rapid access to the resources as well as geographic availability in the case of loss of access to a particular Region.

Regions are names identified by a two-letter country code (e.g., US, EU, CA, CN), a general location (e.g., East, West, Central), and a numeric marker; for example:

- US-East (North Virginia) Region: **us-east-1**
- US West (Oregon) Region: **us-west-2**
- EU (Ireland) Region: **eu -west 1**
- AWS GovCloud (US): **us-gov-west-1**

It is helpful to know the Region names and their programmatic short name when using the AWS CLI or other systems that deploy and manage AWS infrastructure. You will see references throughout this guide for the AWS CLI and links to more resources for other configuration management and Infrastructure as Code (IaC) tools (e.g., Terraform, RackN, Chef, Puppet, Ansible).

Availability Zones

Within each of the AWS Regions are physically separated and isolated datacenters known as *Availability Zones* (AZs) which you can see illustrated in Figure 1-1. Each AZ has high-speed and low-latency networking within a Region and is described as being within a metropolitan distance (approximately 100 km) to provide low enough latency for replication of services and data while also providing protection against a significant business disruption such as power grid failure or some other localized outage.

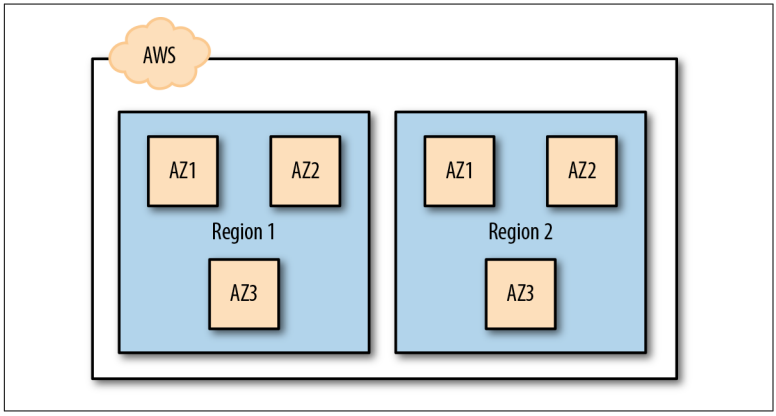


Figure 1-1. Logical view of AZs within Regions

AWS does not publish the physical locations or proximity between datacenters or any technical specifications on the hardware environments. It can be possible to have a single AZ span more than one datacenter; however, there will not be two AZs sharing the same datacenter infrastructure.

Network Access for AWS Infrastructure

Administrative access to AWS is available anywhere using either the AWS console in a web browser or using the AWS CLI, which can each be used on a variety of devices. Network access for the applications and specific resources within your AWS environment is what you must design for to use the services that you create.

There are three methods that you can use to access your AWS infrastructure and services:

Internet
: Open connectivity via direct network across the internet using an Internet Gateway within your AWS environment

Virtual Private Network (VPN)
: Software or hardware VPN with an endpoint on-premises and continuous or on-demand tunnel access to your AWS environment using your own VPN devices

Direct Connect
: Dedicated hardwire access to the AWS network which is available through networking and cloud service providers for high-speed, low-latency, and routed access directly to your AWS Region

These options are not mutually exclusive. Your requirements for access will vary from application to application and service to service. Lots of AWS services are used directly over the internet with public/private key, AWS credentials, and other access-level controls. This reduces the need for dedicated networks for many customers.

Design decisions around internet access, workload placement, and data locality are important because you might require subnet accessibility, internet gateways, IP routing, and other infrastructure configuration from Region to Region.

Direct Connect is ideal for organizations that want bidirectional direct access to network resources. You can use this for database replication services, disaster recovery, replication between on-premises datacenters and the AWS cloud, data warehouse accessibility, and much more.

Just imagine that you want to have your data on-premises but have applications running on AWS to access that data. Using direct network access, caching services, and other options now opens the door to exciting hybrid deployments for real-time applications.

Design Patterns for Availability with AWS

The best way to ensure availability is to take advantage of existing AWS services and its resilient infrastructure. Certain trade-offs must occur when you design for resiliency because of cost and performance. As we get further into this guide, we explore more of the service-specific deployment patterns.

Here are some key tips for designing for availability for core services:

Think globally, act locally
: Just like the earth-friendly phrase goes, you should utilize services with global availability but be mindful of where your customers and users access the environment. Make use of CDNs, caching, and cross-Region services where possible for the best consumer experience.

Use multiple AZs
: They are called "Availability Zones" for a reason. Utilize more than one AZ within your Regions for safety. Designing your network strategy must include this or else you might bump into network addressing challenges as you try to expand later.

Cross-region deployments
: For broad availability, use services that can span Regions as well as the AZs within them. Treat a Region like you would a Metropolitan Area Network and build applications to be able to be run and recovered across Regions.

Back up your data and configuration
: Cloud services can be distributed and have high availability, but that does not ensure the backup of resources. Backups are needed for either *point-in-time* recovery or complete loss recovery after a significant disruption. Even replication will replicate errors and data loss, leaving your team with only backups and snapshots as recovery options.

You will find there are fewer limitations on architecture than there are on your budget. Resiliency is available at nearly every layer of the stack provided that you can budget for it. The value of on-demand infrastructure is that you can scale as needed and design for these burst patterns.

You must design your networking strategy in advance of these bursts and expansions. Early decisions about network addressing within and across your AZs and Regions can affect growth and expansion.

AWS services have quotas and some upper-bound technical limits. For example, you can have no more than five Virtual Private Clouds (VPCs) per Region by default. You can order increases, which also increases dependencies, such as an increase to the number of inter-

net gateways per Region. Hard limit examples include 500 security groups per VPC, and 16 security groups per network interface. All service limits and quotas are available in the AWS documentation.

Conclusion

Now that you have a good understanding of the general AWS infrastructure, we now move on to the AWS VPC environment. This is the basis of access control, networking, and network security for the AWS IaaS platform where you will be launching your compute and storage resources.

CHAPTER 2

Basic Networking and Security with Amazon Web Services Virtual Private Cloud

In this chapter, we explore some of the foundational features of the Amazon Web Services (AWS) Virtual Private Cloud (VPC) platform. You will learn about networking and security options and see practical use with an example cloud instance configuration. These are important to understand when bringing your cloud workloads online. There is the potential to expose data and services to the public-facing internet, which also opens the door to vulnerability and attack.

Understanding VPC features and how to configure one step-by-step is important if you are studying for the AWS *Solutions Architect Associate* exam. It is important in general for your AWS product knowledge, but as many exam-related resources indicate, VPC knowledge might be heavily featured in the certification process.

What Is VPC?

VPC is the logical construct that gives your workloads a common networking and security boundary. Every AWS Region you launch a workload into has a default VPC to allow for immediate use of compute and storage resources without the need to set up a specific VPC.

Creating a VPC is free. The only costs that come up are when you create a Network Address Translation (NAT) gateway or deploy resources within the VPC. There is also an option to create a gateway-type VPC endpoint, which uses internal networking to reach other AWS resources (e.g., Amazon Simple Storage Service [Amazon S3] object storage and Amazon Relational Database Service [RDS] database services). This is handy as you grow your AWS usage across services and want to avoid accessing other AWS-hosted data over the public internet, which incurs networking fees.

Features and capabilities for VPC include the following:

Public subnets
: The ability to create one or more public-facing subnets for internet access using an internet gateway for both.

Private subnets
: The ability to create one or more private subnets for internal communications between VPC resources using both.

VPC Peering
: The option to connect multiple VPCs as a single routed private network, which you can do between VPCs in a Region, across Regions, and even between AWS accounts.

It is recommended to set up your resources within a VPC to pool together the active environment and the supporting networking and security configuration. You can create all of your access rules, networking policies, and give out access in granular ways to the VPC and below with the Identity and Access Management (IAM) service. We implement our example here using root or a full administrative access profile.

Tagging is also your friend for many reasons. Tagging every resource and workload to identify it (production, test, dev, web server, db server, owner, etc.) makes other administrative processes easier in future. You can tag nearly every object in AWS.

Core Networking and Security on AWS

VPC networking and security are a fundamental part of your AWS Infrastructure as a Service (IaaS) design and day-to-day operations. Networking and security are paired up here because they often share the common goal of securing and isolating network access to your

AWS resources. The VPC construct is a way to have a set of access rules and logical networks that can be shared by multiple AWS resources. We cover an example application that uses compute, storage, and networking to illustrate a real use case.

Access to your resources can be protected by Secure Shell (SSH) key pairs, which we use for the example applications. Be sure to set up your first key pair in the us-west-2 region if you would like to follow along with the examples.

Instructions are available in the AWS user guide for creating and uploading your key pair.

VPC Subnets

You have four choices when creating your VPC:

VPC with single public subnet
: Defaults with /16 network and a /24 subnet using Elastic IP for public IP addressing. This is good for general purpose, public-facing web servers and for simplified deployments.

VPC with public and private subnets
: Defaults with /16 network and two /24 subnets using Elastic IP on public and NAT for private IP addressing. This is great for adding application servers or other backend resources that can be on private subnets with access to frontend servers on the private VPC network.

VPC with public and private subnets and hardware VPN access
: Same public/private as the aforementioned public and private subnets with IPsec VPN tunnel from the private subnets to your corporate network. Now you can extend your on-premises environment to share data and application access across a hybrid environment.

VPC with a private subnet only and hardware VPN access
: No public subnets, with an IPsec VPN tunnel to your corporate network. This is a great way to use burstable resources on AWS that are able to access your on-premises environment (e.g., batch processing and AI applications).

Note that in each of these scenarios, there are usage charges for both the VPN and NAT on VPC. Our example deployment will use a VPC with single public subnet, as shown in Figure 2-1, because they are just getting started with their AWS usage and have no need for VPN or private subnets.

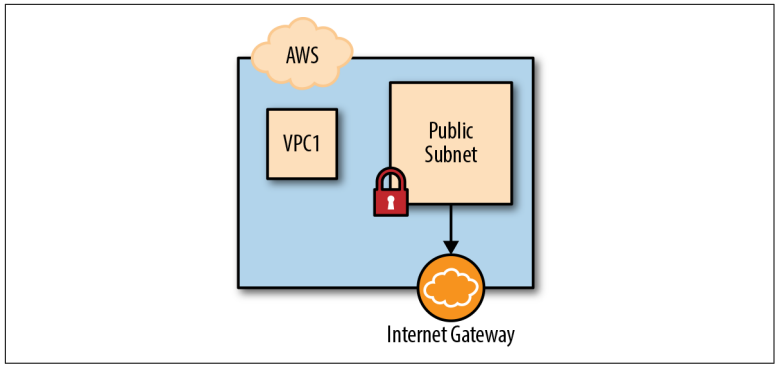


Figure 2-1. Example VPC with single public subnet

If you choose to use private subnets for your architecture, you need NAT access to those private subnets, which you can do with an Elastic IP (must be preallocated before VPC creation) or by using an EC2-based NAT instance. You choose this during the VPC wizard or you can configure it manually if you build your VPC from scratch or after initial deployment at any time.

Your choice of subnetting is one to spend some time on. Especially when it comes to your subnet allocation. Even with the use of private IP subnets (10.0.0.0/8, 172.16.0.0/16, 192.168.0.0/24), you are likely to find yourself running into colliding IP address ranges if you are not careful. Each private subnet in an Availability Zone (AZ) will require you to select from the parent range created for your VPC.

Here's an example for IPv4 subnet of a VPC with one public subnet and two private subnets:

VPC IPv4 CIDR Block
: 10.0.0.0/16 (65,531 available IP addresses)

IPv4 public subnet CIDR Block
: 10.0.0.0/24 (251 available IP addresses)

IPv4 private subnet one CIDR Block
: 10.0.1.0/24 (251 available IP addresses)

IPv4 private subnet two CIDR Block
: 10.0.2.0/24 (251 available IP addresses)

Notice that we have given private IP addresses to the "public" subnet because these are the private interface access addresses used for inter-instance and intra-VPC communication. As instances are brought online, you can also assign them a public IP address given by AWS.

Instances are created with private IP addresses automatically. You also can opt to have a public IP address and AWS-assigned Domain Name System (DNS) entry on an interface at launch. This public IP address is not persistent and can change when the instance restarts. We look at Elastic IP options further into the chapter, which helps with this issue of nonpersistent IP addresses.

Security Groups

VPC Security Groups are stateful policies that allow inbound and outbound network traffic to your EC2 instances. You can apply Security Groups to your EC2 instances and modify them in real time at any time. Inbound rules are defined by port/protocol plus the source network and a description, as shown in Figure 2-2. Notice the Source can be custom, anywhere, or the "My IP" option which detects your IP from the browser and assigns it to the rule.

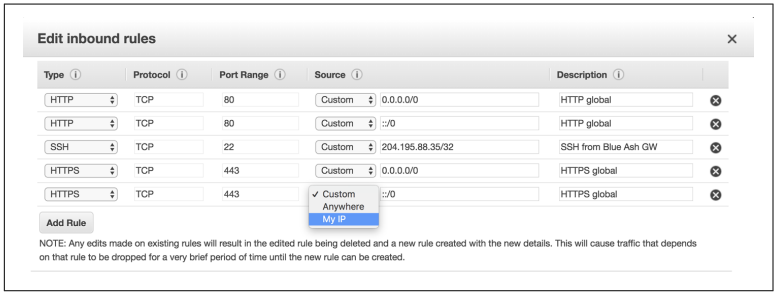


Figure 2-2. Inbound rules on a Security Group

Use the "Anywhere" option as a target/source carefully. It is ideal to use as narrow a network range as possible when creating rules, as well, such as specifying RDP from a particular IP address with a /32 for the CIDR subnet (e.g., 204.195.21.134/32). Granular network access can help in reducing the exposure and risk for your AWS workloads.

You can assign multiple Security Groups to each instance. This is helpful if you have a global rule for allowing SSH or Internet Control Message Protocol (ICMP) that you want to have on the entire VPC along with specific instance-related rules. Figure 2-3 shows how you can use the AWS console to attach multiple Security Groups to your instances.

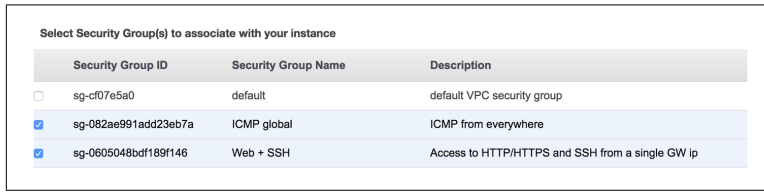


Figure 2-3. Choosing multiple Security Groups

NOTE

The most permissive of the cumulative rules applies when multiple Security Groups or rules are applied to an instance.

Example: one rule for inbound SSH from only one IP address and another inbound SSH rule from Anywhere will result in allowing SSH from Anywhere. Each EC2 instance has the cumulative inbound rules visible from the Description tab, as shown in Figure 2-4, along with which Security Group the rule comes from.

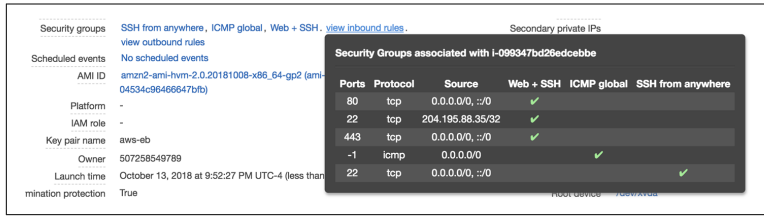


Figure 2-4. View inbound rules and Security Groups

Elastic IPs

You can assign a persistent public IP address to your EC2 instance by using an Elastic IP (EIP). This is free for each provisioned instance or charged at 0.01$ per hour for disassociated EIPs. Even an instance with a public IP address assigned at launch is a dynamic and ephemeral IP. Choosing an EIP means that you will have it con-

sistently for your account and can create a DNS association for that resource now.

You are allowed to map only one EIP to a network interface. You can assign multiple network interfaces to a single instance as a way to allow multiple EIPs to be used. Examples where this can be used is a web server with multiple sites, each assigned to a specific IP address and DNS name.

The default quota of EIPs for each account is five per region. This is important to know so that you can request increases if needed to meet your EIP requirements.

AWS CLI Command Basics

You will notice consistent patterns with the AWS CLI for command structure. This is how it breaks down:

```
aws [options] <command> <subcommand> [parameters]
```

Example: give a list of EC2 instances:

```
aws ec2 describe-instances
```

Example: create a VPC routing table:

```
aws ec2 create-route-table --vpc-id vpc-006960a6c4d805f10
```

Commands use verbs, which include `create`, `describe`, `associate`, `attach`, `delete`, `detach`, `import`, `modify`, and others. The full CLI structure and command reference is available online in the AWS documentation reference page.

Deployment Example: Web Application

For this example, your customer is the Utility Muffin Research Kitchen (UMRK) company, which needs a basic website to display its supply catalog at *http://supplies.utilitymuffinresearchkitchen.com*. The company will use this website to run its custom web application code that is built for a web server that is certified on Amazon Linux.

UMRK uses an on-premises local balancer to route traffic to the web servers and will be using two web servers in AWS to distribute the load. It will want to keep all information together in a geographic region while ensuring availability by spreading the servers across two AZs, as shown in Figure 2-5.

The UMRK operations team needs HTTP and SSH access to each instance to be able to display the website and to manage the web server configuration and code. UMRK is located in Blue Ash, Ohio, with their primary distributors in Cincinnati, so they will choose to deploy into the US (Ohio) Region (us-east-2).

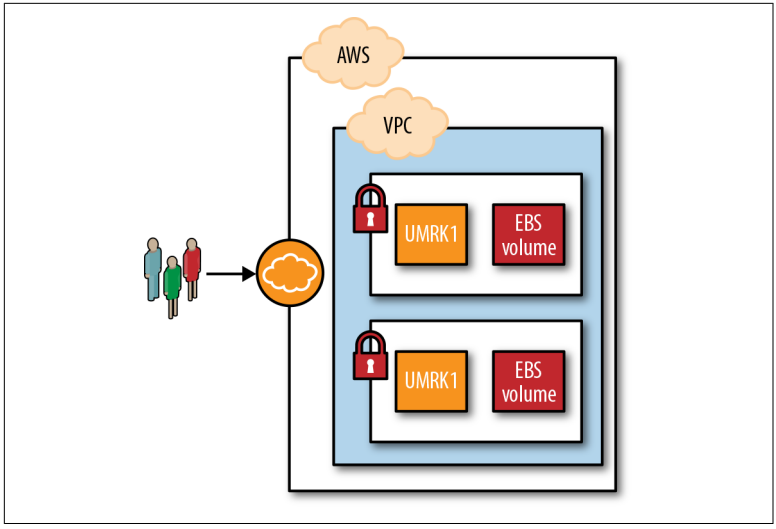


Figure 2-5. UMRK's architecture diagram

Deploying the UMRK VPC

Let's walk through the setup process here for configuring the UMRK VPC based on the requirements that we just defined:

- VPC will be in us-east-2 (Ohio)
- SSH key must be uploaded in advance
- Two IPv4 public subnets will be used for resiliency

To begin, go to the VPC service in your AWS web console, which brings you to the default page and features a prominent Launch VPC Wizard button similar to that shown in Figure 2-6. Click the button to get started with your new VPC.

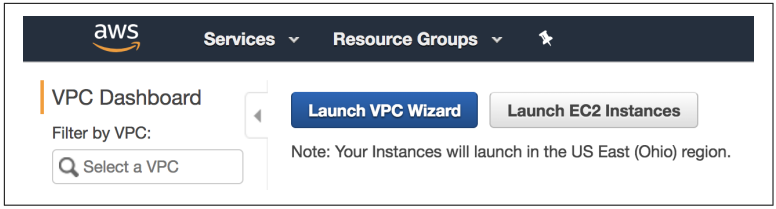


Figure 2-6. Launch the Create VPC Wizard

The four options in Figure 2-7 show the various configurations for a VPC that we discussed earlier in this chapter; for this example, choose Single Public Subnet.

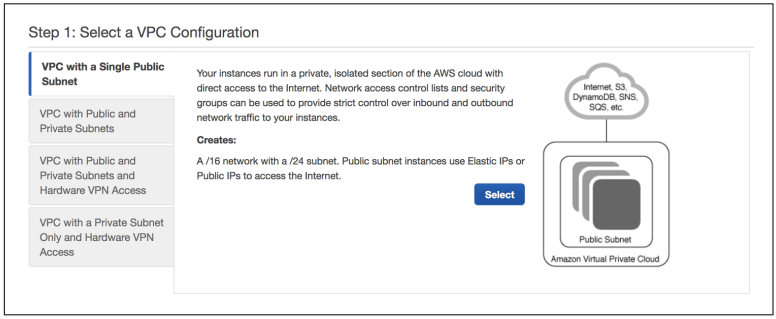


Figure 2-7. Choose the Single Public Subnet option

The UMRK requirements are for a limited number of workloads for now. A good practice is to name your resources (e.g., subnets) with meaningful and readable names. Lots of areas within the AWS console and AWS CLI will include the description and name, so this makes future administration a little easier. Figure 2-8 shows the wizard filled in for the UMRK example using a 10.0.0.0/16 VPC network and a 10.0.0.0/24 public subnet allocation.

It is also ideal to set the Availability Zone manually. This way you know precisely where the first AZ is and which additional AZs to use if you want more resiliency. AWS automatically creates a Fully Qualified Domain Name (FQDN) for your resource, which will display after it's created.

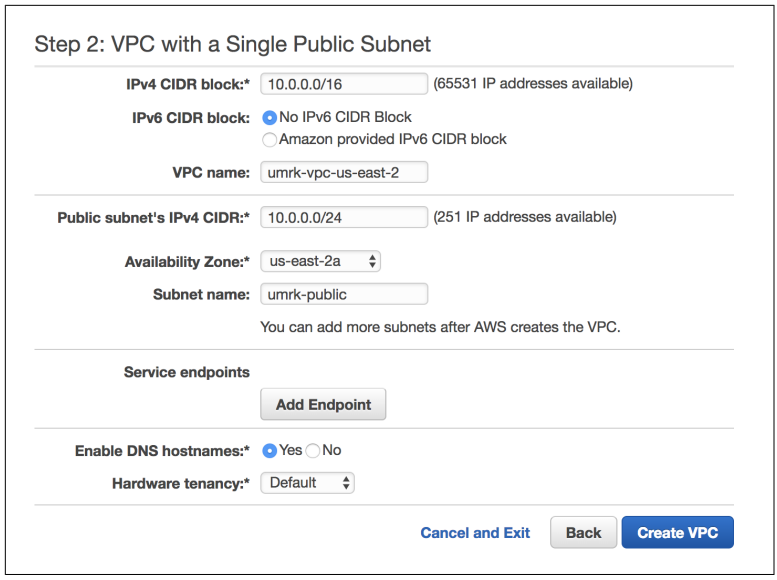


Figure 2-8. Entering your UMRK VPC details

At this point you have your VPC created with a public subnet. Choose the Subnets option in the VPC console and follow along with Figure 2-9 to add the second subnet with another /24 IP range within the 10.0.0.0/16, which is the network range for the VPC. Let's use 10.0.3.0/24 and call this umrk-public-2c to make it easy to know which subnet it's on.

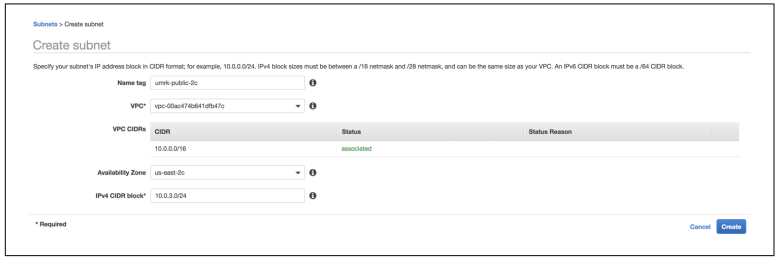


Figure 2-9. Create the second public subnet

Now you have your VPC created with two public subnets. Notice that the networking defaults to /16 and /24, which might work for your environment. There are no security groups in the new VPC by design. You must create new ones directly or when you create your EC2 instances.

Using AWS CLI to Create a VPC

The CLI process is a bit more involved than some of the other examples. This gives you an idea of all that is happening when you use the VPC wizard.

The example commands here are using sample IDs, which you will need to replace with those for your environment.

Following this sequence of CLI commands will create a VPC:

Create your VPC

```
aws ec2 create-vpc --cidr-block 10.0.0.0/16
```

Create your first subnet

```
aws ec2 create-subnet --vpc-id vpc-006960a6c4d805f10
  --cidr-block 10.0.0.0/24 --availability-zone us-east-2a
```

Create your second subnet

```
aws ec2 create-subnet --vpc-id vpc-006960a6c4d805f10
  --cidr-block 10.0.3.0/24 --availability-zone us-east-2c
```

Create your internet gateway

```
aws ec2 create-internet-gateway
```

Attach your internet gateway

```
aws ec2 attach-internet-gateway --internet-gateway-id
  igw-04a86ecc70949724d --vpc-id vpc-006960a6c4d805f10
```

Create a VPC routing table

```
aws ec2 create-route-table
  --vpc-id vpc-006960a6c4d805f10
```

Create a default route to move traffic to the internet gateway

```
aws ec2 create-route
  --route-table-id rtb-01b04201fe9909f9b
  --destination-cidr-block 0.0.0.0/0
  --gateway-id igw-04a86ecc70949724d
```

Add a route to your first subnet

```
aws ec2 associate-route-table
  --subnet-id subnet-01789b9231f45eb62
  --route-table rtb-01b04201fe9909f9b
```

Add a route to your second subnet

```
aws ec2 associate-route-table
  --subnet-id subnet-03c4b8c88971c3097
  --route-table rtb-01b04201fe9909f9b
```

Keep track of the IDs that are generated from the output of each command because they are needed for commands further into the process. Output from each command is in JSON format. You can also pull the data from the AWS console if needed.

Generating Elastic IPs

The last step you have in your UMRK configuration for VPC is to generate two EIPs to ensure a persistent address. In the VPC console, choose the Elastic IPs sidebar menu option and then use the Allocate New Address button to open the dialog box shown in Figure 2-10.

Figure 2-10. Allocating a new Elastic IP address in your VPC

The AWS CLI command to allocate your IP address is a simple one-liner:

```
aws ec2 allocate-address --domain "vpc" --region us-east-2
```

Allocate your two EIPs for the example applications and then you are ready for launching your UMRK instances in the next chapter. You will recall from earlier that there is a quota limit of five EIPs per Region per customer by default which might trigger the need for you to request an increase.

Design Patterns for Availability with AWS VPC

Designing with operational patterns in at the outset ensures a greater chance of success for both security and networking in your VPC. Here are some important tips to follow:

Security taxonomy is important
: A good use of global, instance-specific, and shared Security Groups helps to reduce repetitive rule creation and confusion. You should regularly revisit your Security Groups to determine whether they match your operational model.

Audit your Security Groups regularly
: With multiple rules in place and the most permissive of them applying, it is easy to lose track of what is actually allowed or blocked as your environment scales. It is a good practice to regularly audit, review, and test your access rules.

Subnetting matters
: Network assignment in your VPC should be done with some planning. The allocation must align from the top-level subnet through to the AZs to allow for enough addresses for all potential instances. This might also include working with NAT and VPN back to your private infrastructure, which can create routing challenges.

Conclusion

This exercise took us through creating a new VPC using the built-in wizard, creating a new subnet, and learning how Security Groups will apply to a VPC. Now that you have taken in a lot of information about the AWS VPC configuration, it's time to move on to adding compute resources into your newly launched VPC.

CHAPTER 3

Amazon Web Services Elastic Compute Cloud

In this chapter, we examine the Amazon Web Services (AWS) Elastic Compute Cloud (EC2) using practical examples of deploying an EC2 instance, taking snapshots, creating images, and operational tasks through both the web console and the AWS command-line interface (CLI). These are important tasks that will be familiar to what you might have done in a virtualization environment.

EC2 Fundamentals

Working with Infrastructure as a Service (IaaS) implementations on AWS means using EC2 for your compute platform. This is where you build virtual machines (VMs) that can run nearly any operating system (OS). EC2 runs on shared underlying infrastructure by default but with strict security boundaries protecting resources from reaching one another between customers.

You do have the option to run dedicated instances that are allocated onto physical hosts, which are only provisioned with your EC2 workloads. Another option is to choose dedicated hosts. This lets you have more control at the physical layer and also lets you run licensed applications that have limitations around binding to specific CPUs. You might find this on-premises today if you run Microsoft Windows Server, Microsoft SQL Server, Oracle SQL Server, or other similarly licensed software.

You choose EC2 instance configuration and size based on family type (e.g., T3, C5, R5, H1) and then by what you might call "t-shirt size" (e.g., small, medium, large, xlarge, and 2xlarge). The instance type is not just about size, but also about the capabilities that it presents and, of course, the price per hour.

Scaling up your instances within the family type adjusts the virtual memory and virtual CPU allocated to the instance. It is important to note that changes to the instance size or family will require a restart of the instance to apply the new configuration.

General purpose (T3, T2, M5, M4)
: Various types in the family for general or "bursty" workloads that might not map to another instance type.

Compute optimized (C5, C4)
: Designed for more compute-centric workloads.

Memory optimized (R5, R4, X1e, X1, z1d)
: Higher-speed access to memory and higher memory to CPU ratio.

Accelerated computing (P3, P2, G3, F1)
: GPU-accessible option for workloads that can make use of the enhanced parallel processing power.

Storage optimized (H1, I3, D2)
: Low-latency and high-speed access to the storage subsystem for read/write–intensive workloads.

Reserved Instances

Reserved instances are available to buy as a precommit for one or three years with deep discounts (63% at three-year commitment) off of the hourly on-demand pricing. This is ideal for instances that will be online for a long period. You can pay fully upfront (largest discount), partial upfront (moderate discount), or no upfront (lowest discount) for an instance type.

Reserved purchases are made for a specific instance type in a specific Region, and the reserved discount is applied to active instances that match the size and location. This introduces the challenge of matching your active workloads to the best possible discounts.

Understanding Amazon Machine Images

You can save instances as an Amazon Machine Image (AMI) or launch one from there. This is helpful to speed the creation of your EC2 instances from either the AWS Marketplace or your own library of images.

AMIs are available from a catalog of default images that are either AWS Marketplace AMIs, Community AMIs, or from your own catalog of AMIs, which you can create from your EC2 instances. Keeping your instances as an AMI is a handy way to clone as templates, which makes it easy to launch multiple versions with prebuilt configuration and software installed.

A great example would be when you configure your application server to have specific libraries, security lockdowns, log export configuration, and some custom code that your dev team needs. Creating an AMI takes the fully configured live EC2 instance and makes it your base image, which you can use to launch other EC2 instances. It is similar to using VM templates in a virtualization stack.

Example: Deploying the UMRK Web Servers

The UMRK team needs two web servers, each deployed into a separate Availability Zone (AZ) within the same Region. Your task also gives you a chance to launch an EC2 instance running Amazon Linux. The Amazon Linux is a lightweight, secure, and versatile Linux derivative built and maintained by the AWS team, and it has many operational similarities to CentOS.

You start in the AWS EC2 console by launching a new instance through the wizard, as depicted in Figure 3-1, which takes you through the steps.

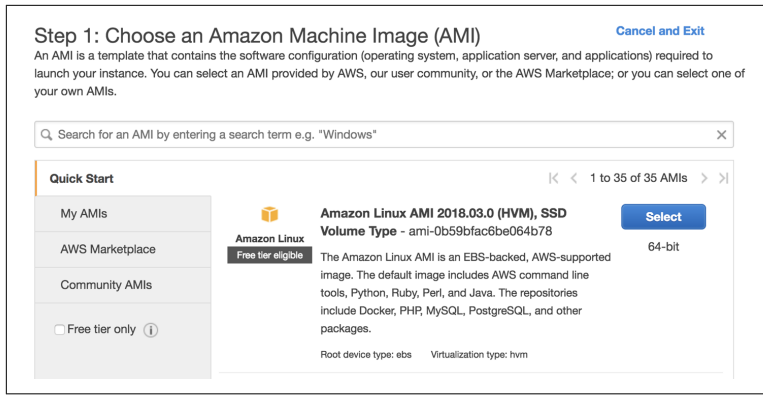


Figure 3-1. Start the EC2 wizard by choosing an image

There are a few details to choose from in the third step of the wizard, which include those highlighted via arrows in Figure 3-2. You must choose to assign a network that is associated to your Virtual Private Cloud (VPC) to prevent ending up in the default VPC. Your subnet will be chosen from the ones you created in Chapter 2. This example for UMRK disables the public IP in favor of using an Elastic IP (EIP).

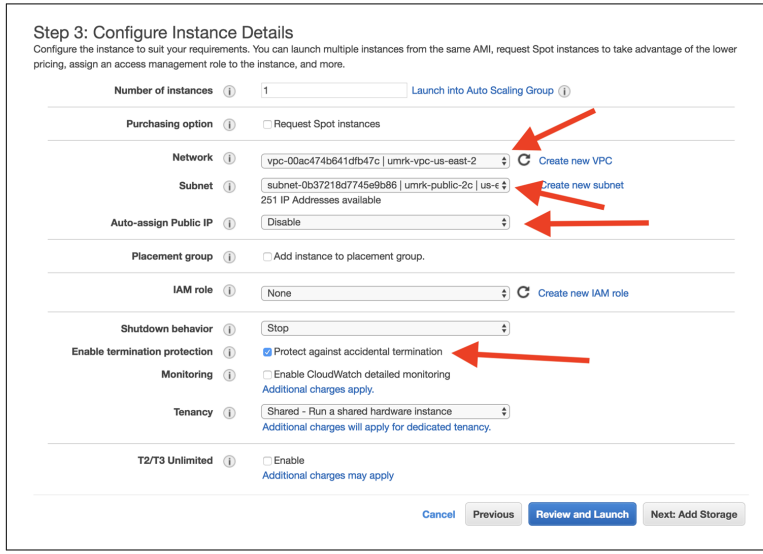


Figure 3-2. Choose the instance details

IAM roles allow for service-to-service and more granular administrative access to resources. We are not covering IAM roles in this guide, so for this example, we are not using an IAM role.

NOTE

It's a good idea to select the "Protect against accidental termination" checkbox to reduce the risk of accidentally terminating and losing your instance. You can disable termination protection later in the EC2 instance details.

Set "Shutdown behavior" for the UMRK example to Stop rather than Terminate to ensure that we don't kill the instance if it is stopped.

The requirements for UMRK include application code, which the developers will want on persistent storage. Figure 3-3 shows choosing a 20 GB volume for the example (default is 8 GB). You should also choose the General Purpose (SSD) option for Volume Type. More detail on the volume types is available in the next chapter.

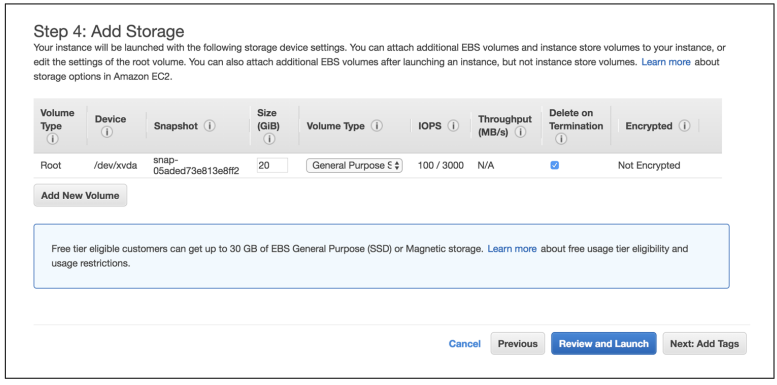


Figure 3-3. Assigning a 20 GB root volume

Tagging resources helps to identify them visually and programmatically. This helps for things like grouping, searching, chargeback/showback ownership, and also helps for many third-party products that you might use with AWS that use tags. Figure 3-4 shows three tags that assign an owner, an application, and an environment label on both the EC2 instance and the associated volume.

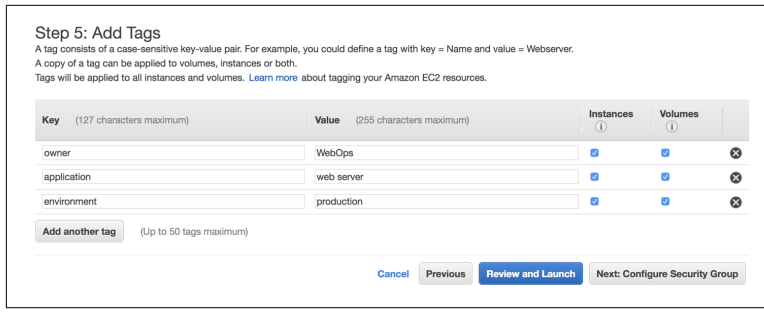


Figure 3-4. Assigning your EC2 instance tags

Every instance creation wizard will default to creating a generically named new rule. You should assign a meaningful name similar to what Figure 3-5 illustrates. You also can add existing Security Groups, which is what you will use in the UMRK example when creating your second instance.

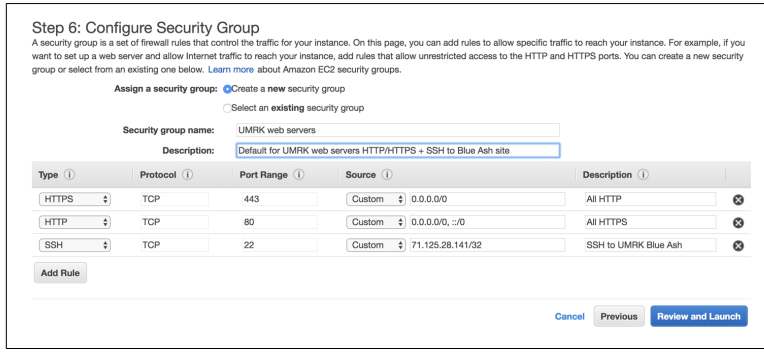


Figure 3-5. Configuring the UMRK security group rules

The following CLI examples show you how to deploy an EC2 instance using the AWS CLI to perform the same task, which requires only the AMI ID of the image from which to launch. You will use the ami-0b59bfac6be064b78 from the same example used in the AWS Console earlier in Chapter 2. You also need to know the name of the SSH key you have uploaded to the Region to which you are deploying.

Get your SSH key name

```
aws ec2 describe-key-pairs
```

Create Security Group

```
aws ec2 create-security-group --description
  "UMRK Blue Ash" --group-name "UMRK HTTP and
  SSH Group" --vpc-id vpc-006960a6c4d805f10
```

Create a rule to allow HTTP on TCP

```
aws ec2 authorize-security-group-ingress --group-id
  sg-01a304e61f20427d9 --protocol tcp --port 80
  --cidr 0.0.0.0/0
```

Create a rule to allow HTTPS on TCP

```
aws ec2 authorize-security-group-ingress --group-id
  sg-01a304e61f20427d9 --protocol tcp --port 443
  --cidr 0.0.0.0/0
```

Create a rule to allow SSH from a single IP address

```
aws ec2 authorize-security-group-ingress
  --group-id sg-01a304e61f20427d9 --protocol tcp
  --port 22 --cidr 71.125.28.141/32
```

Launch EC2 instance

```
aws ec2 run-instances --image-id ami-0b59bfac6be064b78
  --subnet-id subnet-0b37218d7745e9b86
  --key-name discoposse-macbook
  --instance-type t2.micro
  --security-group-ids sg-03b44aa884493f3f8
  --block-device-mappings
  DeviceName=/dev/sdh,Ebs={VolumeSize=20}
  --tag-specifications 'ResourceType=instance,
  Tags=[{Key=owner,Value=WebOps},
  {Key=application,Value="umrk web"},
  {Key=environment,Value=production}]'
```

That takes care of the launch of the EC2 instance, attaching the Security Group, and creating the 20 GB root volume.

Creating the Second UMRK EC2 Instance

You create the second instance similarly except on the second subnet that you created when configuring the VPC, and you use the existing Security Group to assign the access rules instead of creating a separate one. The AWS CLI command will require a different alternate subnet ID, as well.

Associating Your Elastic IP Addresses

The final step for enabling public access on a persistent IP address is to associate your EIP from the VPC console. Figure 3-6 shows the association wizard which maps the available EIP to an instance and assigns it to a network interface.

Use the Reassociation checkbox if you want to force the EIP to be associated to this instance even if it is already associated to another resource. In this case, it's good to leave this unchecked to make sure the EIP is free to be used.

Figure 3-6. Associating an available EIP to an EC2 interface

The AWS CLI command for this step requires the allocation-id from your EIP, the instance ID from your EC2 instance, and the private IP address that you are going to assign the EIP to on your internal network, plus your Region. Here is an example:

```
aws associate-address
    --allocation-id "eipalloc-06ecee9d2670d11e1"
    --instance-id "i-01f0026ea5396f197"
    --no-allow-reassociation --private-ip-address "10.0.3.179"
    --region us-east-2
```

Conclusion

Now that you have learned about EC2 compute and the basics of spinning up your cloud instances within a VPC, you are ready to learn about how persistent block storage is provided on AWS.

CHAPTER 4

Amazon Web Services Elastic Block Storage

This chapter explores the Amazon Web Services (AWS) Elastic Block Storage (EBS) platform. You will learn which types of storage are available, compare the cost and performance impact with EBS storage types, and look at practical examples of operational tasks with EBS using both the web console and the AWS command-line interface (CLI).

Block storage for EC2 instances and other systems needing block volumes will be provided by EBS. You can attach these volumes to instances, as well as clone, archive, and even detach and reattach them to other machines. Block storage is needed in many cases for Infrastructure as a Service (IaaS) instances. It's important that you understand the costs and performance impact of choices for EBS tiers.

Storage Tiers/Types in EBS

EBS volumes are all scalable to 16 TB at a maximum, and each type comes in four storage performance tiers, which vary in features and functionality. Choosing which type is important because this affects the cost, performance, and scaling of the storage, plus it must match the EC2 instance type to ensure that you are getting the most out of your storage.

gp2
: EBS General Purpose SSD—default when launching EC2 instances—good for boot volumes, general mid-performance volumes, and active data volumes.

io1
: Provisioned IOPS SSD—guaranteed input/output operations per second (IOPS) with a maximum of 32 K IOPS—good for database or high-volume data storage.

st1
: Throughput Optimized HDD—high throughput with maximum 500 IOPS—good for lower-access frequencies, but larger files are good. Can be used for some boot volumes on lower-demand instances.

sc1
: Cold HDD—"slow and low" with a maximum of 250 IOPS—good for archival data that must be on block storage.

You can make storage tier changes after initial deployment. This is important because you might find that your compute workloads require different performance and throughput over time. Making the initial move to high-performance storage is a pricey way to avoid the problem of changing storage tiers down the road.

EBS storage is charged at an hourly rate per gigabyte per month for the allocated amount. This is sometimes confusing because many cloud resources are thought to be pay-for-what-you-use when it is actually pay-for-what-you-allocate. A 200 GB EBS volume with 5 GB of data will be charged as 200 GB per month for the Region in which it is located.

Understanding EBS Snapshots

EBS has the ability to create point-in-time incremental snapshots of an existing EBS volume. These snapshots are stored only as the incremental changes (see Figure 4-1) since the last snapshot was taken, which helps reduces the cost and size. You can spawn another volume or instance using a snapshot, which creates a fully inflated live storage volume from the cumulative snapshots.

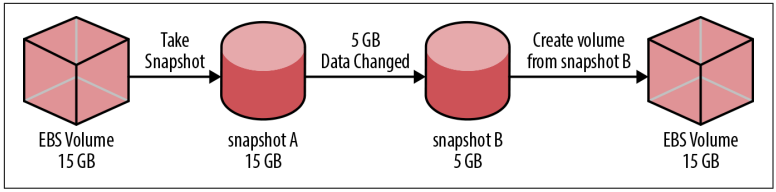


Figure 4-1. Snapshot example of an EBS Volume

Taking a snapshot is done easily from the EBS view, as shown in Figure 4-2, by selecting a volume and using the Create Snapshot action. You are prompted for a snapshot description and then can view active snapshots in the EBS view under snapshots.

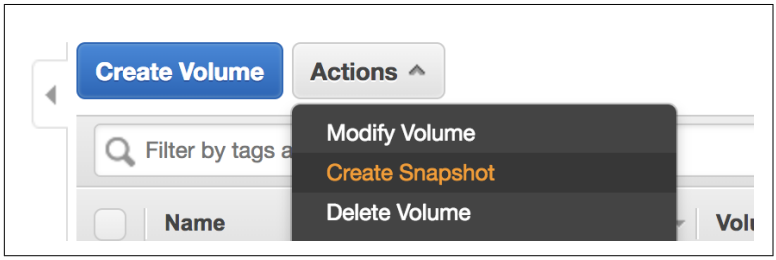


Figure 4-2. Creating an EBS snapshot in the AWS console

The associated AWS CLI is quite simple, needing only the `volume-id` and a description, which is more for readability and searchability later.

```
aws ec2 create-snapshot
--description "umrk pre-launch web snapshot"
--volume-id vol-024e8fc8350b3add0
```

Managing the UMRK EBS Volumes

Your UMRK web servers must each have additional EBS volumes, which will store the data used by the WebOps team to carry out future upgrades of its custom applications.

Open the EC2 console and then click the Create Volume button located in the Elastic Block Store | Volumes section of the page. Follow the prompts shown in Figure 4-3 to choose a size (100 GB in the example), the Availability Zone (must be located with the EC2 instance due to iSCSI network requirements), and create any tags that are relevant for the volume.

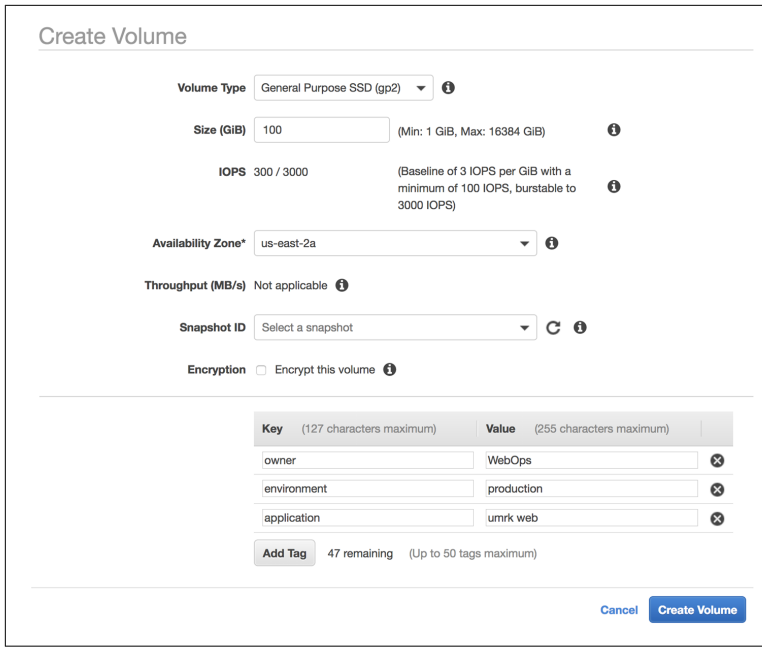


Figure 4-3. Creating the EBS volume in the AWS console

The AWS CLI to do this same task is shown in the following code example, which shows your AZ, volume size, volume type, and the tags:

```
aws ec2 create-volume --availability-zone us-east-2c
--size 109 --volume-type gp2
--tag-specifications 'ResourceType=volume,
Tags=[{Key=owner,Value=WebOps},
{Key=application,Value="umrkweb"},
{Key=environment,Value=production}]'
```

Figure 4-4 demonstrates attaching the volume, which you do in the Volumes section of the EC2 console.

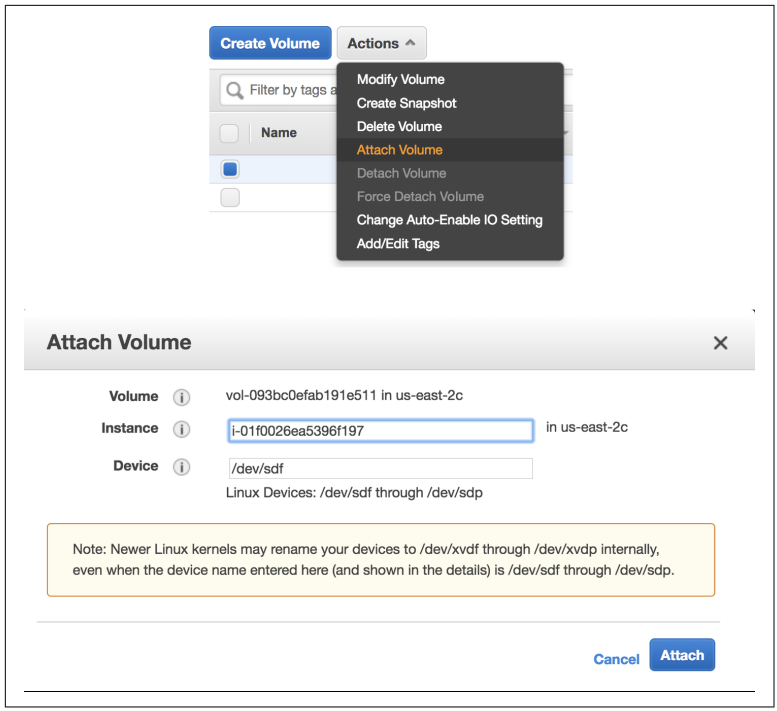


Figure 4-4. Attaching the new EBS volume to an EC2 instance

The associated AWS CLI requires only your `device` name, `volume-id` of the EBS Volume and the `instance-id` for your EC2 instance.

```
aws ec2 attach-volume --device /dev/sdf --volume-id
vol-041b8dcb66cdbb40b --instance-id i-01f0026ea5396f197
```

You now have your new volume created, presented, and attached to the EC2 instance.

Design and Operational Patterns for Availability Using EBS

With EBS, there are not as many design patterns as there are operational patterns to consider, aside from those that require understanding the storage needs of your cloud workloads:

Choose your default carefully
: The default is gp2 for new instances, which might be more than you require for I/O and also for cost. You can carry out storage-

tier migrations nondisruptively, which is helpful to reduce outages during future changes.

Manage snapshots carefully
: Snapshots are not backups! Be sure that you understand the point-in-time nature and what you want to use snapshots for. You should use them for short-term protection ideally and then remove them to reduce overhead and cost.

Backup your volumes and instances
: Data is still vulnerable to loss, corruption, and external access or even malware. It is critical that you have a backup and recovery strategy for your AWS data and applications. Contact your data protection vendor to make sure that you are licensed and able to protect your cloud assets.

Beware of cost versus performance decisions
: Choosing lower-cost volumes can affect performance but choosing higher-throughput volumes could be expensive and unnecessary. Performance will also change over time with application and consumption pattern changes. These are not one-time decisions.

Match storage to instance capabilities
: Make sure that your high-throughput storage is attached to the right instance type to fully use the performance features (e.g., EBS optimized).

Conclusion

Volume management is quite simple on AWS through the AWS console as well as the CLI. You now have completed the requirements for our UMRK example and have a solid foundation to start operating your own AWS instances and volumes.

Next Steps in Your AWS Journey

This guide has been created to give some specific examples of core IaaS service use on AWS and a general coverage of the compute, storage, and networking for AWS features. This is only the beginning of your journey to learning AWS. The next steps are to define what your use-cases and goals are for AWS for personal and work purposes.

If you would like to seek additional learning, resources, detailed code examples, and technical certification, there are resources available in the accompanying website.