O'REILLY®

Compliments of Fugue

# Immutable Infrastructure

## Considerations for the Cloud and Distributed Systems

Josha Stella

Fugue

# Immutable Infrastructure

*Considerations for the Cloud and Distributed Systems*

*Josha Stella*

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

**Immutable Infrastructure**
by Josha Stella

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

**Editor:** Brian Anderson
**Production Editor:** Nicholas Adams
**Copyeditor:** Rachel Head
**Proofreader:** Nicholas Adams
**Interior Designer:** David Futato
**Cover Designer:** Randy Comer
**Illustrator:** Rebecca Demarest

February 2016: First Edition

**Revision History for the First Edition**
2016-01-27: First Release

978-1-491-95080-7

[LSI]

# Table of Contents

# Acknowledgments

While I'm the author of this report, it was really a team effort. Credit goes to Dan Kerrigan, a senior engineer at Fugue for the detailed examples, to Henry Harding, our director of design for the diagrams, and especially to Racquel Yerbury, our lead for publications for taking my ideas and making them into a finished product. Other team members lent a hand too and everyone's efforts are much appreciated.

Our editor Brian Anderson and the other talented professionals at O'Reilly could not have been more helpful and supportive—thank you for great ideas, critique, and encouragement. Finally, sincere thanks to our external reviewers for your time and thoughtful input.

*– Josh*

CHAPTER 1

# Here Then Gone: What Is Immutable Infrastructure?

You're standing on the beach on a bright day. You look out. There's a constant renewal of pointed, flashing light, here then gone, conveying energy, then ceasing to exist without any consequential decay. The sun automates an optical phenomenon on water in motion: *glitter patterns* that comprise millions of ephemeral *glints*. In applied optics, physicists who study the properties of light have long marveled at the Phoenix-like effect we see. Back on the beach, other glints are immediately visible, carrying out the same energetic tasks, then gone. No decay.

It's an imperfect, but provocative, analogy: machines in a data center seem like a far cry from points of natural, strobed light—not least because we relate to them as physical items rather than as organized energy, as long-lived rather than as ephemeral. We tend to rack machines in an n-tier framework in our minds to a greater or lesser degree, instead of thinking in terms of distributed, abstracted instances or resources capable of spanning multiple availability zones in cloud computing. But when infrastructure becomes code, resources are, in fact, more akin to those glints on the sea than to dedicated boxes.

## Toward Cloud Thinking

For decades, we've mulled over basic questions around how we provision machine resources—and those questions are under new scrutiny with cloud computing. The techniques we've traditionally used to manage machines struggle in distributed, scaled environments. Historically, we've thought of machine uptime and maintenance as desirable complements because we associate them with the overall health of a service or application. But cloud computing lends itself to a substantially different model of health. You drill into a more granular kind of abstraction where component *replacement* makes more sense than traditional server maintenance. Think about this contrast:

- *In the data center*, infrastructure is expensive and we need to carefully craft and maintain each individual server to preserve our investments over time.
- *In the cloud*, infrastructure and services are an API call away. A new architecture calls on us to give up the data center mindset in order to create more resilient, simpler, and ultimately more secure services and applications.

Werner Vogels, CTO of Amazon and an early leading thinker on cloud systems, captures this sentiment by imploring us to stop hugging servers—because they don't hug us back. His famous "server as a paper cup" analogy, like our analogy with a striking pattern in optics, is conceptually useful as we dig in and wrap our minds around the vast potential of the cloud and the new implementations of infrastructure it enables.

The premise here is that *immutable infrastructure*—infrastructure that is replaced rather than maintained—offers a real and attainable path to stability, efficiency, and fidelity for your applications in the cloud.

## Immutable Infrastructure in Brief

No rigorous or standardized definition of immutable infrastructure exists yet, but the basic idea is that you create and operate your infrastructure using the programming concept of immutability. That is, once you instantiate something, you never change it. Instead, you replace it with another instance to introduce changes or ensure proper behavior. Immutability doesn't mean that the overall system never changes. In fact, using immutable infrastructure can make modifying the system easier, faster, and more reliable at scale because configuration drift can be substantially reduced. Immutable infrastructure also doesn't require a fully stateless application, but it is best suited for distributed applications that concentrate persistent state in few locations.

Today, immutable infrastructure can be realized in cloud environments that cover compute, storage, networking, access control, and other objects needed to compose the application. For many large, complex deployments involving multiperson teams, full automation of the runtime environment is beneficial, since immutable infrastructure treats all aspects of a system as quanta that can be built, replaced, and destroyed as part of the regular operations of the system. This is facilitated in compute environments that have an API over all aspects of configuration and monitoring. Partial implementation of immutable infrastructure has its benefits, but the big improvements in efficiency and resiliency are realized with thorough implementation. Major public cloud infrastructure providers, including Amazon Web Services (AWS), Google Cloud Platform, and Microsoft Azure, offer APIs over their services, and more private cloud solutions are starting to offer the necessary automation.

As the industry is full of buzzwords, the definition of immutable infrastructure outlined in this guide is an attempt to sharpen parameters around the concept and give the term practical meaning. Take a look at the visual presented in Figure 1-1, which we'll reference and drill into as we go through the chapters.

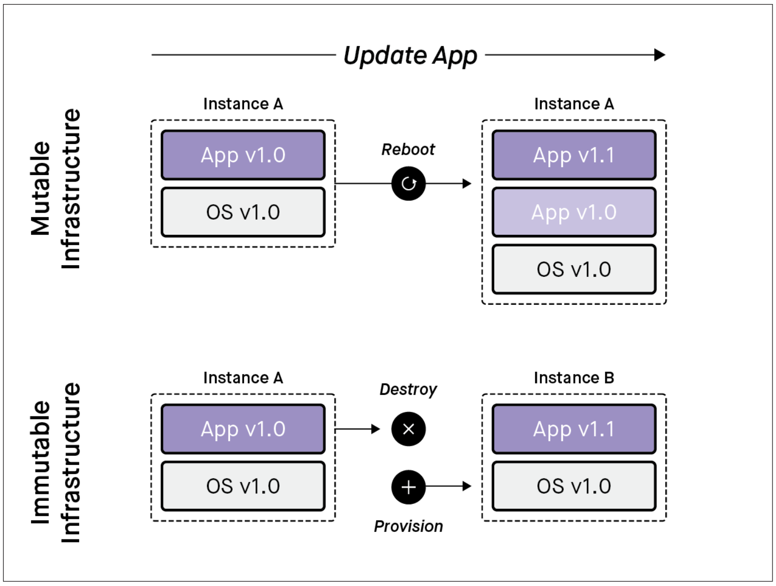

*Figure 1-1. Updating via mutable vs. immutable infrastructures*

Notice that the new Instance B, generated from a "golden" machine image, is provisioned upon the destruction of Instance A in the immutable pattern. Note too that there is no application downtime during instance replacement with well-architected immutable patterns that have multiple instances in service at a given time. By contrast, in the mutable pattern, Instance A isn't replaced. The same instance is modified manually or by using a script or tool, with the application updated from v1.0 to v1.1. The update might include changes in application code, configuration, underlying libraries, combinations thereof, etc.

## Coining the Term

Chad Fowler published the term "immutable infrastructure" in a June 2013 blog post entitled "Trash Your Servers and Burn Your Code: Immutable Infrastructure and Disposable Components." He explained it this way:

> Many of us in the software industry are starting to take notice of the benefits of immutability in software architecture. We've seen an increased interest over the past few years in functional programming techniques with rising popularity of languages such as Erlang, Scala, Haskell, and Clojure. Functional languages offer immutable

> data structures and single assignment variables. The claim (which many of us believe based on informal empirical evidence) is that immutability leads to programs that are easier to reason about and harder to screw up.
>
> So why not take this approach (where possible) with infrastructure? If you absolutely know a system has been created via automation and never changed since the moment of creation, most of the problems [...] disappear. Need to upgrade? No problem. Build a new, upgraded system and throw the old one away. New app revision? Same thing. Build a server (or image) with a new revision and throw away the old ones.

Chad is in good company with others who have thought deeply about the subject. As Martin Fowler wrote in a July 2012 blog post titled "PhoenixServer":

> [ ... ], it is a good idea to virtually burn down your servers at regular intervals. A server should be like a phoenix, regularly rising from the ashes.
>
> The primary advantage of using phoenix servers is to avoid configuration drift: ad hoc changes to a systems configuration that go unrecorded. Drift is the name of a street that leads to Snowflake Servers, and you don't want to go there without a big plough.

And back in August 2011, Greg Orzell pointed out in a Netflix Tech Blog post called "Building with Legos" that:

> In the cloud, we know exactly what we want a server to be, and if we want to change that we simply terminate it and launch a new server with a new AMI. This is enabled by a change in how you think about managing your resources in the cloud or a virtualized environment. Also it allows us to fail as early in the process as possible and by doing so mitigate the inherent risk in making changes.

Kief Morris, James Carr, and Ben Butler-Cole offered compelling early insights as well, in "Immutable Server Blikis" (kief.com, June 2013), "Immutable Servers with Packer and Puppet" (Rants and Musings of an Agile Developer, July 2013), and "Rethinking Building on the Cloud: Part 4: Immutable Servers" (ThoughtWorks, June 2013), respectively, as did many others. Morris's 2015 book *Infrastructure as Code* (O'Reilly) is also a significant, thorough contribution to the conversation.

Immutability itself is not exactly new in computing, with the ideas being explored and advocated as far back as the 1950s. Like most concepts in our business that seem new, immutable infrastructure has a rich, if esoteric, history. However, the technologies and products that make its use practical—particularly, the cloud and the highly modular services therein—have only recently been adopted on a broad scale. That's why the conversation has emerged so robustly in the past few years.

## Identifying Problems and Solutions in the Cloud

When we find ourselves in a changed context, like widespread cloud adoption, it's a function of human personality and ingenuity to scour the landscape for whatever we can use to adapt. We often start with what's familiar and use tools that are immediately at our disposal to work within the new world. They may be improvisations. They may be square pegs grappling with round holes. The next step is to move beyond this kind of forced fit. That is, we move beyond tools and approaches that we've forced to be compatible with cloud computing and, instead, we engineer ones specifically suited to our new cloud environment. These next-generation tools and approaches may integrate ideas from the past and present, but foremost they're crafted in a way that respects the fundamentals and the nuances of the new environment. They, not transitional adaptations, will guide full realization of cloud computing's power.

### Mutable Infrastructure Creates Problems

In our cloud computing context, mutable infrastructure *is* the transitional adaptation. It's the improvisation. It's the approach from a different, but very familiar environment—the data center—that we've forced to fit with the cloud. Mutable infrastructure, composed of traditional, long-lived components, is insufficient to the task of operating modern, distributed services in the cloud. The forced fit has created specific problems that don't need to exist. Let's look at them:

*Increasing operational complexity*
: The rise of distributed service architectures and the use of dynamic scaling result in vastly more maintenance and monitoring needs, much of this in response to changes in the runtime environment. Using mutable maintenance methods for updates via scripts and configuration management tools, or patching configurations across fleets of hundreds or thousands of compute instances, is complex and error-prone.

*Slower deployments, more failures*
: When infrastructure is comprised of "snowflake" components resulting from mutable maintenance methods, there's a lot more that can go wrong. (Snowflake components, per Martin Fowler, are symptomatic of "configuration drift: ad hoc changes to a systems configuration that go unrecorded.") Traditional deployment models require you to first bring up a machine, then apply patches, and then install the application code, each of which takes time and can fail to work properly. Deviating from a straight-from-source-control process means it is impossible to accurately know the state of your infrastructure. Fidelity is lost as infrastructure behaves in unpredictable ways. Time is wasted chasing down configuration drift and debugging the runtime.

*Difficulty identifying errors and threats*
: Long-lived, mutable systems rely on identifying errors and threats to prevent damage. We now know that this is a Sisyphean undertaking, as the near-daily announcements of high-profile and damaging enterprise exploits attest—and those are only the ones reported. Efforts are ongoing to make automated analytics tools smarter about honing in on anomalous patterns, but if history is a guide, defenses will continue to trail offenses.

*Fire drills*
: Mutable, long-lived infrastructure allows for shortcuts on automation that come back to bite us in unexpected ways, such as when a cloud provider reboots underlying instances to perform its own updates or patches. If we build and maintain our infrastructure manually and aren't in the regular routine of immutable infrastructure automation, these events become fire drills. That is, they require teams to be paged, rush to the office, and work all night or day to resolve what needn't be a problem.

## Immutable Infrastructure Provides Solutions

Now, let's turn to short-lived immutable infrastructure, which is *not* a transitional adaptation to the cloud. Rather, it's an approach fundamentally aligned with cloud technology. The problems that mutable infrastructure creates in the cloud are largely resolved by the solutions that immutable infrastructure provides. Those solutions include:

*Simplifying operations*
: With fully automated deployment methods, you can replace old components with new versions to ensure your systems maintain their initial "known-good" state. Managing a fleet of instances becomes much simpler with immutable infrastructure, since there's no need to track the changes that would occur with mutable maintenance methods.

*Continuous deployments, fewer failures*
: With immutable infrastructure, you know what's running and how it behaves. Deploying updates can become routine and continuous, with fewer failures occurring in production. All changes are tracked by your source control and Continuous Integration/Continuous Deployment (CI/CD) processes.

*Mitigation of errors and threats*
: Services are built atop a complex stack of hardware and software, and things do go wrong over time. By automating replacement instead of maintaining instances, we are, in effect, regenerating instances regularly and more often. This reduces configuration drift, vulnerability surface, and the level of effort required to meet service level agreements. Many of the situations that lead to maintenance fire drills in mutable systems are avoided with immutable infrastructure. Many errors and threats are mitigated whether they are detected or not. That mitigation becomes increasingly potent as the rate of resource replacement speeds from every day to every hour to every minute. It's also noteworthy that attacks on a high-refresh-rate immutable infrastructure system mean more log entries. This can aid evaluation by secops and forensics teams.

*Easy cloud rebooting*
: With immutable infrastructure, you know what you have running, and with fully automated recovery methods for your services in place, cloud reboots of your underlying instances should be handled gracefully and with minimal, if any, application downtime.

*Potential for reduced costs*
: If executed well, immutable patterns can result in reduced costs. The fundamental economic benefits of the cloud are in avoiding provisioning for imagined loads on the system. When using immutable patterns, which are typically fully automated, we

gain the ability to scale our infrastructure with the loads. There also can be reductions in personnel costs for deployment and maintenance, as fixing things in situ is considerably more difficult than simply replacing them. Tracking down partial deployments and sorting those takes time and costs money.

Maintaining hardware is difficult work, especially when that hardware is physical boxes in a rack that require manual configuration, not to mention maintenance of the building where they are housed. But with logically isolated compute instances that can be instantiated with an API call in an effectively infinite cloud, "maintaining boxes" is an intellectual ball and chain, tying us to caring about and working on the wrong things.

"Trashing your servers," as Chad Fowler would say, enables you to focus on what matters to the success of your application, rather than being constantly pulled down by high maintenance costs and the difficulty of adopting new patterns.

## Is Immutable Infrastructure the Right Fit?

Most non-cloud, mutable environments are actually not-yet-cloud, not-yet-immutable environments. That is, time moves fast in computing, and hybrid solutions will continue transitioning to cloud-based, fault-tolerant, API-driven ones. But right now, immutable infrastructure's core benefits of *predictability*, *reliability*, and *scalability* are more imperative to some businesses and organizations than others.

For example, a typical independent school won't have a dramatic rise or fall in its number of users. Even with sophisticated project and communication apps distributed in that environment and even with a school's need for security around grades and finances, an urgency for infrastructure upgrades is atypical and the tech shop on campus is likely small. The app businesses themselves, on the other hand, which serve that school and hundreds or thousands of others, have needs much more aligned with cloud computing and the ease of scaling that immutability provides (think of major spikes in homework app usage occurring after 5 p.m., with dramatically decreased usage in the early part of the day). *Very* large public school districts may have those needs as well.

Massive legacy systems, too, cause CIOs to balance competing priorities. A CIO with a system that doesn't run on industry-standard Intel hardware or that has unusual network requirements or uses lots of recently purchased hardware may find it's not the right time to migrate. Many enterprises have made the decision to leave legacy alone for the time being and, instead, build their *new* features on the cloud with fault-tolerant methods. It's worth noting that highly regulated, compliance-oriented industries, perhaps with classified data centers and access controls required to meet specifications, may or may not be candidates for cloud computing and immutable infrastructure.

So, what about you? Where do you fit?

*If you are running a traditional n-tier application* and have never felt the need for more than one or two servers, immutable infrastructure may sound like a solution looking for a problem. But if you have scaling needs, even modest ones, once you begin using some cloud-native architectural patterns, such as automating instance replacement or autoscaling up and down, you'll realize that immutable infrastructure is central to operating effectively at scale. Those patterns and others are explored later in this guide.

*If you are running a distributed system in the cloud*, you may have already adopted some or many of these patterns, but it's uncommon to find examples of full implementations.

*If you can say "yes" to the majority of these statements*, you may benefit from immutable infrastructure:

- You are creating a cloud-native application, as defined in *Cloud Architecture Patterns* (O'Reilly) by Bill Wilder.
- Your application architecture enables you to revise and deploy software changes on an ongoing basis.
- Your servers can boot in a "lights-out" or "headless" environment and be ready to do their task without human intervention.
- You plan to scale your application horizontally using a service such as Auto Scaling Groups (ASGs) on AWS, managed instance groups on Google Cloud Platform, or Scale Sets on Azure.
- You have a mechanism to do automatic updates or to roll out machine images.

- You want to make deployments simple and infinitely repeatable.
- You want to enable one-step deployment of entire infrastructures, including network configurations, application servers, and other resources.
- You hope to achieve continuous deployment; the more complicated your deployment workflow is, the more important it is to isolate state.
- You want to ensure that no on-the-fly changes are made to anything.
- You want to focus on your core business rather than operations.
- Your want your application to gracefully respond to node failures—immutable components can be easily regenerated.

In the next chapter, we'll move beyond definitions, benefits, and fit to pragmatics. Specifically, we'll sample the landscape and consider how immutable infrastructure can be implemented now, as part of complex toolchains. Then, we'll step into a best practices mindset, laying out key ways to think about and work with cloud resources. In doing so, we'll look at an infrastructure approach that realizes those practices, automating robust immutable patterns while significantly limiting complexity.

CHAPTER 2

# Immutable Infrastructure in Action

Using immutable infrastructure for the first time has a learning curve. It's a paradigm shift. But you don't have to go all in to reap the benefits. You can make parts of your infrastructure immutable. In this chapter, we'll:

- Discuss some tools and technologies that exist in the marketplace and provide example implementations using those. At present, the most popular cloud is Amazon Web Services (AWS), so we'll walk through an example in that context and also one that is non-AWS specific but illustrates a common use case. (See "Immutable Infrastructure in the Toolchain" on page 14.)
- Against that backdrop, we'll do a bit of "best practices" analysis, walking through good immutable infrastructure candidates in your existing and greenfield applications across compute, storage, network, and management services commonly available on clouds. (See "Best Practices: How to Make Your Application Immutable" on page 23.)
- As we consider normative patterns and modes of working with cloud resources, we'll illustrate a unified approach emerging in the immutable infrastructure landscape that relies on cloud OS modeling rather than customized toolchains. (See "Immutable Infrastructure in a Unified System" on page 29.)

For big, complex deployments involving multiple team members, integrating immutable patterns into your workflow involves significant automation and testing, but it pays off. The system will be efficient with its use of resources and resilient to infrastructure quality issues and human error.

## Immutable Infrastructure in the Toolchain

First, let's take a closer look at how toolchains can currently be customized to implement immutable patterns and what's accomplished with those customizations. Many teams are exploring how to do immutable infrastructure well. It's possible to put together a solution with a combination of tools like Ansible, CloudFormation, Docker, and Kubernetes, to name a very few, but you'll need to write a lot of glue code and custom scripts. You might develop an automated workflow that integrates instances with cluster management, schedulers, and other open source or commercial products. Dozens of paths to implementation are possible.

To stay within the scope of this guide, we'll address a couple of use cases featuring tool types that have garnered attention. We'll also look briefly at the hard-won, ever-evolving Netflix success story and at types of components beyond instances and containers that might come into play as immutable patterns become more prevalent.

### Implementing with Containers

Containers allow developers and DevOps to produce full OS images containing the application code, supporting libraries, operating system kernels, and patch levels as one highly convenient, downloadable package. CI/CD systems can stamp, version, and track these OS images destined for containers throughout their lifecycles. This allows for a high level of consistency across images in large deployments with a strong level of confidence of correctness.

Because Docker allows for environment variable injection (not unlike user-data scripts for EC2 in AWS) into container images at boot, applications can grab the data they need from the orchestration system instead of configuration files. And because the configuration files no longer need to set application state, images within containers do not require configuration management tools like Chef, Puppet, and Ansible. This eliminates the need for "moving parts" within an image—container images are read-only and don't need to change.

With the combined benefits of image stamping, image versioning, environment variable injection, removing configuration files, and containers' fast boot times, CD systems can stand thousands (or even tens of thousands) of identical images at once via containers, knowing they are all identical. When a new patch or version comes out, the orchestration system floods out another versioned, stamped, and tracked image for the system to run. When a change must be made, rather than currently existing read-only data being modified, new data is supplied and the old data is discarded. Should a piece of infrastructure change, the orchestration system copies the small changing portion of the system, injects something new, and reattaches links or ports to the infrastructure staying behind.

Figure 2-1 details an example implementation for executing immutability with containers. Tools change and improve over time. This is just one feasible implementation—of hundreds possible—in this moment of history.

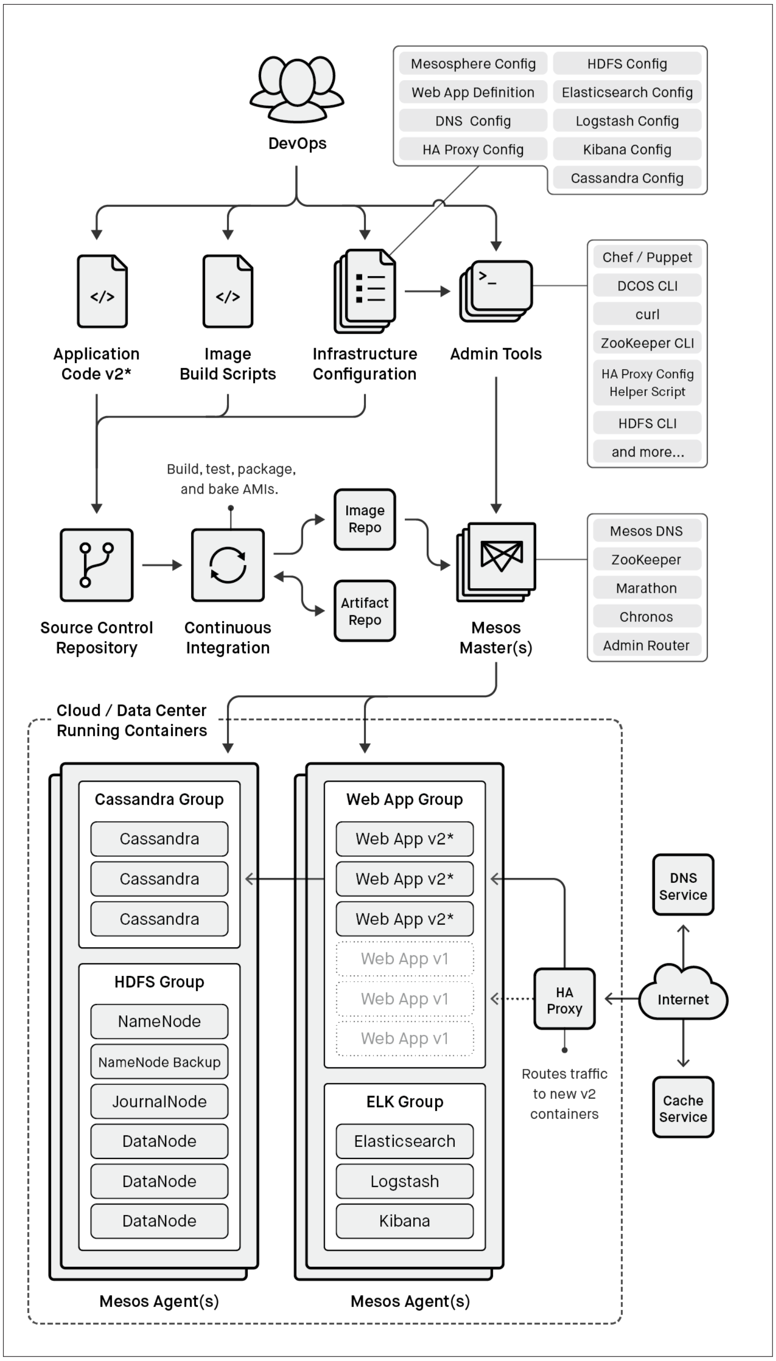

*Figure 2-1. Use case—immutable patterns with containers*

Notice here that web app containers are replaced via scheduled rolling restarts of the application group using Marathon and Chronos. Marathon is used for scheduling container placement and scaling within the cluster. Chronos is used for scheduling recurring actions, such as maintenance, or one-time actions, such as scheduling a rolling restart when an application container is updated. Chronos also can facilitate immutability by triggering a rolling restart periodically. ZooKeeper is responsible for runtime configuration and potentially service discovery. Cassandra is used for dynamic web content, and HDFS is used for large object storage and retrieval. In a production cluster, care should be taken to ensure node separation for individual Cassandra and HDFS nodes. Static web content is served using an external cache service/content delivery network (CDN). An ELK stack (Elasticsearch, Logstash, and Kibana) is used to manage application and Mesos cluster logs. An HA Proxy cluster is used to proxy and load balance incoming requests to the responsible web application containers. HA Proxy can run within Mesosphere, but we are minimizing potential DNS updates by running it externally to Mesosphere.

As noted, this is but one possible implementation of very many; it's included here in order to give explanation beyond lists and high-level discussion, which are often characteristic of this topic in media, and to provide clarity and detail with respect to how immutable infrastructure can get done. Much more can be said about containers, and indeed, much more is in the vibrant publications and conferences in our field. This is not an exhaustive discussion!

It *is* important to recognize that containers bring immutability to the application layer—you might orchestrate containers with Terraform and Consul to achieve immutability at that layer instead of some of the tools used in our implementation, or instead of dozens of other methods. But, regardless of implementation, containers don't generally address the underlying hosts and networks. Immutable infrastructure became popular as companies built and moved systems to the cloud, long before containers were the trend. Auto Scaling Groups (ASGs) and Amazon Machine Images (AMIs) used in concert are the most common manifestation of immutable patterns in the industry today. ASGs have been around and in use since 2008. While it's true that you can now get some of the cloud features that made immutable patterns popular by using containers in a traditional data center, this is not the most common use case, nor is it the primary driver for adoption of the patterns.

## Implementing with Machine Images

Machine images are the master blueprints for the compute instances you run in a virtual environment, whether locally or in the cloud. They are of particular importance in the cloud and with immutable patterns, as you'll be automatically building many instances with them and won't be logging in and making changes, as is common in mutable environments. Building machine images can be done in many ways, but you'll want to integrate the image build into your deployment toolchain.

Take a look at the example implementation illustrated in Figure 2-2. Here, we build machine images (e.g., AMIs in AWS) and deploy them on instances with ASGs to achieve immutability.

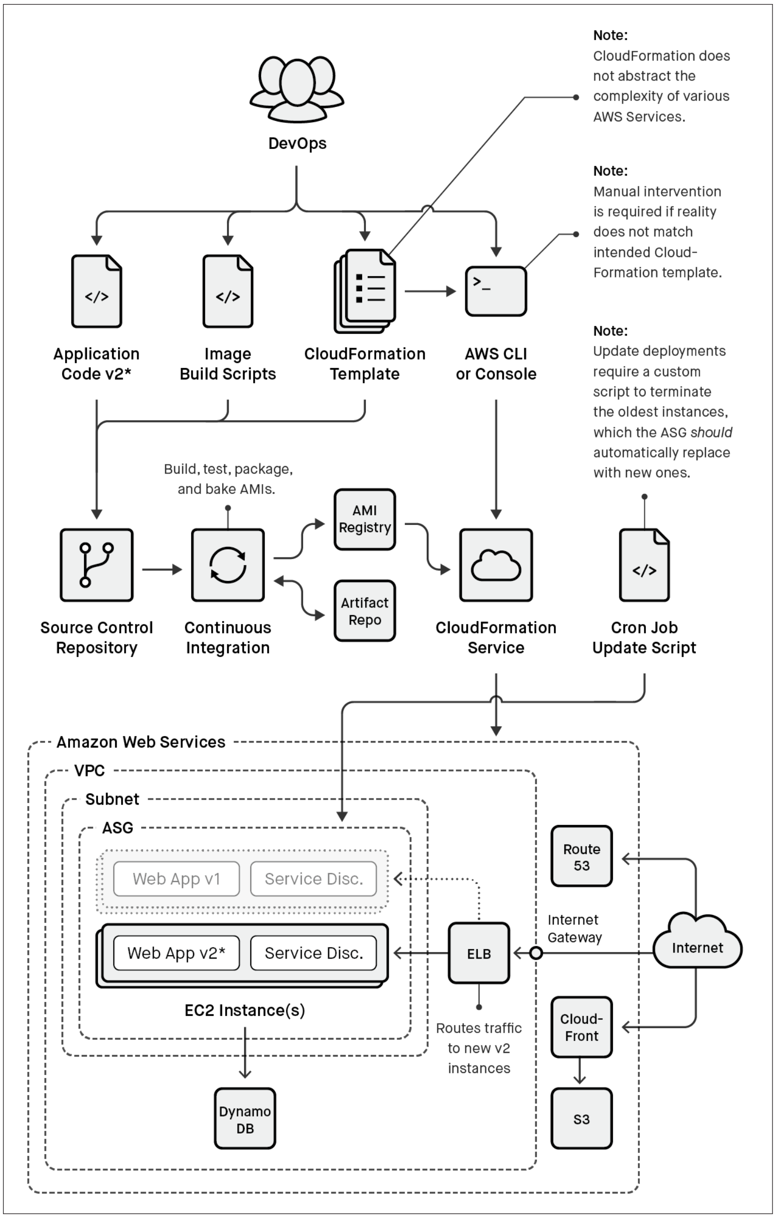
Note:
CloudFormation does not abstract the complexity of various AWS Services.
DevOps
Note:
Manual intervention is required if reality does not match intended Cloud-Formation template.
Application Code v2*
Image Build Scripts
CloudFormation Template
AWS CLI or Console
Note:
Update deployments require a custom script to terminate the oldest instances, which the ASG *should* automatically replace with new ones.
Build, test, package, and bake AMIs.
AMI Registry
Artifact Repo
Source Control Repository
Continuous Integration
CloudFormation Service
Cron Job Update Script
Amazon Web Services
VPC
Subnet
ASG
Web App v1
Service Disc.
Web App v2*
Service Disc.
EC2 Instance(s)
ELB
Routes traffic to new v2 instances
Internet Gateway
Route 53
Internet
Cloud-Front
S3
Dynamo DB

*Figure 2-2. Use case—immutable patterns with machine images on instances*

In this implementation of immutable infrastructure, web app instances are regenerated by terminating instances with the old application version manually and allowing the ASG to start instances with the new version. AWS doesn't provide a specific facility for runtime configuration or service discovery, but a combination of existing AWS services could be used (specifically, EC2 instance data). DynamoDB is used for dynamic web content. S3 is used for large object storage/retrieval and in conjunction with CloudFront. Static web content is served using CloudFront. Elastic load balancing (ELB) is used to load balance incoming requests to the responsible web application instances. DNS services are provided by Route 53. CloudWatch is used to aggregate logs.

The AWS CloudFormation service allows users to control most aspects of an AWS deployment. Once a CloudFormation template is created and submitted to the CloudFormation service, CloudFormation proceeds to instantiate parts of the template, stopping only on completion or exception. In the case of the latter, the user is responsible for determining the proper remediation steps.

We said that the web app instances are terminated manually in this implementation, while an ASG is used to start instances carrying the new app version. "Manually" means using an external script in the build, so here, a cron job script is run to terminate old instances, which the ASG should automatically replace. A script like this has to update launch configuration and associate it with the new ASG. Scripts have to deal with timeouts, API details, failed instance launches, registering instances with the ELB, and health checks, among other things. None of that is trivial. If instances are turned off at the wrong time, with large numbers, that's costly. Keep in mind too that we've either baked database configuration into the AMI or can inject it at launch time via user data and a service discovery tool like ZooKeeper or etcd. Service discovery is an important topic. Let's spend a moment delving into it.

In a highly automated environment, different infrastructure components may be coming up and going down constantly. These components frequently have some amount of runtime configuration that's dependent on the state of their environment. For example, a web server may need to know the hostname or IP address of a database server. Since the environment is changing constantly, these configuration items cannot be static or hardcoded. One way to solve this problem is with a service registry. This is a highly available datastore where components can register configuration for other components to consume. For example, a cache server may come up and register its IP address and available capacity with a service registry so that application servers can make use of it. Depending on the capabilities of the service registry, it may also provide other services, such as acting as a lock server.

## Chaos Engineering with Netflix

Netflix made the decision to go all in with cloud computing and crafted one of the most resilient, demanding systems with global reach—using immutable infrastructure—in operation today. Over the last decade, its teams have developed and fine-tuned a complex but highly effective microservices architecture, the design of which Director of Operations Engineering Josh Evans likens to a living organism. The system's resilience has been bolstered by dedicated implementation of chaos engineering, which Evans defines as "the discipline of experimenting on a distributed system in order to build confidence in the system's capability to withstand turbulent conditions in production."

Not only does Netflix regularly destroy production instances across availability zones (AZs) and regions and use red/black canary releases for new features, but it also now uses fault-injection testing (FIT) to simulate failures of whole services. In order to operate in this way, each component of the infrastructure must be immutable, as they are automatically generated, destroyed, and regenerated to handle instance and service degradation and outages. This is probably the most extensive public implementation of immutable infrastructure in the market.

Netflix engineers use machine images and a mixture of their own autoscaling and AWS's, among other tools. They spin up many services that talk to each other point-to-point and synchronously, as opposed to using something like Amazon's Simple Queue Service (SQS). They've found that their approach scales better for them. This means that to use their patterns, or other container or image-based patterns for that matter, you'll need some way to store the data that the instances need to find each other—i.e., a service registry, as discussed in the previous section.

Netflix engineers have written much on their Tech Blog, and Nicholas Whittier continues to report on Netflix's usage patterns and its open source tools for executing immutable infrastructure, with more forthcoming. Indeed, Netflix has released some great open source tools for doing immutable infrastructure on AWS. It's a bit of a heavy lift to set up and use, but if your business has enough in common with Netflix's from an operations perspective, its model can be beneficial.

## Implementing with Unikernels and Lightweight VMs?

Unikernels and library OSs are cutting-edge technologies that potentially can provide a complete deployment solution since they address fundamental problems, rather than trying to paper over them. Those problems include large image sizes that yield slow deployments, unnecessary code and features that take up memory and expose attack surfaces, and inconsistent environments. There are subtle differences between the unikernels and library OSs, but for our purposes, we'll reference unikernels to limit repetition.

Anil Madhavapeddy, Richard Mortier, et al., who created the Mirage prototype compiling OCaml code into unikernels that run on commodity clouds, described them this way:

> Unikernels are single-purpose appliances that are compile-time specialised into standalone kernels, and sealed against modification when deployed to a cloud platform. In return they offer significant reduction in image sizes, improved efficiency and security, and should reduce operational costs.

Thus, a unikernel is the parts of the OS that are needed to run a service, along with the service itself, and that's all. It's constructed in a compilation/build process that makes a simple virtual appliance for the service you'd like to run. A unikernel is radically smaller and lighter than a traditional OS and application, and it limits the attack surface to the application if used well. Unikernels are a natural fit for immutable patterns, but there's a catch that likely will push adoption out a few years: the unikernel OS needs to directly support the language and features with which you are programming. So, at present, they don't provide the kind of coverage for existing applications that traditional OSs on virtual machines or containers do. They can boot incredibly quickly, though, and they're generally much lighter than even containers. It's a good idea to watch this space develop. Madhavapeddy spoke about research and work concerning immutable distributed infrastructure with unikernels in the summer of 2015. Russell Pavlicek of the Xen Project sees unikernels as part of "the next-generation cloud," and Lars Kurth has discussed early adopters working with R&D partners.

Another intriguing technology to keep an eye on is the very lightweight VM image possible with OSv and Capstan, which can be used in lieu of containers. The documentation indicates that the build yields “a VM image that can run on any hypervisor with OSv support and is only 12–20MB larger than an application itself, with about three seconds added to the build time.” Designed for “private or public cloud using administration tools of choice,” its potential to be integrated within an immutable infrastructure pattern is significant.

What the industry is trying to achieve is a lightweight, and therefore fast, deployment package. The multiuser OSs we’ve grown accustomed to aren’t intended for what we are doing with them—they carry baggage from serving as interactive computers that are long-lived. Unikernels and lightweight VM images (like the one mentioned here) provide opportunities for deployment experimentation, testing, and eventually production.

## Best Practices: How to Make Your Application Immutable

In the example implementations we’ve covered so far, developers ideally would set up systems that consider fundamental patterns for immutable infrastructure and that follow best practice design considerations. In this section, we’ll look at those patterns and considerations. Since most cloud computing is done with compute instance type services, such as AWS EC2 or Azure virtual machine instances, to allow choices about performance characteristics we will focus on those.

### Fundamental Patterns

Immutable infrastructure on cloud compute resources has some fundamental patterns that should be followed:

- *Don’t modify instances in place.* This is the core pattern in immutable infrastructure. Modifying server instances is difficult to successfully automate and track, and allowing it will result in manual modifications. While in theory it is possible to have policies and tools in place to keep things consistent, in practice, both fail with some regularity. There is no concise and trustworthy way to know if and where you have configuration drift.

- *Replace instances to update them.* If nothing is changed in situ, to introduce change we must be able to deploy new instances without affecting the customer experience. While this can be done manually, it rarely works well as humans are bad at following rote procedures consistently. Plan to automate instance replacement.
- *Plan for instance and zone failures at all times.* Instance failure should be a normal part of doing business, whether unintended due to bugs or other issues, or intentional in order to perform component replacements on them. Zone failures on AWS and Google Cloud Platform should also be relatively painless if you architect well. On Azure, you'll want to use Availability Sets.
- *Don't let instances get stale.* Not unlike a computer, the longer an instance has been in use, the greater the chance is that it will have drifted from the optimal configuration. Don't run server instances for a long time.
- *Scale in and out automatically.* One of the great benefits of cloud computing is that you don't have to pre-provision resources. You should scale them based on the amount of utilization you have. If you're doing immutable infrastructure, you gain this ability at the application layer. So, why not use it?

## Working with Compute Resources

Now, let's look specifically at considerations to keep in mind when starting, operating, updating, scaling, and monitoring compute resources in an immutable infrastructure implementation.

### Starting compute resources

The first order of business is getting the bootstrapping fully automated. This means having no hands on the instance other than the run instance command or equivalent. Usually, this involves both some data passed into *cloud-init* and some discovery of the environment. For the former, each cloud service provider has a mechanism to configure at boot, such as AWS's User Data feature. For the latter, you may need to use a service discovery tool such as etcd or ZooKeeper. Running a service discovery cluster can be complex to set up, so you might want to avoid it for a sandbox project by using DNS.

You'll want your service to use the scale-out pattern for high availability, since removing and adding instances automatically is important to immutable infrastructure. The easiest way to start is by putting the instances behind a load balancer, which in some cases can also remove the need for service discovery. It's best if you automate adding and removing instances to and from the load balancer, which you could do with a script or with AWS Cloud Formation or OpsWorks. As we've suggested, scripts can add complexity, and the tools noted have their critics too. A unified approach, covered in the last section of this chapter, gives you an alternative.

### Operating compute resources

Once you can bootstrap instances successfully, you'll need a mechanism to replace instances due to configuration change or staleness. This turns out to be a rather difficult thing to do well, but it can be accomplished by hand-rolling scripts and using AWS's ASGs.

As described in Chapter 1, there are key advantages to routinely regenerating compute instances whether planned changes have occurred or not—like reducing configuration drift, mitigating errors, and, when done well, removing certain resident exploits.

To get immutable infrastructure right, you need a solid and reliable way to measure instance health. Your orchestration layer needs to know whether a particular instance is doing its job correctly. It's worth putting some real effort into good health checks, as false positives or negatives here can really hurt you by keeping nonperforming or broken instances in the user's path or by failing to bring healthy instances to bear. A health check should minimally test that the instance can perform its main function. This usually involves testing a connection through the instance to other services it needs to operate, such as a database connection.

### Updating compute resources

You'll want to introduce changes to your service. With immutable infrastructure, this is done by replacing compute instances. There are two broad approaches for updating your service:

1. You can build the new component version alongside the existing one, then change pointers, such as DNS records, to the new one.

2. You can use blue/green deployment, in which you build two nearly identical production environments. While the “blue” environment is live, you update and test the “green” environment. When “green” is ready, you switch the router to the “green” environment. Having the ability to roll back bad changes is a must, and using blue/green deployment provides a way to do this.

### Scaling compute resources

Immutable infrastructure implicitly allows for automatic scaling of your compute instances. This lets you match your spending on infrastructure to the actual demand on your service. To autoscale well, you need a good measure of your service’s requirements that accurately reflects the demands and constraints of the service as it relates to scale. CPU works for some services, latency for others, memory for others, and so on.

You must understand your service’s performance-limiting factors to scale effectively. For example, some services are I/O-bound, and the limiting factor is disk or network I/O. For other services, it may be CPU or memory, or a combination of some or all of these factors. Often, it’s best to measure service latency as this is a natural aggregate and reflects user experience. One danger in scaling metrics in general is that outside conditions can sometimes cause a mass scaling event, such as introduction of a new feature that causes memory to be over-consumed or latencies to climb due to an error. Always set reasonable upper bounds on your autoscaling.

### Monitoring compute resources

With immutable infrastructure, it’s even more important than usual to monitor the condition of your infrastructure to make sure it conforms to the intended configuration and health. AWS offers CloudWatch metrics as an extensible mechanism to monitor your infrastructure; Google offers Cloud Monitoring Beta at the time of this writing and Azure offers verbose monitoring. There are many vendors offering more detailed tools, such as New Relic. It’s important to ensure your monitoring is automated and integrated with your mechanisms for scaling, operations, and deployment.

## Beyond Compute Resources

Many conversations about immutable infrastructure stop with compute services, but to get the most out of the architecture you will want to employ it for network, storage, and management services to ensure their proper configuration and trustworthiness.

Unlike compute services, where ephemerality and immutability are intrinsically tied together, in other infrastructure services, such as networking, the logical services are long-lived. Immutability in this context involves self-healing and automated changes, rather than replacement.

### Network services

Long-lived components need to conform to a declaration for them to be immutable. Some parts of the infrastructure, such as the network, cannot have the short life spans of a compute instance. However, they can have their configurations declared in a specification. The runtime environment should be continuously checked against that declaration to ensure conformance. You can do this in a unified system, as discussed in the last section of this chapter.

A common example of where the mutability of Software Defined Networking (SDN) services often causes pain is the reuse of security groups on AWS across different applications or parts of an application. One user might depend on a certain port being open, while another user decides it's not necessary. If the second user modifies the definition of the security group, it can break the first user's application.

A good immutable infrastructure networking solution constantly monitors the network configuration against the declaration and either alters or, preferably, self-heals the network to conform to the declaration.

### Data services

Data services are used for many things, but we can consider three categories for our purposes:

- *Data that should be read-only*, such as the bits of the operating system or application. This is an easy type of data for immutable infrastructure, as we can have a master copy that is never used, from which we generate and regenerate the working copies. The

AWS Amazon Machine Image (AMI) and Elastic Block Store (EBS) snapshots are examples of this kind of data service.

- *Service registry and shared variables.* When you are architecting using immutable infrastructure patterns, you often need data to be present in the runtime environment that you can't predict when authoring. This can include data such as the IP addresses of machines that are in a cluster or keys used to access services. It's important for this data to have the correct access and permissions. These should be declarable in the same place you define the rest of the infrastructure.
- *Data that is read/write*, such as the logs of a system or the database behind an application. While it isn't possible to regenerate this sort of data easily and the benefits of trying to do so are questionable, it is possible to make sure the location and configuration of the data service don't mutate.

The simple way to get immutable infrastructure benefits in your data services is to use data services that provide external service level agreements and are managed by others. Amazon DynamoDB is one such service; another is Azure DocumentDB. The goal is to reduce maintenance costs and have a data persistence service with the right features, performance, and pricing. Using such a service is often preferable to rolling your own and maintaining it, as the service provider does the heavy lifting.

#### Management services

Services such as key management and access management also can be automated to function well with immutable infrastructure patterns. A complete expression of immutable infrastructure patterns should be a single place that configures, deploys, and manages your application, so automation of management services is a worthy goal. You can execute that with a system like Fugue, described in the next section.

## Immutable Infrastructure in a Unified System

In this chapter, we've examined what exists now (i.e., immutability in example implementations with commonly used and emergent tools) and what considerations lead us to build optimally (i.e., fundamental patterns and practices for immutability with cloud resources). Let's now synthesize by turning to another approach to immutable infrastructure that we called for in our introduction. It's an approach that considers existing problems and desired outcomes.

A robust, unified system that provides immutable infrastructure must be able to automatically handle the generation, replacement, and regeneration of compute instances (and containers if they are in use). It's a system designed with first principles of cloud computing foremost in mind. Unified doesn't mean monolithic; it means a system that's been designed in an intentional and coordinated manner, with microservices and API utilization given due consideration. And first principles refer to scale-out, automation, lower costs, not having to provision for bursts, and elimination of undifferentiated heavy lifting. The system must know the current state of the infrastructure as a whole and be able to provide immutability of components, such as network configurations, that are presented as mutable objects by cloud service providers.

Consider the implementation in Figure 2-3 and both the similarities and differences with regard to what we saw earlier, in Figures 2-1 and 2-2:

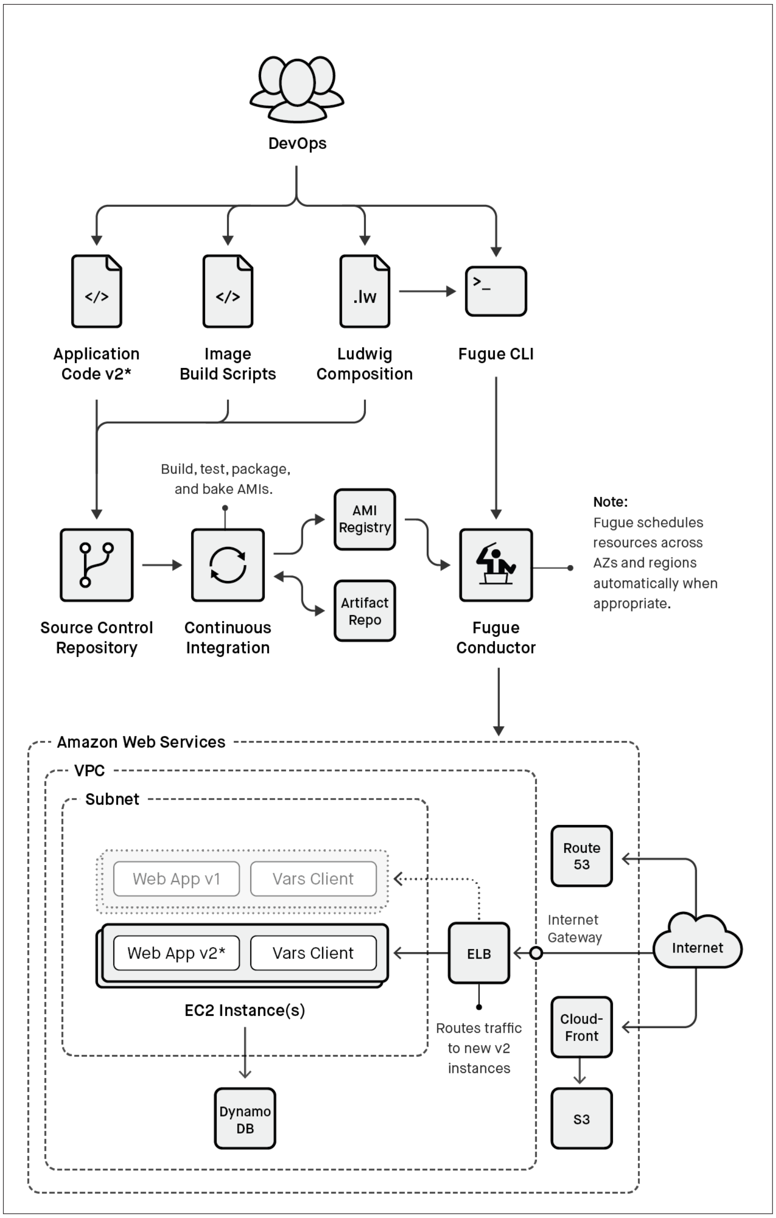

*Figure 2-3. Use case—immutable patterns with Fugue*

Fugue, shown here, is an example of a unified system that allows users to declare how the components of an application should deploy, scale, and interact.[1] Notice that users compose cloud infrastructure with concise declarations in a strongly typed and compiled language (Ludwig) that provides pre-launch error and policy checks. Users have the option to declare an automated rate of replacement for instances and containers and to declare enforcement patterns for other cloud resources and components.

No scripts, configuration management tools, or layers of orchestration that the user has to touch are required. Distributed variables and service registries are included. A unified system builds, continuously optimizes, and enforces infrastructure based on the declarations. It automatically accounts for timeouts, API details, failed instance launches, registering instances with the ELB, and health checks. Every fundamental pattern and best practice we've covered in this chapter, and the decisions embedded in those, should be built into a unified system. The point is to significantly reduce the complexity of hand-rolled and multitool solutions or those that require fine-grained, nuts-and-bolts mastery of a cloud provider's services.

Implicit in immutable infrastructure is that we need to have a place to declare what should exist in our infrastructure, in what configuration. If we can't define what we want, it's impossible to know if we have it. In programming, we have code to instruct the computer what should happen and the compiler to enforce immutability at runtime. In infrastructure, we have no program to declare our intentions, but instead a generally manual, ad hoc, and distributed body of knowledge. Sometimes this is recorded in documentation, but often not completely. A unified system, like Fugue, has the potential to change that. It requires some form of expression that supports immutable patterns (e.g., Fugue's Ludwig domain-specific language) and a runtime environment that enforces them (e.g., Fugue's Conductor).

In order to do cloud and immutable infrastructure well, you need a single source of truth—the state of the system—and a single source of trust—the knowledge that your decisions are being honored—just like a single computer needs an operating system. As we've noted, you can build a system from scratch, but it's a complex technical issue to take on, aimed at a moving target. Whatever tools or systems you choose, it's advantageous to have a single interface to the

1 Disclaimer: Josha Stella is the CEO of Fugue.

configuration of the infrastructure. Running a dynamic system at scale in the cloud is hard. Using immutable infrastructure patterns increases the dynamism of the system.

Two core characteristics shaped our thinking in architecting Fugue and are likely central to any unified approach:

*All aspects of infrastructure are defined and testable.*
: To have immutable infrastructure, every aspect of the infrastructure must have specific and testable definitions. Without the ability to test definitions, it will be impossible to know if the component or relationship conforms to the definition at any given time.

*A control process is used to instantiate and enforce the definitions.*
: A user needs to have a method to operate against the infrastructure in real time and constantly in order to monitor and maintain that the infrastructure is as intended.

CHAPTER 3

# Pressing Questions

In this last part of the guide, we attempt to answer some tough questions. It's our hope that contemplation here spurs many more questions and observations about immutable infrastructure. The point is to spark minds to go further—to encourage individuals and shops with deep reservoirs of talent and creativity to sharpen solutions, explore the cloud's intrinsic nature, and tap into its fullest capacities.

## What Are the Central Challenges with Immutable Infrastructure?

The challenges surrounding immutable infrastructure involve not the pattern itself nor the implementation in the runtime, but rather the process, human organization, and tooling that needs to be in place.

### Process Challenges

To do immutable infrastructure means to fully confront everything about distributed and stateless systems head on. If you end up building big, monolithic programs, you'll find that those don't work efficiently in this environment, nor do they work with immutable patterns because immutable patterns require the ability to replace components automatically and often. Large, monolithic programs generally contain many services that need to be patched and maintained in situ. If you go too small, that can gum a system as well, as you will have more components to keep track of than are necessary. Going too small also can cause underutilization of resources. Knowing how to measure services and determining what to do at what scale is tough for even seasoned architects.

A mature DevOps process is also a prerequisite. Developers need to intimately understand the operating environment due to the automation that immutable infrastructure requires. With demand and services scaling up and down all the time, there's a huge impact on application development itself—the application changes the infrastructure. They become a deeply connected concern.

## Organizational Challenges

Crafting good processes around immutable infrastructure assumes you have architects and developers with relatively rare skill sets. Expensive skill sets. You also need those individuals to be agile, curious, and willing—spreading that ethic across the team. Because the approach is new and because the use of distribution can be tough architecturally, those people tend to be hard to find. But once you have a resilient architecture using immutable patterns, it's actually easier for programmers to write code.

Leveraging multiple tools and/or complex tools usually also necessitates the formation of multiple teams in your organization. Figuring out who's on first can be really tough because the human challenge of coordinating the information exists. No single person probably has expertise across all of the tools.

So, a CIO charged with looking at where and when to use immutable patterns has a lot to weigh. From a business perspective, there's high risk up front in determining whether a shop can pull it off without significant time, costs, and production errors. Some will feel pressure to be more conservative about making the choice until there are dominant, well-tested designs and systems that function with excellence. On the other hand, once your shop is cloud native, you can do things that your peers can't, such as having much greater agility in deploying and managing services, having greater determinism in your environment, reducing maintenance overhead, and ultimately spending more of your time and money on growing your business, rather than on maintaining your infrastructure.

### Tooling Challenges

We looked at toolchains in Chapter 2. While vigorous work is being done, the field is not one with well-established prior art. People are still figuring things out. Hand-rolled solutions are common. You have to piece together big piles of complex tools and engineering conduits yourself, while troubleshooting with few comprehensive roadmaps. There's an immature market for tooling since we're not decades in with time-tested systems, like those used with traditional architectures.

## Where Do You Put the Data?

In Chapter 2, recall that we considered how data services align with immutable infrastructure, with the observation that application or log data that is read/write won't be replaced like compute instances. But those data services can benefit from the "self-healing" that the pattern offers. Let's dig deeper into questions of data, as this is a pressing concern for early adopters.

Where is application data stored—data that must be mutable for a given system to have functionality? Relational databases are often the answer for enterprises like banks. They need atomic, not eventually consistent, data transactions for some of their key business functions. Although relational databases like MySQL or PostgreSQL aren't designed for immutable patterns, they do deeply depend on configuration *enforcement* for viability.

Consider a scenario where someone inside an organization inadvertently changes an ingress rule to access a bastion host via a previously unknown IP address in order to run queries against the database. In the unified system approach, where the state of infrastructure is declared and read-only, the control process used to enforce the declarations has been automatically and continuously monitoring whether runtime state matches declared state. When the control process notices that an unexpected infrastructure change has occurred, it automatically corrects back to the declared state, sending a notification to the team and logging the incident. That's self-healing via immutable thinking. The bogus or mistaken change to infrastructure configuration is destroyed and replaced with the original state.

When you've built atop a cloud service provider's persistence services, like Microsoft's SQL Database service or AWS's RDS for Aurora or its DynamoDB service, you've also gained the cloud's native advantages in your ecosystem: low latency, high scalability, failure detection, data automatically replicated across multiple AZs, easy APIs with loose coupling, etc.

Keep in mind that it's crucial to concentrate data persistence in as few places as possible when designing a complete cloud system that's using immutable infrastructure for compute services and enforcement for data services. Every service that must persist data adds significant complexity to the application and its automation. Striving to make interfaces idempotent wherever possible will make the architecture simpler to modify over time and will make it easier to deal with distributed state issues.

## How Does Immutability in Programming Inform Infrastructure?

With this question, we're moving from our discussion about database services in the last section to examination of immutable data itself—quite a different ball game. Here, we're probing the intellectual inspiration for immutable infrastructure by drilling down into ideas and patterns found in programming and in OS principles. So, it's important to understand that context.

Data can be mutable or immutable: immutability in programming happens at the object level, meaning any code-level object such as data or a function. Programming with immutable languages means that instead of modifying an object, a new object is returned with the requested changes. Instead of changing a variable, you replace it. This principle is illustrated in Figure 3-1. Note the change in the value of a variable versus the mandatory creation of new data.

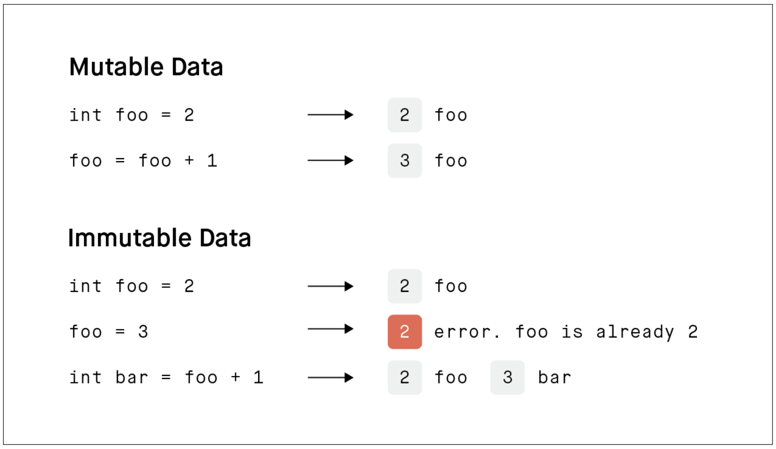

*Figure 3-1. Immutability in programming*

The compiler, runtime, or interpreter in an immutable language guarantees that objects will not be modified once created, enforcing immutability. It prevents users from breaking the rules and returns an error before the program is run. This guarantee is extremely useful when reasoning about program predictability, correctness, and reliability.

In this way, immutable infrastructure is similar to immutable data. Rather than patching or modifying your compute instances, as with your "data" in immutable languages, you replace them with new ones that incorporate the changes. Programming languages that use immutable data, such as Erlang, are safer choices for distributed programs than languages that use mutable data. Similarly, immutable infrastructure provides better reliability and a simpler set of problems for distributed applications, particularly in the cloud.

The analogy of traditional programming and infrastructure has its limits, as the scale is different in the cloud and we have more variability in the kinds of objects we work with and in the interfaces to them. When writing a program, the compiler can check to be sure that you aren't mutating data in situ, but an administrator with the right permissions can change the configuration of a network or a compute instance unless constrained by the logical equivalent of the compiler. That's part of the control mechanism we described at the end of Chapter 2, which enforces infrastructure declarations and automatically responds to attempted modifications. In its other functions and processes, it behaves loosely like an operating system for cloud-as-computer.

It's worth considering that cloud providers have designed their services to cater to traditional, mutable infrastructure models. This means that, given root level access, every component in a cloud is mutable, typically through numerous interfaces. This is also true of memory in a computer—but automating processes with a strict set of parameters that deny root can keep both environments safer.

## What's Next?

With cloud computing, we're increasingly composing systems of elastic collections of services running on many compute instances. We now commonly employ application statelessness in order to exploit cloud system elasticity and to achieve the performance required of web-scale systems. We're discovering the advantages of automating the creation and destruction of components of a system, incorporating changes only on replacement. But we're also finding that our existing methods to do so are complex and undergoing the rapid and uneven development typical of new technologies. It's our contention that automated immutable infrastructure in a unified system, via declaration and enforcement, fortifies applications. It provides consistent resilience to cloud quality issues. As a pattern native to the cloud's resource abstraction, interfacing, and distribution, immutable infrastructure is likely here to stay.

New questions will continue to emerge as we consider how legacy and hybrid systems ultimately will undergo migration. How will the rise of services like Lambda on AWS and similar direct compute services play into cloud systems architecture? What abstractions will be useful five years from now? Ten? What patterns and principles are sound enough to not just withstand but *feed* evolving technologies? Whether it's manipulated by DevOps-style users or eventually maintained within the guts of cloud providers' internal systems, our money is on immutability.

# About the Author

**Josha Stella** is cofounder and CEO of Fugue. He's been programming since his teen years in the 1980s, when he learned to write code to automate animation tasks on his Amiga and realized that programming was more interesting than animation to him. Prior to Fugue, Josh most recently was a Principal Solutions Architect at Amazon Web Services. He has served as a CTO for a prior startup and in numerous other technical and leadership roles over the last 25 years. When he's not working, Josh is likely playing a guitar, riding a bicycle, or cooking with his family.