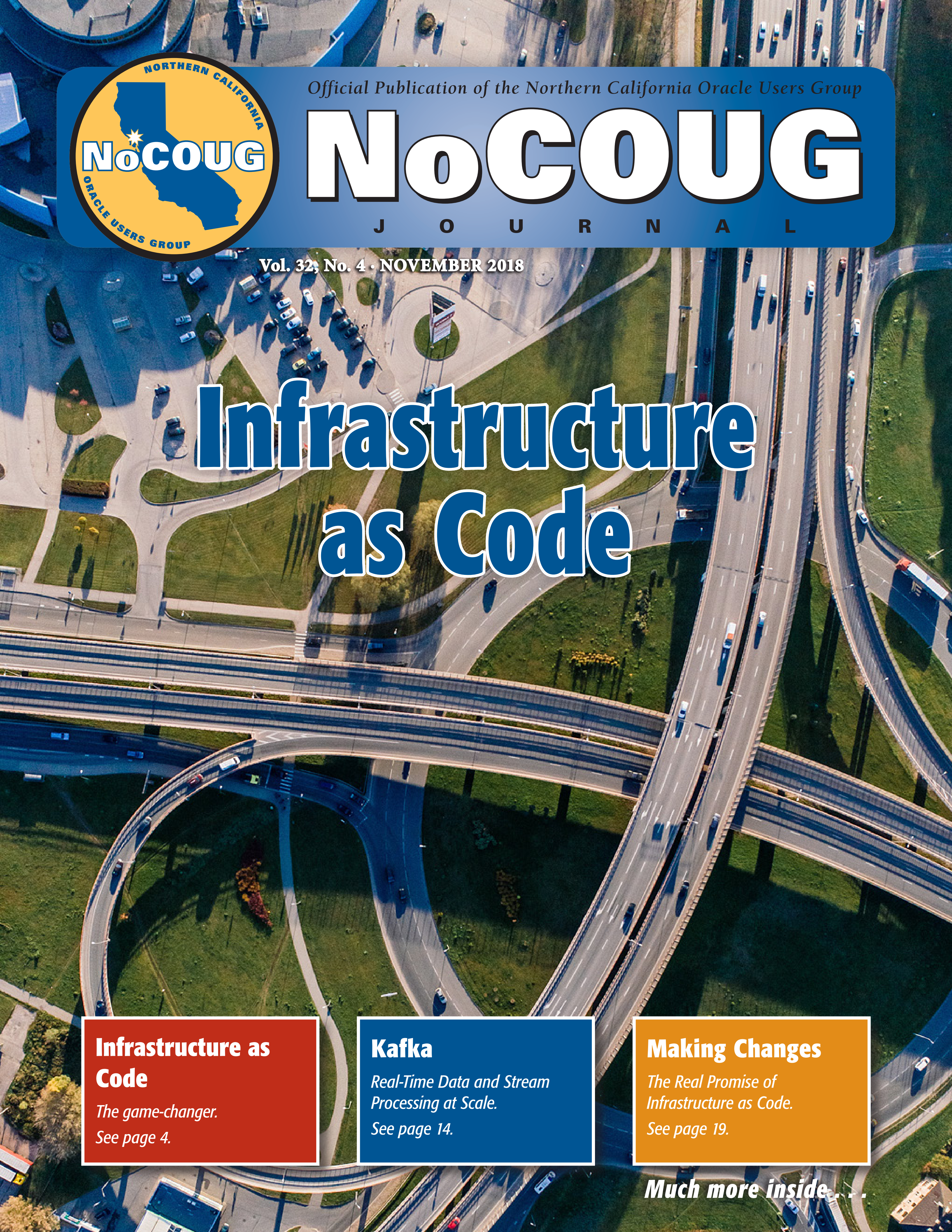
Official Publication of the Northern California Oracle Users Group

NoCOUG JOURNAL

Vol. 32, No. 4 · NOVEMBER 2018



Infrastructure as Code

Infrastructure as Code

The game-changer.

See page 4.

Kafka

Real-Time Data and Stream Processing at Scale.

See page 14.

Making Changes

The Real Promise of Infrastructure as Code.

See page 19.

Much more inside . . .

The New Phoenix by Axxana

For Oracle Databases and Applications

Zero Data Loss at Unlimited Distances

Synchronous Protection at Maximum Performance

- Low Cost Replication Lines
- Shortest Recovery Time
- Full Consistency Across Multiple Databases

AXXANA

BUILT TO LAST

www.axxana.com

ORACLE Gold Partner

ORACLE BIG DATA APPLIANCE READY

ORACLE DATABASE READY

ORACLE EXADATA READY

ORACLE LINUX READY

Professionals at Work

First there are the IT professionals who write for the *Journal*. A very special mention goes to Brian Hitchcock, who has written dozens of book reviews over a 12-year period.

Next, the *Journal* is professionally copyedited and proofread by veteran copyeditor Karen Mead of Creative Solutions. Karen polishes phrasing and calls out misused words (such as "reminiscences" instead of "reminisces"). She dots every i, crosses every t, checks every quote, and verifies every URL.

Then, the *Journal* is expertly designed by graphics duo Kenneth Lockerbie and Richard Repas of San Francisco-based Giraffex.

And, finally, David Gonzalez at Layton Printing Services deftly brings the *Journal* to life on an offset printer.

The front cover has an aerial photo taken using a DJI FC6310 (Phantom 4 Pro) drone by Aleksejs Bergmanis from Pexels. ▲

Table of Contents

Book Excerpt 4
Brian's Notes 14
Special Feature 19
Interview 21
Picture Diary 25

ADVERTISERS
Axxana 2
Vexata 27
Quest 27
OraPub 28

Publication Notices and Submission Format

The *NoCOUG Journal* is published four times a year by the Northern California Oracle Users Group (NoCOUG) approximately two weeks prior to the quarterly educational conferences.

Please send your questions, feedback, and submissions to the *NoCOUG Journal* editor at **journal@nocoug.org**.

The submission deadline for each issue is eight weeks prior to the quarterly conference. Article submissions should be made in Microsoft Word format via email.

Copyright © by the Northern California Oracle Users Group except where otherwise indicated.

NoCOUG does not warrant the NoCOUG Journal *to be error-free.*

ADVERTISING RATES

The *NoCOUG Journal* is published quarterly.

Size	Per Issue	Per Year
Quarter Page	$125	$400
Half Page	$250	$800
Full Page	$500	$1,600
Inside Cover	$750	$2,400

Personnel recruitment ads are not accepted.

journal@nocoug.org

Infrastructure as Code

Managing Servers in the Cloud

by Kief Morris

This is an extract from the book Infrastructure as Code: Managing Servers in the Cloud *by Kief Morris, O'Reilly Media, Jun 27, 2016, ISBN 978-1491924358. Reprinted with permission.*

Infrastructure and software development teams are increasingly building and managing infrastructure using automated tools that have been described as "infrastructure as code." These tools expect users to define their servers, networking, and other elements of an infrastructure in files modeled after software source code. The tools then compile and interpret these files to decide what action to take.

This class of tool has grown naturally with the DevOps movement.[1] The DevOps movement is mainly about culture and collaboration between software developers and software operations people. Tooling that manages infrastructure based on a software development paradigm has helped bring these communities together.

Managing infrastructure as code is very different from classic infrastructure management. I've met many teams who have struggled to work out how to make this shift. But ideas, patterns, and practices for using these tools effectively have been scattered across conference talks, blog posts, and articles. I've been waiting for someone to write a book to pull these ideas together into a single place. I haven't seen any sign of this, so finally took matters into my own hands. You're now reading the results of this effort!

How I Learned to Stop Worrying and to Love the Cloud

I set up my first server, a dialup BBS[2] in 1992. This led to Unix system administration and then to building and running hosted software systems (before we called it SaaS, aka "Software as a Service") for various companies, from startups to enterprises.

I've been on a journey to infrastructure as code the entire time, before I'd ever heard the term.

Things came to a head with virtualization. The story of my stumbling adoption of virtualization and the cloud may be familiar, and it illustrates the role that infrastructure as code has to play in modern IT operations.

My First Virtual Server Farm

I was thrilled when my team got the budget to buy a pair of beefy HP rack servers and licenses for VMware ESX Server back in 2007.

[1] Andrew Clay Shafer and Patrick Debois triggered the DevOps movement with a talk at the Agile 2008 conference (http://www.jedi.be/presentations/agile-infrastructure-agile-2008.pdf). The movement grew, mainly driven by the series of DevOpsDays (http://www.devopsdays.org/) conferences organized by Debois.

[2] A BBS is a bulletin board system (**https://en.wikipedia.org/wiki/Bulletin_board_system**).

We had in our office's server racks around 20 1U and 2U servers named after fruits (Linux servers) and berries (Windows database servers) running test environments for our development teams. Stretching these servers to test various releases, branches, and high-priority, proof-of-concept applications was a way of life. Network services like DNS, file servers, and email were crammed onto servers running multiple application instances, web servers, and database servers.

"Infrastructure as code" tools expect users to define their servers, networking, and other elements of an infrastructure in files modeled after software source code. The tools then compile and interpret these files to decide what action to take.

So we were sure these new virtual servers would change our lives. We could cleanly split each of these services onto its own virtual machine (VM), and the ESX hypervisor software would help us to squeeze the most out of the multicore server machines and gobs of RAM we'd allocated. We could easily duplicate servers to create new environments and archive those servers that weren't needed onto disk, confident they could be restored in the future if needed.

Those servers did change our lives. But although many of our old problems went away, we discovered new ones, and we had to learn completely different ways of thinking about our infrastructure.

Virtualization made creating and managing servers much easier. The flip side of this was that we ended up creating far more servers than we could have imagined. The product and marketing people were delighted that we could give them a new environment to demo things in well under a day, rather than need them to find money in the budget and then wait a few weeks for us to order and set up hardware servers.

The Sorcerer's Apprentice

A year later, we were running well over 100 VMs and counting. We were well underway with virtualizing our production servers and experimenting with Amazon's new cloud hosting service. The benefits virtualization had brought to the business people meant we had money for more ESX servers and for shiny SAN devices to feed the surprising appetite our infrastructure had for storage.

But we found ourselves a bit like Mickey Mouse in "The Sorcerer's Apprentice" from *Fantasia*. We spawned virtual servers, then more, then even more. They overwhelmed us. When something broke, we tracked down the VM and fixed whatever was wrong with it, but we couldn't keep track of what changes we'd made where.

Well, a perfect hit!
See how he is split!
Now there's hope for me,
and I can breathe free!
Woe is me! Both pieces
come to life anew,
now, to do my bidding
I have servants two!
Help me, O great powers!
Please, I'm begging you!

—Excerpted from Brigitte Dubiel's translation of "Der Zauberlehrling" ("*The Sorcerer's Apprentice*") by Johann Wolfgang von Goethe

As new updates to operating systems, web servers, app servers, database servers, JVMs, and various other software packages came out, we would struggle to install them across all of our systems. We would apply them successfully to some servers, but on others the upgrades broke things, and we didn't have time to stomp out every incompatibility. Over time, we ended up with many combinations of versions of things strewn across hundreds of servers.

We had been using configuration automation software even before we virtualized, which should have helped with these issues. I had used CFEngine in previous companies, and when I started this team, I tried a new tool called Puppet. Later, when spiking out ideas for an AWS infrastructure, my colleague Andrew introduced Chef. All of these tools were useful, but particularly in the early days, they didn't get us out of the quagmire of wildly different servers.

The problem was that, although Puppet (and Chef and the others) should have been set up and left running unattended across all of our servers, we couldn't trust it. Our servers were just too different. We would write manifests to configure and manage a particular application server. But when we ran it against another, theoretically similar app server, we found that different versions of Java, application software, and OS components would cause the Puppet run to fail, or worse, break the application server.

So we ended up using Puppet ad hoc. We could safely run it against new VMs, although we might need to make some tweaks after it ran. We would write manifests for a specific task and then run them against servers one at a time, carefully checking the result and making fixes as needed.

So configuration automation was a useful aid, somewhat better than shell scripts, but the way we used it didn't save us from our sprawl of inconsistent servers.

Cloud from Scratch

Things changed when we began moving things onto the cloud. The technology itself wasn't what improved things; we could have done the same thing with our own VMware servers. But because we were starting fresh, we adopted new ways of managing servers based on what we had learned with our virtualized farm and on what we were reading and hearing from IT Ops teams at companies like Flickr, Etsy, and Netflix. We baked these new ideas into the way we managed services as we migrated them onto the cloud.

The key idea of our new approach was that every server could be automatically rebuilt from scratch, and our configuration tooling would run continuously, not ad hoc. Every server added into our new infrastructure would fall under this approach. If automation broke on some edge case, we would either change the automation to handle it, or else fix the design of the service so it was no longer an edge case.

The DevOps movement is mainly about culture and collaboration between software developers and software operations people. Tooling that manages infrastructure based on a software development paradigm has helped bring these communities together. Infrastructure as Code is one of the cornerstones of DevOps. It is the "A" in "CAMS": culture, automation, measurement, and sharing.

The new regime wasn't painless. We had to learn new habits, and we had to find ways of coping with the challenges of a highly automated infrastructure. As the members of the team moved on to other organizations and got involved with communities such as DevOpsDays, we learned and grew. Over time, we reached the point where we were habitually working with automated infrastructures with hundreds of servers, with much less effort and headache than we had been in our "Sorcerer's Apprentice" days.

Joining ThoughtWorks was an eye-opener for me. The development teams I worked with were passionate about using XP engineering practices like test-driven development (**http://martinfowler.com/bliki/TestDrivenDevelopment.html**) (TDD), continuous integration (**http://www.martinfowler.com/articles/continuousIntegration.html**) (CI) and continuous delivery (**http://martinfowler.com/books/continuousDelivery.html**) (CD). Because I had already learned to manage infrastructure scripts and configuration files in source control systems, it was natural to apply these rigorous development and testing approaches to them.

Working with ThoughtWorks has also brought me into contact with many IT operations teams, most of whom are using virtualization, cloud, and automation tools to handle a variety of challenges. Working with them to share and learn new ideas and techniques has been a fantastic experience.

Why I'm Writing This Book

I've run across many teams who are in the same place I was a few years ago: people who are using cloud, virtualization, and automation tools but haven't got it all running as smoothly as they know they could.

Much of the challenge is time. Day-to-day life for system administrators is coping with a never-ending flow of critical work. Fighting fires, fixing problems, and setting up new business-critical projects doesn't leave much time to work on the fundamental improvements that will make the routine work easier.

My hope is that this book provides a practical vision for how to manage IT infrastructure, with techniques and patterns that teams can try and use. I will avoid the details of configuring and using specific tools so that the content will be useful for working with different tools, including ones that may not exist yet. Meanwhile, I will use examples from existing tools to illustrate points I make.

The infrastructure-as-code approach is essential for managing cloud infrastructure of any real scale or complexity, but it's not exclusive to organizations using public cloud providers. The techniques and practices in this book have proven effective in virtualized environments and even for bare-metal servers that aren't virtualized.

Infrastructure as Code is one of the cornerstones of DevOps. It is the "A" in "CAMS" (**http://itrevolution.com/devops-culture-part-1/**): culture, automation, measurement, and sharing.

Who This Book Is For

This book is for people who work with IT infrastructure, particularly at the level of managing servers and collections of servers. You may be a system administrator, infrastructure engineer, team lead, architect, or a manager with technical interest. You might also be a software developer who wants to build and use infrastructure.

I'm assuming you have some exposure to virtualization or IaaS (Infrastructure as a Service) cloud, so you know how to create a server, and the concepts of configuring operating systems. You've probably at least played with configuration automation software like Ansible, Chef, or Puppet.

Virtualization made creating and managing servers much easier. The product and marketing people were delighted that we could give them a new environment in well under a day but we found ourselves like Mickey Mouse in "The Sorcerer's Apprentice" from Fantasia. We spawned virtual servers, then more, then even more.

While this book may introduce some readers to infrastructure as code, I hope it will also be interesting to people who work this way already and a vehicle through which to share ideas and start conversations about how to do it even better.

What Tools Are Covered

This book doesn't offer instructions in using specific scripting languages or tools. There are code examples from specific tools, but these are intended to illustrate concepts and approaches, rather than to provide instruction. This book should be helpful to you regardless of whether you use Chef on OpenStack, Puppet on AWS, Ansible on bare metal, or a completely different stack.

The specific tools that I do mention are ones which I'm aware of, and which seem to have a certain amount of traction in the field. But this is a constantly changing landscape, and there are plenty of other relevant tools.

The tools I use in examples tend to be ones with which I am familiar enough to write examples that demonstrate the point I'm trying to make. For example, I use Terraform for examples of infrastructure definitions because it has a nice, clean syntax, and I've used it on multiple projects. Many of my examples use Amazon's AWS cloud platform because it is likely to be the most familiar to readers.

How to Read This Book

Read Chapter 1, or at least skim it, to understand the terms this book uses and the principles this book advocates. You can then use this to decide which parts of the book to focus on.

If you're new to this kind of automation, cloud, and infrastructure orchestration tool- ing, then you'll want to focus on Part I, and then move on to Part II. Get comfortable with those topics before proceeding to Part III.

If you've been using the types of automation tools described here, but don't feel like you're using them the way they're intended after reading Chapter 1, then you may want to skip or skim the rest of Part I. Focus on Part II, which describes ways of using dynamic and automated infrastructure that align with the principles outlined in Chapter 1.

If you're comfortable with the dynamic infrastructure and automation approaches described in Chapter 1, then you may want to skim Parts I and II and focus on Part III, which gets more deeply into the infrastructure management regime: architectural approaches as well as team workflow.

Challenges and Principles

The new generation of infrastructure management technologies promises to transform the way we manage IT infrastructure. But many organizations today aren't seeing any dramatic differences, and some are finding that these tools only make life messier. As we'll see, infrastructure as code is an approach that provides principles, practices, and patterns for using these technologies effectively.

Why Infrastructure as Code?

Virtualization, cloud, containers, server automation, and software-defined networking should simplify IT operations work. It should take less time and effort to provision, configure, update, and maintain services. Problems should be quickly detected and resolved, and systems should all be consistently configured and up to date. IT staff should spend less time on routine drudgery, having time to rapidly make changes and improvements to help their organizations meet the ever-changing needs of the modern world.

But even with the latest and best new tools and platforms, IT operations teams still find that they can't keep up with their daily workload. They don't have the time to fix longstanding problems with their systems, much less revamp them to make the best use of new tools. In fact, cloud and automation often makes things worse. The ease of provisioning new infrastructure leads to an ever-growing portfolio of systems, and it takes an ever-increasing amount of time just to keep everything from collapsing.

Adopting cloud and automation tools immediately lowers barriers for making changes to infrastructure. But managing changes in a way that improves consistency and reliability doesn't come out of the box with the software. It takes people to think through how they will use the tools and put in place the systems, processes, and habits to use them effectively.

Some IT organizations respond to this challenge by applying the same types of processes, structures, and governance that they used to manage infrastructure and software before cloud and automation became commonplace. But the principles that applied in a time when it took days or weeks to provision a new server struggle to cope now that it takes minutes or seconds.

Legacy change management processes are commonly ignored, bypassed, or overruled by people who need to get things done.[3] Organizations that are more successful in enforcing these processes are increasingly seeing themselves outrun by more technically nimble competitors.

Legacy change management approaches struggle to cope with the pace of change offered by cloud and automation. But there is still a need to cope with the ever-growing, continuously changing landscape of systems created by cloud and automation tools. This is where infrastructure as code[4] comes in.

The Iron Age and the Cloud Age

In the "iron age" of IT, systems were directly bound to physical hardware. Provisioning and maintaining infrastructure was manual work, forcing humans to spend their time pointing, clicking, and typing to keep the gears turning. Because changes involved so much work, change management processes emphasized careful up-front consideration, design, and review work. This made sense because getting it wrong was expensive.

In the "cloud age" of IT, systems have been decoupled from the physical hardware. Routine provisioning and maintenance can be delegated to software systems, freeing the humans from drudgery. Changes can be made in minutes, if not seconds. Change management can exploit this speed, providing better reliability along with faster time to market.

What Is Infrastructure as Code?

Infrastructure as code is an approach to infrastructure automation based on practices from software development. It emphasizes consistent, repeatable routines for provisioning and changing systems and their configuration. Changes are made to definitions and then rolled out to systems through unattended processes that include thorough validation.

The premise is that modern tooling can treat infrastructure as if it were software and data. This allows people to apply software development tools such as version control systems (VCS), automated testing libraries, and deployment orchestration to manage infrastructure. It also opens the door to exploit development practices such as test-driven development (TDD), continuous integration (CI), and continuous delivery (CD).

Infrastructure as code has been proven in the most demanding environments. For companies like Amazon, Netflix, Google, Facebook, and Etsy, IT systems are not just business critical; they are the business. There is no tolerance for downtime. Amazon's systems handle hundreds of millions of dollars in transactions every day. So it's no surprise that organizations like these are pioneering new practices for large scale, highly reliable IT infrastructure.

This book aims to explain how to take advantage of the cloud-era, infrastructure-as-code approaches to IT infrastructure management. This chapter explores the pitfalls that organizations often fall into when adopting the new generation of infrastructure technology. It describes the core principles and key practices of infrastructure as code that are used to avoid these pitfalls.

The infrastructure-as-code approach is essential for managing cloud infrastructure of any real scale or complexity, but it's not exclusive to organizations using public cloud providers. The principles and practices of infrastructure as code can be applied to infrastructure whether it runs on cloud, virtualized systems, or even directly on physical hardware.

Goals of Infrastructure as Code

The types of outcomes that many teams and organizations look to achieve through infrastructure as code include:

- ➤ IT infrastructure supports and enables change, rather than being an obstacle or a constraint.
- ➤ Changes to the system are routine, without drama or stress for users or IT staff.
- ➤ IT staff spends their time on valuable things that engage their abilities, not on routine, repetitive tasks.
- ➤ Users are able to define, provision, and manage the resources they need, without needing IT staff to do it for them.
- ➤ Teams are able to easily and quickly recover from failures, rather than assuming failure can be completely prevented.
- ➤ Improvements are made continuously, rather than done through expensive and risky "big bang" projects.
- ➤ Solutions to problems are proven through implementing, testing, and measuring them, rather than by discussing them in meetings and documents.

[3] "Shadow IT" is when people bypass formal IT governance to bring in their own devices, buy and install unapproved software, or adopt cloud-hosted services. This is typically a sign that internal IT is not able to keep up with the needs of the organization it serves.

[4] The phrase "infrastructure as code" doesn't have a clear origin or author. While writing this book, I followed a chain of people who have influenced thinking around the concept, each of whom said it wasn't them, but offered suggestions. This chain had a number of loops. The earliest reference I could find was from the Velocity conference in 2009, in a talk by Andrew Clay-Shafer and Adam Jacob. John Willis may be the first to document the phrase, in an article about the conference (**http://itknowledgeexchange.techtarget.com/cloud-computing/infrastructure-as-code/**). Luke Kaines has admitted that he may have been involved, the closest anyone has come to accepting credit.

In the "iron age" of IT, provisioning and maintaining infrastructure was manual work, forcing humans to spend their time pointing, clicking, and typing to keep the gears turning. Because changes involved so much work, change management processes emphasized careful up-front consideration, design, and review work. In the "cloud age" of IT, routine provisioning and maintenance can be delegated to software systems, freeing the humans from drudgery. Changes can be made in minutes, if not seconds.

Infrastructure as Code Is Not Just for the Cloud

Infrastructure as code has come into its own with cloud, because it's difficult to manage servers in the cloud well without it. But the principles and practices of infrastructure as code can be applied to infrastructure whether it runs on cloud, virtualized systems, or even directly on physical hardware.

I use the phrase "dynamic infrastructure" to refer to the ability to create and destroy servers programmatically; Chapter 2 is dedicated to this topic. Cloud does this naturally, and virtualization platforms can be configured to do the same. But even hardware can be automatically provisioned so that it can be used in a fully dynamic fashion. This is sometimes referred to as "bare-metal cloud."

It is possible to use many of the concepts of infrastructure as code with static infrastructure. Servers that have been manually provisioned can be configured and updated using server configuration tools. However, the ability to effortlessly destroy and rebuild servers is essential for many of the more advanced practices described in this book.

Challenges with Dynamic Infrastructure

This section looks at some of the problems teams often see when they adopt dynamic infrastructure and automated configuration tools. These are the problems that infrastructure as code addresses, so understanding them lays the groundwork for the principles and concepts that follow.

Server Sprawl

Cloud and virtualization can make it trivial to provision new servers from a pool of resources. This can lead to the number of servers growing faster than the ability of the team to manage them as well as they would like.

When this happens, teams struggle to keep servers patched and up to date, leaving systems vulnerable to known exploits. When problems are discovered, fixes may not be rolled out to all of the systems that could be affected by them. Differences in versions and configurations across servers mean that software and scripts that work on some machines don't work on others.

This leads to inconsistency across the servers, called *configuration drift*.

Configuration Drift

Even when servers are initially created and configured consistently, differences can creep in over time:

- Someone makes a fix to one of the Oracle servers to fix a specific user's problem, and now it's different from the other Oracle servers.
- A new version of JIRA needs a newer version of Java, but there's not enough time to test all of the other Java-based applications so that everything can be upgraded.
- Three different people install IIS on three different web servers over the course of a few months, and each person configures it differently.
- One JBoss server gets more traffic than the others and starts struggling, so someone tunes it, and now its configuration is different from the other JBoss servers.

Being different isn't bad. The heavily loaded JBoss server probably should be tuned differently from ones with lower levels of traffic. But variations should be captured and managed in a way that makes it easy to reproduce and to rebuild servers and services.

Unmanaged variation between servers leads to snowflake servers and automation fear.

Snowflake Servers

A snowflake server is different from any other server on your network. It's special in ways that can't be replicated.

Years ago I ran servers for a company that built web applications for clients, most of which were monstrous collections of Perl CGI. (Don't judge us, this was the dot-com era, and everyone was doing it.) We started out using Perl 5.6, but at some point the best libraries moved to Perl 5.8 and couldn't be used on 5.6. Eventually almost all of our newer applications were built with 5.8 as well, but there was one particularly important client application that simply wouldn't run on 5.8.

It was actually worse than this. The application worked fine when we upgraded our shared staging server to 5.8, but crashed when we upgraded the staging environment. Don't ask why we upgraded production to 5.8 without discovering the problem with staging, but that's how we ended up. We had one special server that could run the application with Perl 5.8, but no other server would.

We ran this way for a shamefully long time, keeping Perl 5.6 on the staging server and crossing our fingers whenever we deployed to production. We were terrified to touch anything on the production server, afraid to disturb whatever magic made it the only server that could run the client's application.

This situation led us to discover Infrastructures.Org (**http://www.infrastructures.org/index.shtml**), a site that introduced me to ideas that were a precursor to infrastructure as code. We made sure that all of our servers were built in a repeatable way, installing the operating system with the Fully Automated Installation (FAI) tool (**http://bit.ly/1spUXvl**), configuring the server with CFEngine, and checking everything into our CVS version control system (**http://www.nongnu.org/cvs/**).

As embarrassing as this story is, most IT operations teams have similar stories of special servers that couldn't be touched, much less reproduced. It's not always a mysterious fragility; sometimes there is an important software package that runs on an entirely different OS than everything else in the infrastructure. I recall an accounting package that needed to run on AIX, and a PBX system running on a Windows NT 3.51 server specially installed by a long-forgotten contractor.

Once again, being different isn't bad. The problem is when the team that owns the server doesn't understand how and why it's different, and wouldn't be able to rebuild it. An operations team should be able to confidently and quickly rebuild any server in their infrastructure. If any server doesn't meet this requirement, constructing a new, reproducible process that can build a server to take its place should be a leading priority for the team.

Fragile Infrastructure

A fragile infrastructure is easily disrupted and not easily fixed. This is the snowflake server problem expanded to an entire portfolio of systems.

The solution is to migrate everything in the infrastructure to a reliable, reproducible infrastructure, one step at a time. The *Visible Ops Handbook*[5] outlines an approach for bringing stability and predictability to a difficult infrastructure.

Don't touch that server. Don't point at it. Don't even look at it.

There is the possibly apocryphal story of the data center with a server that nobody had the login details for, and nobody was certain what the server did. Someone took the bull by the horns and unplugged the server from the network. The network failed completely, the cable was plugged back in, and nobody ever touched the server again.

Automation Fear

At an Open Space session (**http://en.wikipedia.org/wiki/Open_Space_Technology**) on configuration automation at a DevOpsDays conference (**http://www.devopsdays.org/**), I asked the group how many of them were using automation tools like Puppet or Chef. The majority of hands went up. I asked how many were running these tools unattended, on an automatic schedule. Most of the hands went down.

Many people have the same problem I had in my early days of using automation tools. I used automation selectively—for example, to help build new servers, or to make a specific configuration change. I tweaked the configuration each time I ran it, to suit the particular task I was doing.

I was afraid to turn my back on my automation tools, because I lacked confidence in what they would do.

I lacked confidence in my automation because my servers were not consistent.

My servers were not consistent because I wasn't running automation frequently and consistently.

This is the automation fear spiral, as shown in Figure 1-1, and infrastructure teams need to break this spiral to use automation successfully. The most effective way to break the spiral is to face your fears. Pick a set of servers, tweak the configuration definitions so that you know they work, and schedule them to run unattended, at least once an hour. Then pick another set of servers and repeat the process, and so on until all of your servers are continuously updated.

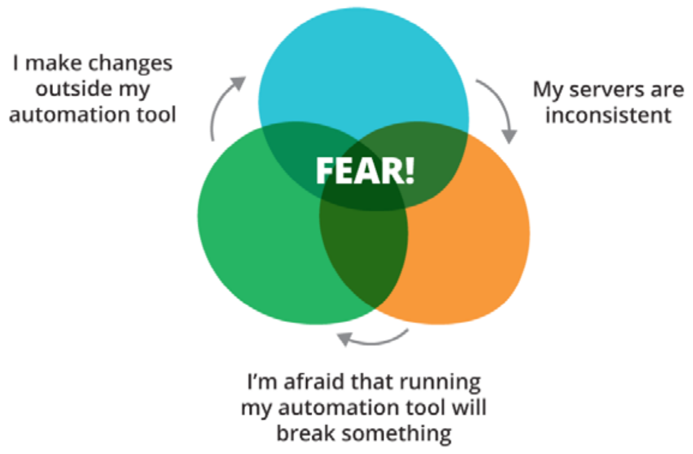

I make changes outside my automation tool
FEAR!
My servers are inconsistent
I'm afraid that running my automation tool will break something

Figure 1-1. The automation fear spiral

In an ideal world, you would never need to touch an automated infrastructure once you've built it, other than to support something new or fix things that break. Sadly, the forces of entropy mean that even without a new requirement, infrastructure decays over time. The folks at Heroku call this erosion (**https://devcenter.heroku.com/articles/erosion-resistance**). Erosion is the idea that problems will creep into a running system over time.

The Heroku folks give these examples of forces that can erode a system over time:

- Operating system upgrades, kernel patches, and infrastructure software (e.g., Apache, MySQL, SSH, OpenSSL) updates to fix security vulnerabilities
- The server's disk filling up with logfiles
- One or more of the application's processes crashing or getting stuck, requiring someone to log in and restart them
- Failure of the underlying hardware causing one or more entire servers to go down, taking the application with it

Principles of Infrastructure as Code

This section describes principles that can help teams overcome the challenges described earlier in this chapter.

Systems Can Be Easily Reproduced

It should be possible to effortlessly and reliably rebuild any element of an infrastructure. Effortlessly means that there is no need to make any significant decisions about how to rebuild the thing. Decisions about which software and versions to install on a server, how to choose a hostname, and so on should be captured in the scripts and tooling that provision it.

The ability to effortlessly build and rebuild any part of the infrastructure is powerful. It removes much of the risk, and fear, when making changes. Failures can be handled quickly and with confidence. New services and environments can be provisioned with little effort.

Approaches for reproducibly provisioning servers and other infrastructure elements are discussed in Part II of this book.

[5] First published in 2005, the *Visible Ops Handbook* (**http://www.amazon.com/Visible-Ops-Handbook-Implementing-Practical-ebook/dp/B002BWQBEE**) by Gene Kim, George Spafford, and Kevin Behr (IT Process Institute, Inc.) was written before DevOps, virtualization, and automated configuration became mainstream, but it's easy to see how infrastructure as code can be used within the framework described by the authors.

Systems Are Disposable

One of the benefits of dynamic infrastructure is that resources can be easily created, destroyed, replaced, resized, and moved. In order to take advantage of this, systems should be designed to assume that the infrastructure will always be changing. Software should continue running even when servers disappear, appear, and when they are resized.

The types of outcomes include: IT infrastructure supports and enables change, rather than being an obstacle or a constraint; changes to the system are routine, without drama or stress for users or IT staff; IT staff spends their time on valuable things that engage their abilities, not on routine, repetitive tasks; and users are able to define, provision, and manage the resources they need, without needing IT staff to do it for them.

The ability to handle changes gracefully makes it easier to make improvements and fixes to running infrastructure. It also makes services more tolerant to failure. This becomes especially important when sharing large-scale cloud infrastructure, where the reliability of the underlying hardware can't be guaranteed.

Cattle, Not Pets

A popular expression is to "treat your servers like cattle, not pets."[6] I miss the days of having themes for server names and carefully selecting names for each new server I provisioned. But I don't miss having to manually tweak and massage every server in our estate.

A fundamental difference between the iron age and cloud age is the move from unreliable software, which depends on the hardware to be very reliable, to software that runs reliably on unreliable hardware.[7] See Chapter 14 for more on how embracing disposable infrastructure can be used to improve service continuity.

The Case of the Disappearing File Server

The idea that servers aren't permanent things can take time to sink in. On one team, we set up an automated infrastructure using VMware and Chef, and got into the habit of casually deleting and replacing VMs. A developer, needing a web server to host files for teammates to download, installed a web server onto a server in the development environment and put the files there. He was surprised when his web server and its files disappeared a few days later.

After a bit of confusion, the developer added the configuration for his file repository to the Chef configuration, taking advantage of tooling we had to persist data to a SAN. The team ended up with a highly reliable, automatically configured file sharing service.

To borrow a cliche, the disappearing server is a feature, not a bug. The old world where people installed ad hoc tools and tweaks in random places leads straight to the old world of snowflakes and untouchable fragile infrastructure. Although it was uncomfortable at first, the developer learned how to use infrastructure as code to build services—a file repository in this case—that are reproducible and reliable.

Systems Are Consistent

Given two infrastructure elements providing a similar service—for example, two application servers in a cluster—the servers should be nearly identical. Their system software and configuration should be the same, except for those bits of configuration that differentiate them, like their IP addresses.

Letting inconsistencies slip into an infrastructure keeps you from being able to trust your automation. If one file server has an 80 GB partition, while another has 100 GB, and a third has 200 GB, then you can't rely on an action to work the same on all of them. This encourages doing special things for servers that don't quite match, which leads to unreliable automation.

Teams that implement the reproducibility principle can easily build multiple identical infrastructure elements. If one of these elements needs to be changed (e.g., one of the file servers needs a larger disk partition), there are two ways that keep consistency. One is to change the definition so that all file servers are built with a large enough partition to meet the need. The other is to add a new class, or role, so that there is now an "xl-file-server" with a larger disk than the standard file server. Either type of server can be built repeatedly and consistently.

Being able to build and rebuild consistent infrastructure helps with configuration drift. But clearly, changes that happen after servers are created need to be dealt with. Ensuring consistency for existing infrastructure is the topic of Chapter 8.

Processes Are Repeatable

Building on the reproducibility principle, any action you carry out on your infrastructure should be repeatable. This is an obvious benefit of using scripts and configuration management tools rather than making changes manually, but it can be hard to stick to doing things this way, especially for experienced system administrators.

For example, if I'm faced with what seems like a one-off task like partitioning a hard drive, I find it easier to just log in and do it, rather than to write and test a script. I can look at the system disk, consider what the server I'm working on needs, and use my experience and knowledge to decide how big to make each partition, what filesystem to use, and so on.

The problem is that later on, someone else on my team might partition a disk on another machine and make slightly different decisions. Maybe I made an 80 GB /var partition using ext3 on

[6] CloudConnect CTO Randy Bias attributed this expression to former Microsoft employee Bill Baker, from his presentation "Architectures for Open and Scalable Clouds" (**http://www.slideshare.net/randybias/ architectures-for-open-and-scalable-clouds**). I first heard it in Gavin McCance's presentation "CERN Data Centre Evolution" (**http://www.slideshare.net/gmccance/cern-data-centre-evolution**). Both of these presentations are excellent.

[7] Sam Johnson described this view of the reliability of hardware and software in his article, "Simplifying Cloud: Reliability" (**http://samj.net/2012/03/08/simplifying-cloud-reliability/**).

one file server, but Priya made /var 100 GB on another file server in the cluster, and used xfs. We're failing the consistency principle, which will eventually undermine the ability to automate things.

Effective infrastructure teams have a strong scripting culture. If a task can be scripted, script it. If a task is hard to script, drill down and see if there's a technique or tool that can help, or whether the problem the task is addressing can be handled in a different way.

Design Is Always Changing

With iron-age IT, making a change to an existing system is difficult and expensive. So limiting the need to make changes to the system once it's built makes sense. This leads to the need for comprehensive initial designs that take various possible requirements and situations into account.

Because it's impossible to accurately predict how a system will be used in practice, and how its requirements will change over time, this approach naturally creates overly complex systems. Ironically, this complexity makes it more difficult to change and improve the system, which makes it less likely to cope well in the long run.

With cloud-age dynamic infrastructure, making a change to an existing system can be easy and cheap. However, this assumes everything is designed to facilitate change. Software and infrastructure must be designed as simply as possible to meet current requirements. Change management must be able to deliver changes safely and quickly.

The most important measure to ensure that a system can be changed safely and quickly is to make changes frequently. This forces everyone involved to learn good habits for managing changes, to develop efficient, streamlined processes, and to implement tooling that supports doing so.

Practices

The previous section outlined high-level principles. This section describes some of the general practices of infrastructure as code.

Use Definition Files

The cornerstone practice of infrastructure as code is the use of definition files. A definition specifies infrastructure elements and how they should be configured. The definition file is used as input for a tool that carries out the work to provision and/or configure instances of those elements. Example 1-1 is an example of a definition file for a database server node.

The infrastructure element could be a server; a part of a server, such as a user account; network configuration, such as a load balancer rule; or many other things. Different tools have different terms for this: for example, playbooks (Ansible), recipes (Chef), or manifests (Puppet). The term "configuration definition file" is used in this book as a generic term for these.

Example 1-1. Example of a definition file using a DSL

```
server: dbnode
  base_image: centos72
  chef_role: dbnode
  network_segment: prod_db
  allowed_inbound:
    from_segment: prod_app
    port: 1521
  allowed_inbound:
    from_segment: admin
    port: 22
```

Definition files are managed as text files. They may use a standard format such as JSON, YAML, or XML. Or they may define their own domain-specific language (DSL).[8]

Keeping specifications and configurations in text files makes them more accessible than storing them in a tool's internal configuration database. The files can also be treated like software source code, bringing a wide ecosystem of development tools to bear.

Self-Documented Systems and Processes

IT teams commonly struggle to keep their documentation relevant, useful, and accurate. Someone might write up a comprehensive document for a new process, but it's rare for such documents to be kept up to date as changes and improvements are made to the way things are done. And documents still often leave gaps. Different people find their own shortcuts and improvements. Some people write their own personal scripts to make parts of the process easier.

So although documentation is often seen as a way to enforce consistency, standards, and even legal compliance, in practice it's a fictionalized version of what really happens.

With infrastructure as code, the steps to carry out a process are captured in the scripts, definition files, and tools that actually implement the process. Only a minimum of added documentation is needed to get people started. The documentation that does exist should be kept close to the code it documents, to make sure it's close to hand and mind when people make changes.

The problems that Infrastructure as Code addresses are Server Sprawl, Snowflake Servers, Fragile Infrastructure, and Erosion. The Principles of Infrastructure as Code are: Systems Can Be Easily Reproduced; Systems Are Disposable; Cattle, Not Pets; Systems Are Consistent; Processes Are Repeatable; and Design Is Always Changing.

Automatically Generating Documentation

On one project, my colleague Tom Duckering found that the team responsible for deploying software to production insisted on doing it manually. Tom had implemented an automated deployment using Apache Ant, but the production team wanted written documentation for a manual process.

So Tom wrote a custom Ant task that printed out each step of the automated deployment process. This way, a document was

[8] As defined by Martin Fowler and Rebecca Parsons in *Domain-Specific Languages* (**http://martinfowler.com/books/dsl.html**) (Addison-Wesley Professional), "DSLs are small languages, focused on a particular aspect of a software system. You can't build a whole program with a DSL, but you often use multiple DSLs in a system mainly written in a general-purpose language." Their book is a good reference on domain-specific languages, although it's written more for people thinking about implementing one than for people using them.

generated with the exact steps, down to the command lines to type. His team's continuous integration server generated this document for every build, so they could deliver a document that was accurate and up to date. Any changes to the deployment script were automatically included in the document without any extra effort.

Version All the Things

The version control system (VCS) is a core part of infrastructure that is managed as code. The VCS is the source of truth for the desired state of infrastructure. Changes to infrastructure are driven by changes committed to the VCS.

The practices of Infrastructure as Code are Use Definition Files; Self-Documented Systems and Processes; Automatically Generating Documentation; Version All the Things; Continuously Test Systems and Processes; Small Changes Rather Than Batches; Keep Services Available Continuously; and Antifragility: Beyond "Robust"

Reasons why VCS is essential for infrastructure management include:

Traceability
VCS provides a history of changes that have been made, who made them, and ideally, context about why. This is invaluable when debugging problems.

Rollback
When a change breaks something—and especially when multiple changes break something—it's useful to be able to restore things to exactly how they were before.

Correlation
When scripts, configuration, artifacts, and everything across the board are in version control and correlated by tags or version numbers, it can be useful for tracing and fixing more complex problems.

Visibility
Everyone can see when changes are committed to a version control system, which helps situational awareness for the team. Someone may notice that a change has missed something important. If an incident happens, people are aware of recent commits that may have triggered it.

Actionability
VCSs can automatically trigger actions when a change is committed. This is a key to enabling continuous integration and continuous delivery pipelines.

Chapter 4 explains how VCS works with configuration management tools, and Chap-ter 10 discusses approaches to managing your infrastructure code and definitions.

Continuously Test Systems and Processes

Effective automated testing is one of the most important practices that infrastructure teams can borrow from software development. Automated testing is a core practice of high-performing development teams. They implement tests along with their code and run them continuously, typically dozens of times a day as they make incremental changes to their codebase.

It's difficult to write automated tests for an existing, legacy system. A system's design needs to be decoupled and structured in a way that facilitates independently testing components. Writing tests while implementing the system tends to drive clean, simple design, with loosely coupled components.

Running tests continuously during development gives fast feedback on changes. Fast feedback gives people the confidence to make changes quickly and more often. This is especially powerful with automated infrastructure, because a small change can do a lot of damage very quickly (aka DevOops, as described in "DevOops" on page 228). Good testing practices are the key to eliminating automation fear.

Chapter 11 explores practices and techniques for implementing testing as part of the system, and particularly how this can be done effectively for infrastructure.

Small Changes Rather Than Batches

When I first got involved in developing IT systems, my instinct was to implement a complete piece of work before putting it live. It made sense to wait until it was "done" before spending the time and effort on testing it, cleaning it up, and generally making it "production ready." The work involved in finishing it up tended to take a lot of time and effort, so why do the work before it's really needed?

However, over time I've learned to the value of small changes. Even for a big piece of work, it's useful to find incremental changes that can be made, tested, and pushed into use, one by one. There are a lot of good reasons to prefer small, incremental changes over big batches:

- It's easier, and less work, to test a small change and make sure it's solid.
- If something goes wrong with a small change, it's easier to find the cause than if something goes wrong with a big batch of changes.
- It's faster to fix or reverse a small change.
- One small problem can delay everything in a large batch of changes from going ahead, even when most of the other changes in the batch are fine.
- Getting fixes and improvements out the door is motivating. Having large batches of unfinished work piling up, going stale, is demotivating.

As with many good working practices, once you get the habit, it's hard to not do the right thing. You get much better at releasing changes. These days, I get uncomfortable if I've spent more than an hour working on something without pushing it out.

Keep Services Available Continuously

It's important that a service is always able to handle requests, in spite of what might be happening to the infrastructure. If a server disappears, other servers should already be running, and new ones quickly started, so that service is not interrupted. This

Every element of the infrastructure can be rebuilt quickly, with little effort; all systems are kept patched, consistent, and up to date; standard service requests, including provisioning standard servers and environments, can be fulfilled within minutes, with no involvement from infrastructure team members; SLAs are unnecessary; maintenance windows are rarely, if ever, needed; and changes take place during working hours, including software deployments and other high-risk activities.

is nothing new in IT, although virtualization and automation can make it easier.

Data management, broadly defined, can be trickier. Service data can be kept intact in spite of what happens to the servers hosting it through replication and other approaches that have been around for decades. When designing a cloud-based system, it's important to widen the definition of data that needs to be persisted, usually including things like application configuration, logfiles, and more.

The chapter on continuity (Chapter 14) goes into techniques for keeping service and data continuously available.

Antifragility: Beyond "Robust"

Robust infrastructure is a typical goal in IT, meaning systems will hold up well to shocks such as failures, load spikes, and attacks. However, infrastructure as code lends itself to taking infrastructure beyond robust, becoming antifragile.

Nicholas Taleb coined the term "antifragile" with his book of the same title (**http://www.amazon.com/Antifragile-Things-that-Gain-Disorder/dp/0141038225**), to describe systems that actually grow stronger when stressed. Taleb's book is not IT-specific—his main focus is on financial systems—but his ideas are relevant to IT architecture.

The effect of physical stress on the human body is an example of antifragility in action. Exercise puts stress on muscles and bones, essentially damaging them, causing them to become stronger. Protecting the body by avoiding physical stress and exercise actually weakens it, making it more likely to fail in the face of extreme stress.

Similarly, protecting an IT system by minimizing the number of changes made to it will not make it more robust. Teams that are constantly changing and improving their systems are much more ready to handle disasters and incidents.

The key to an antifragile IT infrastructure is making sure that the default response to incidents is improvement. When something goes wrong, the priority is not simply to fix it, but to improve the ability of the system to cope with similar incidents in the future.

The Secret Ingredient of Antifragile IT Systems

People are the part of the system that can cope with unexpected situations and modify the other elements of the system to handle similar situations better the next time around. This means the people running the system need to understand it quite well and be able to continuously modify it.

This doesn't fit the idea of automation as a way to run things without humans. Someday it might be possible to buy a standard corporate IT infrastructure off the shelf and run it as a black box, without needing to look inside, but this isn't possible today. IT technology and approaches are constantly evolving, and even in nontechnology businesses, the most successful companies are the ones continuously changing and improving their IT.

The key to continuously improving an IT system is the people who build and run it. So the secret to designing a system that can adapt as needs change is to design it around the people.[9]

Conclusion

The hallmark of an infrastructure team's effectiveness is how well it handles changing requirements. Highly effective teams can handle changes and new requirements easily, breaking down requirements into small pieces and piping them through in a rapid stream of low-risk, low-impact changes.

Some signals that a team is doing well:

- Every element of the infrastructure can be rebuilt quickly, with little effort.
- All systems are kept patched, consistent, and up to date.
- Standard service requests, including provisioning standard servers and environments, can be fulfilled within minutes, with no involvement from infrastructure team members. SLAs are unnecessary.
- Maintenance windows are rarely, if ever, needed. Changes take place during working hours, including software deployments and other high-risk activities.
- The team tracks mean time to recover (MTTR) and focuses on ways to improve this. Although mean time between failure (MTBF) may also be tracked, the team does not rely on avoiding failures.[10]
- Team members feel their work is adding measurable value to the organization.

[9] Brian L. Troutwin gave a talk at DevOpsDays Ghent in 2014 titled "Automation, with Humans in Mind" (**http://www.slideshare.net/BrianTroutwine1/automation-with-humans-in-mind-making-complex-systems-predictable-reliable-and-humane**). He gave an example from NASA of how humans were able to modify the systems on the Apollo 13 spaceflight to cope with disaster. He also gave many details of how the humans at the Chernobyl nuclear power plant were prevented from interfering with the automated systems there, which kept them from taking steps to stop or contain disaster.

[10] See John Allspaw's seminal blog post, "MTTR is more important than MTBF (for most types of F)" (**http://www.kitchensoap.com/2010/11/07/mttr-mtbf-for-most-types-of-f/**).

Kafka: The Definitive Guide

Real-Time Data and Stream Processing at Scale

Book notes by Brian Hitchcock

Brian Hitchcock

Details

Author: Neha Narkhede, Gwen Shapira, and Todd Palino

ISBN-13: 978-1-491-93616-0

Publication Date: September 29, 2017

Publisher: O'Reilly Media

Summary

I had not worked with Kafka or any stream-processing software before reading this book. If you already know that you need to process stream data and you've already chosen Kafka as your solution, this book appears to be the definitive guide.

Foreword

We are told that Kafka is being used by thousands of organizations and is part of a movement to manage streams of data. Kafka came from internal systems at LinkedIn, where they had systems to store data but could not process continuous flows of data. Earlier attempts used messaging systems, log aggregation, and ETL tools, none of which really worked. Data is always growing and evolving, and it has become a continuous stream. Kafka is a streaming platform that supports publishing and subscribing to data streams as well as storing and processing them. Kafka is a powerful abstraction for creating applications. It is like a messaging system, but it has three important differences: it is distributed (it runs as a cluster and scales); it stores data as long as you need to; and it provides stream processing (it can compute derived streams dynamically with less code).

Kafka was designed to be a real-time version of Hadoop. It can be seen as a superset of the batch processing usually done with Hadoop, featuring the continuous, low-latency functionality needed by near-real-time business applications. Making use of streams of data requires a mind shift compared to a world of request-and-response systems and relational databases. Clearly, relational databases have been moved into the natural history museum next to the diorama depicting cave people. Please come see me: I'm next to the pterodactyls and the punch card reader.

Preface

Popular use cases for Kafka are described: as a message bus for event-driven microservices, for stream processing, and for large-scale data pipelines.

We are told that this book is for software and production engineers as well as data architects. It is assumed that the reader has some knowledge of Java and Linux. The book is also written for those who want to know what guarantees Kafka offers to provide support for managers who move to Kafka.

Chapter 1—Meet Kafka

This chapter starts off with a philosophical tone. The authors state that data is used by every business, and all of our applications create data. Okay so far. Then they assert that every byte of data is important, but here I disagree: I think that most, perhaps 80%, of what is stored in the cloud isn't worth much, if anything. I realize that perhaps even this data can be analyzed in some way, but I remain skeptical. I think we store a lot more data than we will ever make use of, and we do this because it's cheaper to store data than it is to determine which data is worth keeping. I think the value of most of the data we store is vanishingly small, but time will tell.

Next, we have a description of publish/subscribe messaging with diagrams, and we see how the problem starts, with individual queue systems that create lots of duplication. Kafka solves this problem by providing a distributed commit log and a distributing streaming platform. Data in Kafka is stored in order and stored safely so it can be found as needed.

Messages and batches are covered next. The unit of data in Kafka is a message, similar to a database row. A message in Kafka is an array of bytes that may have a metadata key. Messages are written in batches for efficiency. Schemas can be used to provide

"LinkedIn had problems with its custom collectors. Data was not collected often enough; the systems required lots of human intervention, and many different systems were gathering and providing different kinds of data. Kafka was developed to address these issues—specifically, to decouple producers and consumers using a push-pull model, support persistence for message data, provide high throughput, and support horizontal scaling."

structure to the messages. These can be JSON, XML, or Apache Avro, a serialization framework from Hadoop. It is very important to use a consistent data format so that writing and reading messages don't have to be tightly coupled.

Topics are categories of messages and are made up of partitions, and messages are ordered within each partition. Partitions can be on different servers for scaling. A stream is a single topic of data, which may have many partitions.

Producers create messages and consumers read them in order by subscribing to topics. Within each partition, the offset is also metadata, an increasing integer value that shows where a message is located in a partition. Consumers can read, stop reading, and then start reading again where they left off.

Consumers belong to consumer groups; each partition is read by only one consumer, but multiple partitions can be read by a single consumer and we see multiple diagrams illustrating all of this. A broker is a single Kafka server that handles the messages. Clusters are made up of multiple brokers, with one being the cluster controller. If the controller fails, there is a protocol for the remaining brokers to choose a new controller. Sounds like Oracle RAC, doesn't it?

Retention is a key feature of Kafka and can be specified by time and topic size. Defaults are set for time, perhaps 7 days, or topic size, perhaps 1 GB. Multiple clusters are used for separating types of data, security, and disaster recovery. Kafka includes Mirror Maker for replicating data within a cluster, and there are diagrams explaining different configuration options.

The next section poses the question "Why Kafka?" but the answer didn't work for me. The reasons given are that Kafka handles multiple producers and consumers without interference; it features disk-based retention; and it is scalable and high performance, supporting subsecond message latency from producer to consumer. These are all great reasons to use Kafka but only if we assume there are no other products that offer any of these features. What evidence is there to support the claims of scalability and performance? Is there any standard test to measure stream processing? I don't think this is enough for me to go to my management and explain why we must move to Kafka.

Kafka is described as the circulatory system of the data ecosystem and we have more figures. The following use cases are described: activity tracking, messaging, metrics and logging, commit log, and stream processing.

I was hoping that one particular topic would be covered: Kafka's origin story. It started when LinkedIn had problems with its custom collectors. Data was not collected often enough; the systems required lots of human intervention, and many different systems were gathering and providing different kinds of data. Kafka was developed to address these issues—specifically, to decouple producers and consumers using a push-pull model, support persistence for message data, provide high throughput, and support horizontal scaling. Another feature is that Kafka is open source which means there is a large community working to improve the software all the time. The name was chosen because Kafka was a writer and this software writes lots of messages—which, we are told, isn't a very good reason for the name.

Chapter 2—Installing Kafka

To start the process, you install Apache Kafka broker and Apache Zookeeper for storing broker metadata. First you must choose an operating system; Linux is recommended. When you install Java, it should be Java 8. Next is to install Zookeeper, and there are diagrams and code samples for this task. The Kafka broker is then installed and configured, followed by setting up topic defaults and the number of partitions for each topic.

Hardware selection is discussed with sections on disk throughput, capacity, and memory. Kafka should run on a dedicated system. Networking and CPU are issues as well. It was interesting to see options for running Kafka in the cloud using Amazon Web Services (AWS) as an example. Diagrams show the setup for a single cluster and multiple brokers. For virtual memory it is generally recommended that you don't configure any swap space, but this can be a bad thing on recent versions of Linux. Filesystem choices and networking configuration options are reviewed.

"It is like a messaging system, but it has three important differences: it is distributed (it runs as a cluster and scales); it stores data as long as you need to; and it provides stream processing (it can compute derived streams dynamically with less code)."

When you move to production, Garbage Collection (GC) concerns and options need to be reviewed. You also need to consider the data center layout; ideally you would have brokers in different physical racks so that the failure of one rack means only losing one broker.

This chapter shows us that Kafka has lots of tuning options. Just as the database becomes autonomous, we now need to tune the messaging system. I guess this is progress?

I expected a discussion of in-memory operations as an option under disk storage, but it wasn't covered. I wonder if anyone is running Kafka totally in memory as some databases are these days.

Chapter 3—Kafka Producers: Writing Messages to Kafka

You may use Kafka as a message bus, a queue, or a data storage system, but in all cases you create a producer to write data to Kafka and a consumer to read that data. You may create an application that does both.

A credit card transaction processing system is described as an example.

There are sections covering, with diagrams, how to set up the producer, construct a Kafka producer and required properties, send a message to Kafka with code samples, and send a message synchronously and asynchronously.

When configuring producers, the ACKS parameter controls what, if any, reply is required from the broker before a message is considered sent. Other parameters include buffer memory, compression type, retries, and batch size. Kafka comes with serializers that support strings, integers, and byte arrays. For other data types you need to use a custom serializer. Debugging compatibility issues between serializer versions can be difficult. Using Avro records with Kafka is described, including diagrams and code samples. Other items that are covered include partitions, custom partition strategy, and older producer APIs that shouldn't be used.

Chapter 4—Kafka Consumers: Reading Data from Kafka

Consumers are used by applications that need to read data from Kafka.

We start with Kafka consumer concepts and consumer groups. There are diagrams showing different ways that consumers in groups relate to topics and partitions. We are told that Kafka scales, without a performance hit, to support many more consumers and groups than previous messaging systems. I would have liked more details and metrics on this assertion, but none are presented.

The next section covers consumer groups and partition rebalance. Rebalance is a short window during which consumers can't read messages when, for example, some part of the messaging system fails.

Heartbeats are sent to the broker to maintain which consumers are part of which groups and which partitions they own. Note that heartbeat behavior has changed in more recent versions of Kafka. The process of assigning partitions to brokers is described.

Next is how to create a Kafka consumer and subscribe to topics.

The consumer uses a poll loop to find out if the server has more data and a code sample of this loop is provided. When configuring consumers, there are many parameters to set up.

In the description of commits and offsets we learn that Kafka is unique in that it does not track consumer acknowledgments as JMS queues do. Consumers use the offset to track their location in a partition. When the current position of the offset is updated, this is a commit. When the consumer restarts reading messages, it locates the last committed offset and starts reading from there. Several diagrams are presented to explain all of this.

There are multiple ways a client application can choose to handle commits, and each choice has consequences. For example, messages may be read multiple times or not at all during a recovery after a failure.

When you commit, the current offset involves tradeoffs and choices between manual or asynchronous or some of both. We see code samples for several options. There are sections covering commit specified offset, rebalancing listeners, consuming records with specific offsets, and exiting the pool loop.

Consumers use deserializers, and the Kafka developer must track which serializer(s) were used to write to each topic and only use compatible deserializers in the corresponding consumers. There will be support issues down the line: what if one of your publishers changes serializer?

You may need custom deserializers, and while code samples are shown, this is not recommended as it can make support more complicated.

You can use a standalone consumer that is not part of a consumer group; the pros and cons of this choice are covered.

Finally, commit processing is complicated. What if different parts of your business do it differently? How do you know what the impact will be on whatever business app uses all this data, some of which may be repeated and some of which may be lost during recovery from a failure?

Chapter 5—Kafka Internals

The authors tell us that while is it not necessary to understand the material presented here, it helps with troubleshooting. There are three topics: replication, requests from producers and consumers, and how files and indexes are stored.

First up under replication is how cluster membership works and how the controller elects partition leaders. Replication is at the heart of the Kafka architecture, which can be thought of as a distributed, replicated commit log service.

Many diagrams are shown to explain request processing. How fetch requests from clients are processed is shown. Kafka is known for using a zero-copy method to send messages directly from the file to the network channel. This means there are no buffers involved, which improves performance.

The basic storage unit of Kafka for physical storage is the partition replica. We see more discussion of rack information with diagrams. The retention period for file management is discussed. The default is that each segment can store 1 GB of data or a week of data; after that, a new file is created. The same data format is used on disk as is used by producers to send and consumers to read messages. This supports the zero-copy optimization since no compression or decompression is needed.

Kafka maintains an index for each partition. Compaction, where Kafka keeps the latest value for each key in a topic and all the other events are deleted, is explained with diagrams.

I found several things interesting here. For all our virtualization and our totally cloud-based lifestyle, we still need to know which server is in which rack. I'm not clear how this works with the major cloud hosting vendors. Does an upgrade move brokers to different racks? What about a failover in the data center? Lots of tuning options and requirements. Assuming that your database is autonomous, all the resources now freed up can be assigned to Kafka tuning!

Chapter 6—Reliable Data Delivery

This chapter starts with a great quote: "Because of its flexibility, it is also easy to accidentally shoot yourself in the foot when using Kafka." As with all things in life, the shooting of the foot remains a big issue. The assumption of system reliability must be thoroughly tested. No word on exactly how to do this or what amount of resources it will take. Here we learn about the reliability guarantees. ACID is the standard in the relational database world, and we learn about the similar concepts in the Kafka world. Kafka guarantees the order of messages in a partition. Messages are committed when written to all in-sync replicas; once committed, messages are not lost while one or more replicas exist, and consumers can only read committed messages.

Kafka allows administrators and developers to decide what level of reliability is needed, and replication is central to the reliability guarantees. Topics are broken down into partitions and each partition is stored on a single disk. I'm not sure what "single disk" means in today's virtualized cloud storage systems, and I'm not clear that we can know exactly where anything is being stored physically.

There are sections on broker configuration and replication factor. It turns out there is a minimum required for in-sync replicas. You can insert your favorite boy band joke here.

Next we learn about using producers in a reliable system. This involves using the correct ACKS configuration for your reliability requirements, handling errors due to configuration and code, sending acknowledgements, configuring producer retries, and additional error handling.

Similarly, there are issues for using consumers in a reliable system. This involves consumer configuration properties and dealing with all the details around explicitly committing offsets.

You must validate system reliability, and this includes the configuration and applications. We are also told we must monitor reliability in our production systems. For both the validation and monitoring tasks I would have liked to hear how this is done in a real-world case.

All this talk of reliability and commit logs and various guarantees sounds like the complexities of a database system. I also have questions about the administrator; the way it is described assumes there is a single admin that knows all the settings and why they are what they are. How does the admin task scale when you have lots of topics all with different levels of reliability? One of the examples discussed is a bank. How hard would it be to dial down the reliability to let a few transactions get lost—and then dial it back up again? Perhaps those "lost" transactions send rather large bags of money to faraway places? How is this handled by the very large organizations currently using Kafka?

Chapter 7—Building Data Pipelines

Use cases for data pipelines are described: first, instances in which Kafka itself is one end of the two endpoints, and second, where Kafka is the intermediate link between two different systems. APIs were added to Kafka so that users did not have to create their own API from scratch.

Various considerations that arise when building data pipelines are discussed. Timeliness is one: do you expect data to arrive in a large set once a day or very shortly after it is generated? In this context, "very shortly" means milliseconds. Kafka's scalable, reliable storage can support both of these timeliness requirements. Kafka is a giant buffer between producers and consumers—producers that may produce data in real time and consumers that take in data once a day.

Other considerations include reliability, high and varying throughput, and data formats. A pipeline must deal with differing data types and formats. Transformations are an issue, as is security. Data in the pipeline should be encrypted, and you need to consider who can access the data while it is in transit as well as how the pipeline authenticates to the endpoints. Other considerations are coupling and agility, and while a pipeline should decouple the source and target of the data, this can be degraded. Pipelines can have problems such as loss of metadata through the pipeline and too much processing in the pipeline.

When you can embed the Kafka clients into your own application code, they are the best way to write to and read from Kafka. When you need to interact with data stores you can't modify, you use Kafka Connect.

We learn how to run Kafka Connect, and Kafka Connect examples are shown. These include file source and file sink and MySQL to Elasticsearch, and for both we see extensive code examples. Many more details about Kafka Connect are covered. To implement pipelines, we need to code connectors. This sounds like a lot of custom coding just to get this going, and even more as we add more sources and subscribers. How do we know if this development burden and the ongoing support effort are worth it?

Chapter 8—Cross-Cluster Data Mirroring

Most of the time we are told to use a single Kafka cluster, but now we see some exceptions. Examples are departments that require their own clusters and different requirements for different use cases. When needed, moving data between clusters is called "mirroring." You'd be right that this would normally be called "replication," but that term was already used when discussing moving data between nodes in a cluster. Kafka provides MirrorMaker to support this.

We see various use cases of cross-cluster mirroring: regional and central clusters, redundancy for disaster recovery and cloud migrations. Multicluster architectures are reviewed. Next to be discussed are the realities of cross–data center communication: high latencies, limited bandwidth, and higher costs. There are diagrams for different architecture options, such as active-active and active-standby. You need to test your failover solution. Netflix created Chaos Monkey to randomly create disasters. Data loss and inconsistencies can occur with an unplanned failover, and lots of diagrams are used to explain what can happen.

"You may use Kafka as a message bus, a queue, or a data storage system, but in all cases you create a producer to write data to Kafka and a consumer to read that data. You may create an application that does both."

Finally, we have coverage of Apache Kafka's MirrorMaker and other cross-cluster mirroring solutions.

Chapter 9—Administering Kafka

On the very first page of this chapter we have a highlighted box with the title "Authorizing Admin Operations." Security is critical, so I was surprised to read the following paragraph, which I quote in its entirety:

"While Apache Kafka implements authentication and authorization to control topic operations, most cluster operations are not yet supported. This means that these CLI tools can be used without any authentication required, which will allow operations such as topic changes to be executed with no security check or audit. This functionality is under development and should be added soon."

I quote this entire paragraph because I want you to see all of it just as it is in the book. I'm new to Kafka—perhaps there is some nuanced explanation of why this isn't a big problem. Can you imagine a mainstream software product that is being used to process credit card data that doesn't have any audit features?

There are sections covering each of the following topic operations: creating a new topic, adding partitions, deleting a topic, consumer groups, deleting a group, offset management, dynamic configuration changes, partition management, and replication configuration.

At the end of this chapter we have another quote you need to think about: "Running a Kafka cluster can be a daunting endeavor." I'm still stuck on the no security check and no audit part. Given the list of topic operations that are discussed in this chapter, does this mean all of these operations can be done without any auditing? I was hoping there would be a detailed discussion of why this isn't a big problem. I would have liked to hear why this isn't a concern for LinkedIn. I did not find answers to either of these questions.

Chapter 10—Monitoring Kafka

There are so many available metrics that it can easily become confusing. With that warning we start looking at monitoring. The things that can be monitored range from simple, like overall traffic rates, to the very detailed timing of every request type. First we look at metric basics and the basics of monitoring a Java application. Next we see where the metrics are found. We also learn about internal or external measurements, application health checks, and alert fatigue. It is easy to monitor too much and generate too many alerts. Those responsible for monitoring will burn out.

In the section covering Kafka broker metrics we see that the most important metric is under-replicated partitions, which are replicas that are not caught up to the lead partition. The main cluster problems to watch for are unbalanced load and resource exhaustion. We have yet another quote you need to read: "[B] alancing traffic within a Kafka cluster can be a mind-numbing process."

There are many metrics that can be monitored for the broker, topics, and partitions. JVM and OS monitoring are covered. Disk is by far the most important subsystem for Kafka performance. In addition to monitoring, there is logging; there is also coverage of lag and end-to-end monitoring. At the end of this chapter, I see that many organizations use Kafka for petabyte-scale data flows. I need to think about what that means.

Chapter 11—Stream Processing

Kafka provides more than just a reliable source of data streams; it also provides powerful stream processing. Stream processing, we are told, is often misunderstood. To start explaining stream processing, we start with the question of what a data stream is. A data stream is also called an "event stream" and is an abstraction representing an unbounded dataset. Event streams are ordered; once events have happened they cannot be changed.

Event streams are replayable; this sets Kafka apart as it provides capturing and replaying of event streams. Stream processing can be request-response like OLTP or batch-processing. Stream processing is needed because most business processes don't need OLTP-like speed and don't want to wait for a daily batch job either. Note that stream processing must be continuous and ongoing

Within stream processing, the most important concept is time—which is also the most confusing. There is event time, log append time, and processing time. It is important that a data pipeline uses a single time zone. State is the information that is stored between events.

Stream-table duality is next and, in my opinion, this is the best part of this book. This paragraph gave me a clear understanding of why I should care about stream processing: "Unlike tables, streams contain a history of changes. Streams are a string of events wherein each event caused a change. A table contains a current state of the world, which is the result of many changes. From this description, it is clear that streams and tables are two sides of the same coin—the world always changes, and sometimes we are interested in the events that caused those changes, whereas other times we are interested in the current state of the world. Systems that allow you to transition back and forth between the two ways of looking at data are more powerful than systems that support just one."

This is really good information, and it provides a clear explanation of why stream processing is important.

There is coverage of converting a stream to a table, which is called "materializing the stream." There are stream-processing design patterns, including single-event processing and multiphase processing/repartitioning; processing with external lookup (stream-table join); streaming join; out-of-sequence events; and reprocessing.

In the section on Kafka streams by example we have code samples for applications for word count, stock market statistics, and click stream enrichment.

An architecture overview of Kafka streams is next and includes sections on scaling the topology and surviving failures.

Stream processing use cases covers customer service, the internet of things, and fraud detection.

This chapter ends with a discussion of how to choose a stream-processing framework. You should consider the types of applications you are supporting, your response time requirements, and whether you need real-time or asynchronous processing.

I would put this chapter first; I'd start with why stream processing is cool and follow with why Kafka is the best way to do it.

Conclusion

I did not end up with a definitive understanding of how to decide whether or not I need stream processing or why, exactly, Kafka would be the best solution. Perhaps it should be obvious to me: if thousands of organizations are using Kafka, I guess my business should as well. But are we developing software because of what we really need or are we all chasing the latest popular point solution? Are we building our enterprise systems to fight the last business war or really preparing for the future?

How many current software solutions come from a small number of huge organizations, so large that they can dictate all aspects of their environment? Should all the businesses outside of this group of behemoths be trying to apply the same solutions? Does something that works for LinkedIn necessarily work for a business that is nowhere as big?

I thought about the petabyte-scale data flows and how Kafka can replay data flows: how much data are we storing and for how long? Petabytes for each data flow, how many data flows, for a week, a month? How long does it take to replay these petabyte-scale data flows? Days, weeks? How do we decide which data flows to keep and which to delete? Do we have a central repository of the huge data flows? So many questions!

Since I read this book in Safari, I can search the full text. Searching for Oracle we find only four references in the book, and two of those are to Oracle Java. Oracle is mentioned only once in the entire book as a data source. I'm just sayin'. ▲

Brian Hitchcock works for Oracle Corporation where he has been supporting Fusion Middleware since 2013. Before that, he supported Fusion Applications and the Federal OnDemand group. He was with Sun Microsystems for 15 years (before it was acquired by Oracle Corporation) where he supported Oracle databases and Oracle Applications. His contact information and all his book reviews and presentations are available at **www.brianhitchcock.net/oracle-dbafmw/.** *The statements and opinions expressed here are the author's and do not necessarily represent those of Oracle Corporation.*

Making Changes: The Real Promise of Infrastructure as Code

by Paul Marcelin

Paul Marcelin

In the first article of the series—in the August 2018 issue of the *NoCOUG Journal*—we learned how to specify a managed Oracle database server in CloudFormation, Amazon's infrastructure-as-code system, and then we launched the server. That is useful in and of itself, particularly if you have to set up lots of database servers or if you and your colleagues want to adopt a standard template. But the infrastructure-as-code concept goes beyond creating resources: you will also want to update them. In the present article, I introduce CloudFormation change sets, which I see as the true promise of infrastructure as code. To prepare for the change set exercises, please re-create the sample stack from the first article before you continue reading.

Simple: Parameter Change

Imagine that you have forgotten the DBA password. Panic sets in. Then you remember that it was one of the parameters you set when you created the database using the CloudFormation template.

You also remember that you can review parameter values in the CloudFormation console. Go to **https://console.aws.amazon.com**, log in, navigate to Services → CloudFormation → Stacks, click the name of your stack, and then click the arrow to open the Parameters section.

Drat, DbMasterUserPassword is masked as ***! Why is this so, when all of the other parameter values are visible? Go back and check the CloudFormation template, at **https://github.com/sqlxpert/infra-as-code-aws-nocoug-journal/blob/master/cloudformation/0-aws-rds-oracle-all-in-one.yaml**. In case the reason is not apparent at a glance, I will give the answer at the end of the article.

The only option left is to reset the password, and you can do so by creating a minimal CloudFormation change set.

1. In the CloudFormation Console, click to place a checkmark in the box to the left of your stack's name.
2. Above the list of stacks and to the right of the blue Create Stack button, click Actions. A pop-up menu will appear. Select "Create Change Set For Current Stack."
3. Click the blue Next button. In this exercise, you are not changing the template; you are only changing the value of a parameter.
4. Under Specify Details, type a name and a description for the change set. Don't put too much thought into these values, because they are not stored permanently. The change set name and description matter only until the change set has either been executed successfully or rolled back.
5. Under Parameters, leave all values unchanged *except* DbMasterUserPassword. Take note of the new password that you type in.
6. Click the blue Next button.
7. There is no need to make any entries on the Options page. Click the blue Next button again.
8. Check the Review page (although in this case, the new password will merely show up as .....) before clicking the blue "Create change set" button.
9. Wait for CloudFormation to finish computing changes.
10. Scroll down to the Changes section and check the line items. Pay special attention to the Replacement column on the right. Many changes to AWS resources can be made in place, but some require deleting and re-creating a resource. If a resource from one CloudFormation stack is referenced by another stack or is referenced in AWS at large, it cannot be deleted and your change set would fail to execute. Thankfully, changing the DBA password is a safe operation.
11. Click the Execute button near the top right.
12. Monitor progress by clicking on the stack's name and checking the Events section. You can cancel the changes by clicking Cancel Update Stack at the top right. (This is useful because some errors cause CloudFormation to hang. Be alert for the possibility if no new events have appeared for several minutes.)
13. If execution fails for any reason, CloudFormation automatically rolls all resources back to their initial state. Clues about the error appear in the Events section. If no error message appears in the Status Reason column, look for a resource with an "initiated" event but no matching "complete" event. After an execution failure, you must manually delete the change set and create another.

14. If execution succeeds, CloudFormation changes the stack's status to UPDATE_COMPLETE and automatically deletes the change set. In this case, the DBA password has been reset.

Complex: Template Change

Now, imagine that your managed Oracle database has, due to its performance and reliability, seen lots of use. It is now almost out of space. To enlarge the disk, you will edit the CloudFormation template and create a change set.

1. In the CloudFormation Console, click to place a checkmark in the box to the left of your stack's name.
2. Above the list of stacks and to the right of the blue Create Stack button, click Actions. A pop-up menu will open. Select "Create Change Set For Current Stack."
3. Under Use Current Template, click "View/Edit template in Designer."
4. The panel at the bottom displays an editable version of your original CloudFormation template. Scroll down to—or search for (Windows: Control+F; MacOS: Command+F)—AllocatedStorage. Change the value from 20 to 50. For reference, an edited version of the template is available at **https://github.com/sqlxpert/infra-as-code-aws-nocoug-journal/blob/master/cloudformation/1-aws-rds-oracle-disk-bigger.yaml**. diff is a useful tool for comparing old and new template files.
5. Click the checkmark in the grey bar near the top of the window. This validates the template's syntax.
6. Click the cloud in the grey bar. This uploads the template.
7. Notice that "Use current template" has been deselected and "Specify an Amazon S3 template URL" has been selected instead. Your new template has been uploaded successfully.
8. Click the blue Next button.
9. Under Specify Details, type a name and a description for the change set. In this exercise, you have edited the template itself; there are no changes to be made in the Parameters section.
10. Click the blue Next button.
11. There is no need to make any entries on the Options page. Click the blue Next button again.
12. Check the Review page (although in this case there are no parameter changes to see) before clicking the blue "Create change set" button.
13. Wait for CloudFormation to finish computing changes.
14. Scroll down to the Changes section and check the line items. Enlarging the disk is a safe operation from CloudFormation's perspective. In our low-cost, non-production, single-availability-zone example, the operation will take some time and will interrupt access to the database. Because those properties are particular to the Relational Database Service (RDS), CloudFormation, which is a separate AWS service, cannot warn about them.
15. Click the arrow to open the Details section. Notice the reference to AllocatedStorage, exactly the property that you are changing. In my experience, the more numerous and more complex the changes, the less intelligible the Details section will be.
16. Click the Execute button near the top right.
17. Monitor progress by clicking on the stack's name and checking the Events section.
18. If execution fails, CloudFormation automatically rolls all resources back to their initial state.
19. If execution succeeds, CloudFormation reports UPDATE_COMPLETE. In this case, the disk has been enlarged.
20. To minimize AWS charges, please use the CloudFormation console to delete the stack.

Benefits of Change Sets

As you have seen from the exercises, change sets facilitate *controlled* updates. Before you execute a change set, CloudFormation informs you of the scope of the changes. You can create a change set in advance and execute it later. An orchestration tool—which goes beyond the scope of CloudFormation—is useful if you want to schedule change set execution or if you want to change many stacks.

Although you probably didn't experience an execution failure when completing the exercises, if ever you do, CloudFormation will roll back the changes. Also not apparent from the exercises is the possibility of using AWS Identity and Access Management (IAM) to grant some people the right to create stacks and others only the right to make changes.

Epilogue: Infrastructure as Code Can Replace Managed Services

NoCOUG's Iggy Fernandez posits that the infrastructure as code idea eliminates the need for managed services like AWS RDS. I think he is right and that he has defined the measure of completeness for an infrastructure-as-code system. Today, AWS CloudFormation lacks a *good* way to control the installation and configuration of operating system and application software on arbitrary, unmanaged servers. Generic alternatives like SaltStack excel at that task but don't support the full range of AWS services, resource types, and properties. I challenge cloud service providers and configuration management system designers to meet in the middle. The ideal configuration management system will be able to accomplish everything that a human can do in the AWS Console (just like CloudFormation) and at the Linux command line (just like SaltStack, Chef, etc.). ▲

Paul Marcelin can be reached at **marcelin@alumni.cmu.edu**. *Please send suggestions for the next infrastructure-as-code article. Should it be about AWS CloudFormation fine points? Microsoft Azure? Oracle Cloud?*

Answer to question about masked password: In the original CloudFormation template, I specified NoEcho: true when I defined the DbMasterUserPassword parameter. See **https://github.com/sqlxpert/infra-as-code-aws-nocoug-journal/blob/master/cloudformation/0-aws-rds-oracle-all-in-one.yaml#L25**.

Singing the NoCOUG Blues

with Tim Gorman

Tim Gorman

Editor's note: RMOUG president Tim Gorman recently stepped away from the Rocky Mountain Oracle Users Group (RMOUG) after 26 years as a member and 23 years on the board with 9 of those years as president. This interview with him was originally printed in the August 2014 issue of the *NoCOUG Journal*.

You are old, father Gorman (as the young man said) and your hair has become very white. You must have lots of stories. Tell us a story!

Well, in the first place, it is not my hair that is white. In point of fact, I'm as bald as a cue ball, and it is my skin that is pale from a youth misspent in data centers and fluorescent-lit office centers.

It is a mistake to think of wisdom as something that simply accumulates over time. Wisdom accumulates due to one's passages through the world, and no wisdom accumulates if one remains stationary. It has been said that experience is what one receives soon after they need it, and experience includes both success and failure. So wisdom accumulates with experience, but it accumulates fastest as a result of failure.

About four years ago, or 26 years into my IT career, I dropped an index on a 60 TB table with 24,000 hourly partitions; the index was over 15 TB in size. It was the main table in that production application, of course.

Over a quarter-century of industry experience as a developer, production support, systems administrator, and database administrator: if that's not enough time to have important lessons pounded into one's head, then how much time *is* needed?

My supervisor at the time was amazing. After the shock of watching it all happen and still not quite *believing* it had happened, I called him at about 9:00 p.m. local time and told him what occurred. I finished speaking and waited for the axe to fall—for the entirely justified anger to crash down on my head. He was silent for about 3 seconds, and then said calmly, "Well, I guess we need to fix it."

And that was it.

No anger, no recriminations, no humiliating micro-management. We launched straight into planning what needed to happen to fix it.

He got to work notifying the organization about what had happened, and I got started on the rebuild, which eventually took almost 2 weeks to complete.

It truly happens to all of us. And anyone who pretends otherwise simply hasn't been doing anything important.

How did I come to drop this index? Well, I wasn't *trying* to drop it; it resulted from an accident. I was processing an approved change during an approved production outage. I was trying to disable a unique constraint that was supported by the index. I wanted to do this so that a system-maintenance package I had written could perform partition exchange operations (which were blocked by an enabled constraint) on the table. When I tested the disabling of the constraint in the development environment, I used the command ALTER TABLE . . . DISABLE CONSTRAINT and it indeed disabled the unique constraint without affecting the unique index. Then I tested the same operation again in the QA/Test environment successfully. But when it came time to do so in production, it dropped the index as well.

Surprise!

"You are old, Father William," the young man said,
"And your hair has become very white;
And yet you incessantly stand on your head—
Do you think, at your age, it is right?"

"In my youth," Father William replied to his son,
"I feared it might injure the brain;
But now that I'm perfectly sure I have none,
Why, I do it again and again."

I later learned that the unique constraint and the supporting unique index had been created out of line in the development and QA/test environments. That is, first the table was created, then the unique index was created, and finally the table was altered to create the unique constraint on the already-existing unique index.

But in production, the unique constraint and the supporting unique index had been created in-line. When the table was created, the CREATE TABLE statement had the unique constraint within, along with the USING INDEX clause to create the unique index.

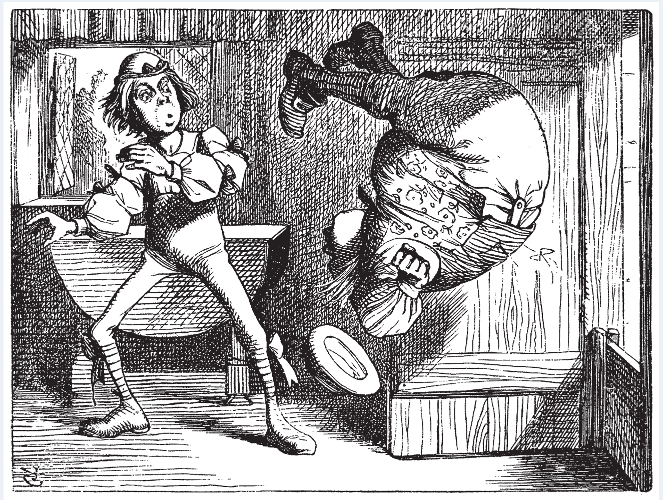
So when I altered the table in production, disabling the constraint also caused the index to be dropped.

After the mishap, I found the additional syntax for KEEP INDEX, which could have been added to the end of the ALTER TABLE . . . DISABLE CONSTRAINT command because Oracle recognized the difference in default behaviors.

But that was a discovery I experienced after I needed it.

As to why my supervisor was so calm and matter-of-fact throughout this disaster, I was not surprised; he was always that way, it seemed. What I learned over beers long after this incident is that, in his early life, he learned the true meaning of the words "emergency" and "catastrophe." He was born in Afghanistan, and he was a young child during the 1980s after the Soviets invaded. His family decided to take refuge in Pakistan, so they sought the help of professional smugglers, similar to what we call "coyotes" on the Mexican-American border. These smugglers moved through the mountains bordering Afghanistan and Pakistan at night on foot, using camels to carry baggage and the very old, the sick and injured, and the very young.

My supervisor was about 9 years old at the time, so the smugglers put him on a camel so he would not slow them down.



"You are old," said the youth, "As I mentioned before,
And have grown most uncommonly fat;
Yet you turned a back-somersault in at the door—
Pray, what is the reason of that?"

"In my youth," said the sage, as he shook his grey locks,
"I kept all my limbs very supple
By the use of this ointment—one shilling the box—
Allow me to sell you a couple?"

During the night, as they were crossing a ridge, they were spotted by the Soviets, who opened fire on them using machine guns with tracer bullets. Everyone in the caravan dove to the ground to take cover. Unfortunately, they all forgot about the 9-year-old boy on top of the 8-foot-high camel. My supervisor said he saw the bright tracer bullets arching up toward him from down below in the valley, passing over his head so close that he felt he could just reach up and grab them. He wanted to jump down, but he was so high off the ground he was terrified. Finally, someone realized that he was exposed and they pulled him down off the camel.

As he told this story, he laughed and commented that practically nothing he encountered in IT rose to the level of what he defined as an emergency. The worst that could happen was no more catastrophic than changing a tire on a car.

I've not yet been able to reach this level of serenity, but it is still something to which I aspire.

We love stories! Tell us another story!

A little over 10 years ago, I was working in downtown L.A. and arrived in the office early (5:00 a.m.) to start a batch job. I had a key card that got me into the building and into the office during the day, but I was unaware that at night they were locking doors in the elevator lobby. I banged on the doors and tried calling people, to no avail. Finally, after a half-hour, out of frustration, I grabbed one of the door handles and just *yanked* hard.

It popped open.

I looked at it in surprise, thought "sweet!", walked in to the cubicle farm, sat down, and started my batch job. All was good.

Around 7:00 a.m., the LAPD showed up. There were about a dozen people in the office now, so the two officers began questioning folks nearest the door. From the opposite side of the room, I stood up, called out "Over here," and 'fessed up.

They told me that if I hadn't called them over immediately, they would have arrested me by the time they got to me. Have a nice day, sir.

The NoCOUG Blues

NoCOUG membership and conference attendance have been declining for years. Are user groups still relevant in the age of Google? Do you see the same trends in other user groups? What are we doing wrong? What can we do to reverse the dismal trend? Give away free stuff like T-shirts and baseball caps? Bigger raffles? Better food?

Yes, the same trends are occurring in other users groups. IT organizations are lean and can't spare people to go to training. The IT industry is trending older as more and more entry-level functions are sent offshore.

Users groups are about education. Education in general has changed over the past 20 years as online searches, blogs, and webinars have become readily available.

The key to users groups is the quality of educational content that is offered during live events as opposed to online events or written articles. Although online events are convenient, we all know that we, as attendees, get less from them than we do from face-to-face live events. They're better than nothing, but communities like NoCOUG have the ability to provide the face-to-face live events that are so effective.

One of the difficulties users groups face is fatigue. It is difficult to organize events month after month, quarter after quarter, year after year. There is a great deal of satisfaction in running such an organization, especially one with the long and rich history enjoyed by NoCOUG. But it is exhausting. Current volunteers have overriding work and life conflicts. New volunteers are slow to come forward.

One thing to consider is reaching out to the larger national and international Oracle users groups, such as ODTUG, IOUG, OAUG, Quest, and OHUG. These groups have similar missions and most have outreach programs. ODTUG and IOUG in particular organize live onsite events in some cities, and have webinar programs as well. They have content, and NoCOUG has the membership and audience. NoCOUG members should encourage the board to contact these larger Oracle users groups for opportunities to partner locally.

Another growing trend is meet-ups, specifically through Meetup.com. This is a resource that has been embraced by all manner of tech-savvy people, from all points on the spectrum of the IT industry. I strongly urge all NoCOUG members to join Meetup.com, indicate your interests, and watch the flow of announcements visit your inbox. The meet-ups run the gamut from Hadoop to Android to Oracle Exadata to In-Memory to Big Data to Raspberry Pi to vintage Commodore. I think the future of local technical education lies in the intersection of online organization of local face-to-face interaction facilitated by Meetup.com.

Four conferences per year puts a huge burden on volunteers. There have been suggestions from multiple quarters that we organize just one big conference a year like some other user groups. That would involve changing our model from an annual membership fee of less than $100 for four single-day conferences (quarterly) to more than $300 for a single multiple-day conference (annual), but change is scary and success is not guaranteed. What are your thoughts on the quarterly vs. annual models?

I disagree with the idea that changing the conference format requires increasing annual dues. For example, RMOUG in Colorado (**http://rmoug.org/**) has one large annual conference with three smaller quarterly meetings, and annual dues are $75 and have been so for years. RMOUG uses the annual dues to pay for the three smaller quarterly education workshops (a.k.a. quarterly meetings) and the quarterly newsletter; the single large annual "Training Days" conference pays for itself with its own separate registration fees, which of course are discounted for members.

Think of a large annual event as a self-sufficient, self-sustaining organization within the organization, open to the public with a discount for dues-paying members.

Other Oracle users groups, such as UTOUG in Utah (**http://utoug.org/**), hold two large conferences annually (in March and November), and this is another way to distribute scarce volunteer resources. This offers a chance for experimentation as well, by hiring one conference-coordinator company to handle one event and another to handle the other, so that not all eggs are in one basket.

The primary goal of larger conferences is ongoing technical education of course, but a secondary goal is to raise funds for the continued existence of the users group and to help subsidize and augment the website, the smaller events, and the newsletter, if necessary.

It costs a fortune to produce and print the* NoCOUG Journal*, but we take a lot of pride in our unbroken 28-year history, in our tradition of original content, and in being one of the last printed publications by Oracle user groups. Needless to say it also takes a lot of effort. But is there enough value to show for the effort and the cost? We've been called a dinosaur. Should we follow the other dinosaurs into oblivion?

I don't think so. There are all kinds of formats for publication, from tweets to LinkedIn posts to blogs to magazines to books. Magazines like the *NoCOUG Journal* are an important piece of the educational ecosystem. I don't think that any of the Oracle users groups who no longer produce newsletters planned to end up this way. They ceased publishing because the organization could no longer sustain them.

I think today the hurdle is that newsletters can no longer be confined within the users group. Both NoCOUG and RMOUG have independently come to the realization that the newsletter must be searchable and findable online by the world, which provides the incentive for authors to submit content. Today, if it cannot be verified online, it isn't real. If it isn't real, then there is little incentive for authors to publish.

So making the *NoCOUG Journal* available online has been key to its own viability, and NoCOUG membership entitles one to a real hard-copy issue, which is a rare and precious bonus in this day and age.

Oracle Database 12c

Mogens Norgaard (the co-founder of the Oak Table Network) claims that* "since Oracle 7.3, that fantastic database has had pretty much everything normal customers need," *but the rest

"You are old," said the youth, "And your jaws are too weak
For anything tougher than suet;
Yet you finished the goose, with the bones and the beak—
Pray, how did you manage to do it?"

"In my youth," said his father, "I took to the law,
And argued each case with my wife;
And the muscular strength which it gave to my jaw,
Has lasted the rest of my life."

of us are not confirmed Luddites. What are the must-have features of Oracle 12c that give customers the incentive to upgrade from 11g to 12c? We've heard about pluggable databases and the in-memory option, but they are extra-cost options aren't they?

I know for a fact that the Automatic Data Optimization (ADO) feature obsolesces about 3,000 lines of complex PL/SQL code that I had written for Oracle 8*i*, 9*i*, 10*g*, and 11*g* databases. The killer feature within ADO is the ability to move partitions online, without interrupting query operations. Prior to Oracle 12*c*, accomplishing that alone consumed hundreds of hours of code development, testing, debugging, and release management. Combining ADO with existing features like OLTP compression and HCC compression truly makes transparent "tiers" of storage within an Oracle database feasible and practical. The ADO feature alone is worth the effort of upgrading to Oracle 12*c* for an organization with longer data retention requirements for historical analytics or regulatory compliance.

What's not to love about pluggable databases? How different is the pluggable database architecture from the architecture of SQL Server, DB2, and MySQL?

I think that first, in trying to explain Oracle pluggable databases, most people make it seem more confusing than it should be.

Stop thinking of an Oracle database as consisting of software, a set of processes, and a set of database files.

Instead, think of a database server as consisting of an operating system (OS) and an Oracle 12*c* container database software; a set of Oracle processes; and the basic control files, log files, and a minimal set of data files. When "gold images" of Oracle database servers are created, whether for jumpstart servers or for virtual machines, the Oracle 12*c* CDB should be considered part of that base operating system image.

Pluggable databases (PDBs) then are the data files installed along with the application software they support. PDBs are just tablespaces that plug into the working processes and infrastructure of the CDBs.

When PDBs are plugged in, all operational activities involving data protection—such as backups or redundancy like Data Guard replication—are performed at the higher CDB level.

Thus, all operational concerns are handled at the CDBs and the operational infrastructure from the PDBs and the applications.

Once the discussion is shifted at that high level, then the similarities are more visible between the Oracle 12*c* database and other multitenant databases, such as SQL Server and MySQL. Of course there will always be syntactic and formatting differences, but functionally Oracle 12*c* has been heavily influenced by its predecessors, such as SQL Server and MySQL.

"You are old," said the youth, "one would hardly suppose
That your eye was as steady as ever;
Yet you balanced an eel on the end of your nose—
What made you so awfully clever?"

"I have answered three questions, and that is enough,"
Said his father; "don't give yourself airs!
Do you think I can listen all day to such stuff?
Be off, or I'll kick you down stairs!"

Bonus Question

Do you have any career advice for the younger people reading this interview so that they can be like you some day? Other than actively participating in user groups!

This sounds corny and trite, but there is no such thing as a useless experience, and while it may be frustrating, it presents the opportunity to build. Understand that everyone starts at the bottom, and enjoy the climb.

Understand that learning causes stress. Stress is stress and too much can be unhealthy, but if it is a result of learning something new, then recognize it for what it is, know it is temporary and transitory, tough it out, and enjoy knowing the outcome when it arrives.

Also, don't voice a complaint unless you are prepared to present at least one viable solution, if not several. Understand what makes each solution truly viable and what makes it infeasible. If you can't present a solution to go with the complaint, then more introspection is needed. The term "introspection" is used deliberately, as it implies looking within rather than around.

Help people. Make an impact. Can we go wrong in pursuing either of those as goals? Sometimes I wish I had done more along these lines. Never do I wish I had done less. ▲

*Tim Gorman is a technical consultant for Delphix (**http://www.Delphix.com**), who enable database and storage virtualization to increase the agility of IT development and testing operations. He has co-authored six books, tech-reviewed eight more, and written articles for RMOUG SQL_Update and IOUG SELECT magazines. He has been an Oracle ACE since 2007, an Oracle ACE Director since 2012, a member of the Oak Table Network since 2002, and has presented at Oracle OpenWorld, Collaborate, KScope, Hotsos, and local Oracle users groups in a lot of wonderful places around the world. Tim lives in Westminster, Colo., with his partner, Kellyn Pot'Vin, and their children.*

NoCOUG Conference #127

Post-Conference Reception Hosted by Quest Shareplex

NoCOUG Conference #127

Post-Conference Reception Hosted by Quest Shareplex

vexata

Supercharge Oracle Performance with Vexata NVMe Arrays

- Accelerates OLTP & Analytic workloads
- Deploys with FC SANs or Gigabit Ethernet Fabrics
- Unmatched low-latency IOPS and throughput

DATABASE MANAGEMENT SOLUTIONS

Develop | Manage | Optimize | Monitor | Replicate

Maximize your Database Investments.

Quest™

NoCOUG
P.O. Box 3282
Danville, CA 94526

RETURN SERVICE REQUESTED

LIVE VIRTUAL CLASSES

TAUGHT BY
CRAIG SHALLAHAMER

- Oracle Tuning Fastpath
- Oracle Buffer Cache Performance Analysis And Tuning
- Tuning Oracle Using An AWR Report
- Tuning Oracle Using Advanced ASH Strategies

Use coupon code NOCOUG10 for 10% off!
Questions? Contact support@orapub.com.

ORAPUB

REGISTER TODAY

Find out more at orapub.com.