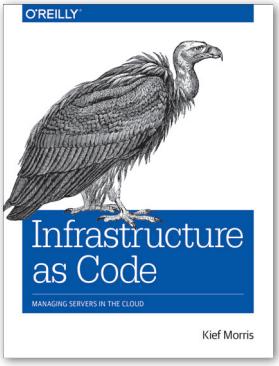
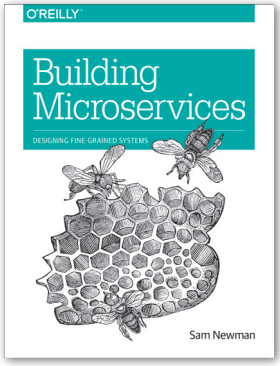
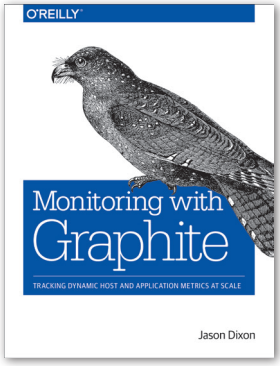
O'REILLY®

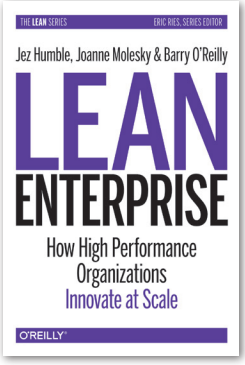
Modern Web Operations

A Curated Collection of Chapters from the O'Reilly Operations Library









4 Easy Ways to Stay Ahead of the Game

The world of web ops and performance is constantly changing. Here's how you can keep up:

1. **Download free reports** on the current and trending state of web operations, dev ops, business, mobile, and web performance. http://oreil.ly/free_resources
2. **Watch free videos and webcasts** from some of the best minds in the field—watch what you like, when you like, where you like. http://oreil.ly/free_resources
3. **Subscribe** to the weekly O'Reilly Web Ops and Performance newsletter. http://oreil.ly/getnews
4. **Attend the O'Reilly Velocity Conference**, the must-attend gathering for web operations and performance professionals, with events in California, New York, Europe, and China. http://velocityconf.com

For more information and additional Web Ops and Performance resources, visit **http://oreil.ly/Web_Ops**.

O'REILLY®

Modern Web Operations

A Curated Collection of Chapters from the O'Reilly Web Operations Library

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'REILLY®

Modern Web Operations
by O'Reilly Media, Inc.

Printed in the United States of America.

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

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

May 2015: First Edition

Revision History for the First Edition
2015-05-01 First Release

978-1-491-92807-3

[LSI]

Table of Contents

A Curated Collection of Chapters from the O'Reilly Web Operations Library v

Challenges and Principles 1
What Is Infrastructure as Code? 1
Values 3
Challenges with Dynamic Infrastructure 3
Habits that Lead to These Problems 8
Principles of Infrastructure as Code 10
When the Infrastructure Is Finished 18
What Good Looks Like 20

Deployment 23
A Brief Introduction to Continuous Integration 23
Mapping Continuous Integration to Microservices 25
Build Pipelines and Continuous Delivery 28
Platform-Specific Artifacts 30
Operating System Artifacts 31
Custom Images 32
Environments 36
Service Configuration 37
Service-to-Host Mapping 38
Automation 45
From Physical to Virtual 47
A Deployment Interface 52

Summary 56

Monitoring Conventions. 59
Three Tenets of Monitoring 60
Rethinking the Poll/Pull Model 63
Where Does Graphite Fit into the Picture? 66
Composable Monitoring Systems 67
Conclusion 77

Deploy Continuous Improvement. 79
The HP LaserJet Firmware Case Study 81
Drive Down Costs Through Continuous Process Innovation Using the Improvement Kata 84
How the HP LaserJet Team Implemented the Improvement Kata 90
Managing Demand 98
Creating an Agile Enterprise 100
Conclusion 102

IT as Conversational Medium. 105
Agile 107
DevOps 111
Cloud Computing 116
Design Thinking 119
Unifying Design and Operations 121
From Design Thinking to DevOps and Back Again 127

A Curated Collection of Chapters from the O'Reilly Web Operations Library

Learning the latest methodologies, tools, and techniques is critical for web operations, whether you're involved in systems administration, configuration management, systems monitoring, performance optimization, or consider yourself equal parts "Dev" and "Ops."

The O'Reilly Web Operations Library provides experienced Ops professionals with the knowledge and guidance you need to build your skillset and stay current with the latest trends.

This free ebook gets you started. With a collection of chapters from the library's published and forthcoming books, you'll learn about the scope and challenges that await you in the world of web operations, as well as the methods and mindset you need to adopt.

This ebook includes excerpts from the following books:

Infrastructure as Code
: Available in Early Release, Chapter 1. Challenges and Principles

Building Microservices
: Available now, Chapter 6. Deployment

Monitoring with Graphite
: Available in Early Release, Chapter 2. Monitoring Conventions

Lean Enterprise
: Available now, Chapter 6. Deploy Continuous Improvement

Designing Delivery
: Available in Early Release, Chapter 3. IT as Conversational Medium

For more information on current and forthcoming Web Operations content, check out http://www.oreilly.com/webops-perf/.

—Courtney Nash, Strategic Content Lead, courtney@oreilly.com

O'REILLY®

Infrastructure as Code

MANAGING SERVERS IN THE CLOUD

Kief Morris

Challenges and Principles

The following content is excerpted from *Infrastructure as Code*, by Kief Morris. Available now in Early Release.

Virtualization, cloud infrastructure, and configuration automation tools have swept into mainstream IT over the past decade. The promise of these tools is that they will automatically do the routine work of running an infrastructure, without involving the humans on the infrastructure team. Systems will all be up to date and consistent by default. Team members can spend their time and attention on high level work that makes significant improvements to the services they support. They can quickly, easily, and confidently adapt their infrastructure to meet the changing needs of their organization.

However, most IT infrastructure teams don't manage to get to this state, no matter which tools they adopt. They may be able to easily add servers and resources to their infrastructure, but still spend their time and attention on routine tasks like setting up and updating servers. They struggle to keep all of their servers consistently configured and up to date with system patches. They don't have enough time to spend on the more important projects and initiatives that they know will really make a difference to their organization.

What Is Infrastructure as Code?

Infrastructure as code is an approach to using newer technologies to build and manage dynamic infrastructure. It treats the infrastructure, and the tools and services that manage the infrastructure itself,

as a software system, adapting software engineering practices to manage changes to the system in a structured, safe way. This results in infrastructure with well tested functionality to manage routine operational tasks, and a team that has clear, reliable processes for making changes to the system.

Dynamic Infrastructure

The components of a dynamic infrastructure[1] change continuously and automatically. Servers may appear or disappear so that capacity is matched with load, to recover from failures, or to enable services. These changes could be triggered by humans, for example a tester needing an environment to test a new software build. Or the system itself may apply changes in response to events such as a hardware failure. But the process of creating, configuring, and destroying the infrastructure elements happens automatically.

The foundation of infrastructure as code is the ability to treat infrastructure elements as if they were data. Virtualization and cloud hosting platforms decouple infrastructure from its underlying hardware, providing a programmatic interface for managing servers, storage, and network devices. LOM (Lights Out Management) and other tooling also make it possible to manage operating systems installed directly on hardware in a similar way, which makes it possible to use infrastructure as code in non-virtualized environments as well. Automated infrastructure management platforms are discussed in more detail in Chapter 2.

Automated configuration management tools like Ansible, Cfengine, Chef, Puppet, and Salt (among others) allow infrastructure elements themselves to be configured using specialized programming languages. IT operations teams can manage configuration definitions using using tools and practices which have been proven effective for software development, including Version Control Systems (VCS), Continuous Integration (CI), Test Driven Development (TDD), and Continuous Delivery (CD). Configuration management tools are discussed in Chapter 3, and software engineering practices are covered throughout this book.

1 Some further reading on dynamic infrastructures include: Web Operations, by Allspaw, Robbins, et. al.; and The Practice of Cloud System Administration, by Limoncelli, Chalup, and Hogan.

So an automated infrastructure management platform and server configuration tools are a starting point, but they aren't enough. The practices and approaches of the Iron Age of Infrastructure - the days when infrastructure was bound to the hardware it ran on - don't cope well with dynamic infrastructure, as we'll see in the next section. Infrastructure as code is a new way of approaching dynamic infrastructure management.

Values

These are some of the values that lead to infrastructure as code:

- IT infrastructure should support and enable change, not be an obstacle or a constraint.
- IT staff should spend their time on valuable things which engage their abilities, not on routine, repetitive tasks.
- Users should be able to provision and manage the resources they need, without needing IT staff to do it for them.
- Teams should know how to recover quickly from failure, rather than depending on avoiding failure.
- Changes to the system should be routine, without drama or stress for users or IT staff.
- Improvements should be made continuously, rather than done through expensive and risky "big bang" projects.
- Solutions to problems should be proven through implementing, testing, and measuring them, rather than by discussing them in meetings and documents.

Challenges with Dynamic Infrastructure

In this section, we'll look 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

As I mentioned in the preface, when my team first adopted virtualization our infrastructure exploded from around twenty servers to well over one hundred in under a year, leading to a "Sorcerer's Apprentice" situation where we couldn't keep them all well managed.

The ability to create new servers with the snap of a finger is satisfying – responding quickly to the needs of users can only be a good thing. The trap is that the number of servers can grow out of control. The more servers you create, the harder it is to keep them updated, optimized, and running smoothly. Server sprawl leads to configuration drift.

Configuration Drift

Servers may be consistent when they're created, but over time differences creep in.

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, and now the Jira server is different from other servers that have Java installed. Three different people install IIS on three different web servers over 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 it's configuration is different from the other JBoss servers.

Being different isn't bad. The heavily loaded JBoss server probably should be tuned differently from the ones with low 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

Years ago the company I was with built web applications for clients, most of which were monstrous collections of Perl CGI. (Don't judge us, this was the dot-com days, 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, which 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 not on our staging environment. Don't ask why we upgraded production to 5.8 without discovering the problem with our test environment, but that's how we ended up. We had a 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[2], 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, using FAI for PXE Boot installations, Cfengine to configure servers, and everything checked into CVS.

As embarrassing as this story is, most IT Ops teams have similar stories of special stories that can'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 IT Ops 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.

2 Sadly, the infrastructures.org site hadn't been updated since 2007 when I last looked at it.

Jenga Infrastructure

A Jenga infrastructure is easily disrupted and not easily fixed. This is the snowflake server, scaled up.

The solution is to migrate everything in the infrastructure to a reliable, reproducible infrastructure, one step at a time. The Visible Ops Handbook[3] outlines an approach for bringing stability and predictability to a difficult infrastructure.

> 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 re-plugged, and nobody touched the server again.

Automation Fear

At an open spaces session on configuration automation at a DevOpsDays conference, I asked the group how many of them were using an 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.

3 The Visible Ops Handbook by Gene Kim, George Spafford, and Kevin Behr. The book was originally 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.

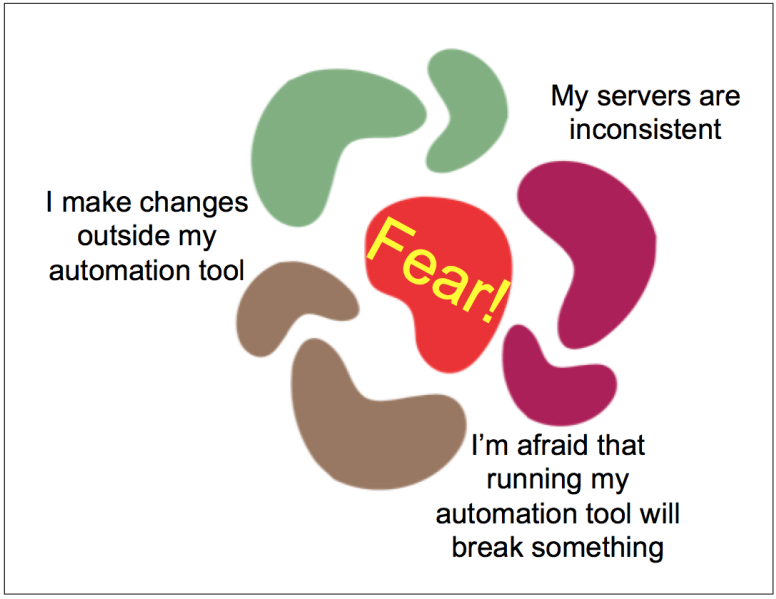


Figure 1-1. The automation fear spiral

This is the automation fear spiral, 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.

Good monitoring, and effective automated testing regimes as described in Part III of this book will help build confidence that configuration can be reliably applied and problems caught quickly.

Erosion

In an ideal world we would never need to touch an automated infrastructure once we'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, our infrastructure will decay over time. The folks at Heroku call this erosion. 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.

Habits that Lead to These Problems

In this section, we'll look at some of the habits that teams fall into that can lead to problems like snowflake servers and configuration drift. These habits tend to reinforce each other, each habit creating conditions which encourage the others.

Doing Routine Tasks Manually

When people have to spend their time doing routine things like setting up servers, applying updates, and making configuration changes across a group of servers, not only does it take their time and attention away from more important work, it also leads to things being done inconsistently. This in turn leads to configuration drift, snowflakes, and the various other evils we've discussed.

Running Scripts Manually

Most infrastructure teams write scripts for at least some of their routine work, and quite a few have adopted automated configuration tools to help with this. But many teams run their scripts by hand, rather than having them run unattended on a schedule, or automatically triggered by events. A human may need to pass parameters to the script, or just keep a close eye on it and check things afterwards to make sure everything has worked correctly. Using a script may help keep things consistent, but still takes time and attention away from more useful work.

Needing humans to babysit scripts suggests that there are not enough controls - tests and monitoring - to make the script properly safe to run. This topic is touched on throughout this book, but espe-

cially in Part III. Team members also need to learn habits for writing code that is safe to run unattended, such as error detection and handling.

Applying Changes Directly to Important Systems

Most mature IT organizations would never allow software developers to make code changes directly to production systems, without testing the changes in a development environment first. But this standard doesn't always apply to infrastructure. I've been to organizations which put their software releases through half a dozen stages of testing and staging, and seen IT operations staff routinely make changes to server configurations directly on production systems.

I've even seen teams do this with configuration management tools. They make changes to Puppet manifests and apply them to business critical systems without any testing at all. Editing production server configuration by hand is like playing Russian Roulette with your business. Running untested configuration scripts against a groups of servers is like playing Russian Roulette with a machine gun.

Running Automation Infrequently

Many teams who start using an automated configuration tool only run it when they need to make a specific change. Sometimes they only run the tool against the specific servers they want to make the change on.

The problem with this is that the longer you wait to run the configuration tool, the more difficult and risky it is to run the tool. Someone may have made a change to the server - or some of the servers - since the last time it was run, and that change may be incompatible with the definitions you are applying now. Or someone else may have changed the configuration definitions, but not applied them to the server you're changing now, which again means more things that can go wrong. If your team is in the habit of making changes to scripts for use with specific servers, there's an even greater chance of incompatibilities.

This of course leads to automation fear, configuration drift, and snowflakes.

Bypassing Automation

When scripts and configuration tools are run infrequently and inconsistently, and there is little confidence in the automation tools, then it's common to make changes outside of the tools. These are done as a supposed one-off to deal with an urgent need, with the hope that you'll go back and update the automation scripts later on.

Of course, these out-of-band changes will cause later automation runs to fail, or to revert important fixes. The more this happens, the more unreliable and unusable our configuration scripts become.

Principles of Infrastructure as Code

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

Principle: Reproducibility

It should be possible to rebuild any element of an infrastructure quickly, easily, and effortlessly. 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 enables many powerful capabilities of infrastructure as code. It cuts down on much of the fear and risk of infrastructure operations. Servers are trivial to replace, so they can be treated as disposable. Reproducibility supports continuous operations (as discussed in Chapter 14), automated scaling, and creating new environments and services on demand.

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

Principle: Consistency

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 exactly the same, except for those bits of configuration that differentiate them from one another, 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's 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, such as finding that 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 repeatably 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.

Principle: Repeatability

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 than 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 file system 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 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 our ability to automate things.

Effective infrastructure teams have a strong scripting culture. If a task can be scripted, script it. If as 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.

Principle: Disposability

> "Treat your servers like cattle, not pets."[4]
>
> —Bill Baker

We should assume that any infrastructure element can and will be destroyed at any time, without notice. It could be destroyed unexpectedly, when hardware fails, or it could be deliberately destroyed in order to reduce capacity or to replace it. If we design our services so that infrastructure can disappear without drama, then we can freely destroy and replace elements whenever we need, to upgrade, to reconfigure resources, to reduce capacity when demand is low, or for any other reason.

This idea that we can't expect a particular server to be around tomorrow, or even in a few minutes time, is a fundamental shift. Logging into a server and making changes manually is pointless except for debugging or testing potential changes. This requires that any change that matters be made through the automated systems that create and configure servers.

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.

4 The cattle/pets analogy was been attributed to former Microsoft employee Bill Baker, according to CloudConnect CTO Randy Bias in his presentation Architectures for open and scalable clouds. I first heard the analogy in Gavin McCance's presentation CERN Data Centre Evolution. Both of these presentations are excellent.

> 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 jenga 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.

Principle: Service Continuity

Given a service hosted on our infrastructure, that service must be continuously available to its users even when individual infrastructure elements disappear.

To run a continuously available service on top of disposable infrastructure components, we need to identify which things are absolutely required for the service, and make sure they are decoupled from the infrastructure. This tends to come down to request handling and data management.

We need to make sure our service is always able to handle requests, in spite of what might be happening to the infrastructure. If a server disappears, we need to have other servers already running, and be able to quickly start up new ones, so that service is not interrupted. This 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 data management chapter (Chapter 13) has ideas for this.

State and transactions are an especially tricky consideration. Although there are technologies and tools to manage distributed state and transactions, in practice the most reliable approach is to design software not to depend on this. The twelve-factor application methodology is a helpful approach. For larger, enterprise-scale systems, microservice architecture has proven effective for many organizations.

The chapter on continuity (Chapter 14) goes into techniques for continuous service and disaster recovery.

Principle: Self-Testing Systems

Effective automated testing is one of the most important practices infrastructure operations teams can adopt from software development. The highest performing software development teams I've worked with make automated testing a core part of their development process. 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.

The main benefit they see from this is fast feedback as to whether the changes they're making work correctly and without breaking other parts of the system. This immediate feedback gives the team confidence that errors will usually be caught quickly, which then gives them 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 "DevOps"). Good testing practices are the key to eliminating automation fear.

However while most software development teams aspire to use automated testing, in my experience the majority struggle to do it rigorously. There are two factors that make it difficult. The first is that it takes time to build up the skills, habits, and discipline that make test writing a routine, easy thing to do. Until these are built up, writing tests tends to make things feel slow, which discourages teams and drives them to drop test writing in order to get things done.

The second factor that makes effective automated testing difficult is that many teams don't integrate test writing into their daily working process, but instead write them after much of their code is already implemented. In fact, tests are often written by a separate QA team rather than by the developers themselves, which means the testing is not integrated with the development process, and not integrated with the software.

Writing tests separately from the code means the development team doesn't get that core benefit of testing - they don't get immediate feedback when they make a change that breaks something, and so they don't get the confidence to code and deliver quickly.

Chapter 10 explores practices and techniques for implementing testing as part of the system, and particularly how this can be done effectively for infrastructure.

Principle: Self-Documenting Systems

A common pattern with IT teams is the struggle to keep documentation relevant, useful, and accurate. When a new tool or system is implemented someone typically takes the time to create a beautiful, comprehensive document, with color glossy screenshots with circles and arrows and a paragraph for each step. But it's difficult to take the time to update documentation every time a script or tool is tweaked, or when a better way is discovered to carry out a task. Over time, the documentation becomes less accurate. It doesn't seem to matter what tool or system is used to manage the documentation - the pattern repeats itself with expensive document management systems, sophisticated collaboration tools, and simple, quick to edit wikis.

And of course, not everyone on the team follows the documented process the same way. People tend to find their own shortcuts and improvements, or write their own little scripts to make parts of the process easier. So although documenting a process is often seen as a way to enforce consistency, standards and even legal compliance, it's generally a fictionalized version of reality.

A great benefit of the infrastructure as code approach is that the steps to carry out a process are captured in the scripts and tooling that actually carry out that process. Documentation outside the scripts can be minimal, indicating the entry points to the tools, how to set them up (although this should ideally be a scripted process itself), and where to find the source code to learn how it works in detail.

Some people in an organization, especially those who aren't directly involved with the automation systems, may feel more comfortable having extensive documentation outside the tools themselves. A useful exercise is to consider the use cases for documentation, and then agree on how each use case will be addressed. For example, a common use case is a new technical team member joins and needs to learn the system. This can be addressed by fairly lightweight documentation and whiteboarding sessions to give an overview, and learning by reading and working with the automated scripts.

One use case that tends to need a bit more documentation is non-technical end users of a system, who won't read configuration scripts to learn how to use it. Ideally, the tooling for these users should have a clear, straightforward user experience, with the information needed to make decisions included in the interface. A rule of thumb for any documentation should be, if it becomes out of date, and work goes on without anyone noticing, then that document is probably unnecessary.

Principle: Small Changes

When I first got involved in developing IT systems, my instinct was to complete the whole chunk of work I was doing 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
- When something goes wrong with a big batch of changes, you often need to delay everything, including useful changes, while you fix the small broken thing
- 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.

Principle: Version All the Things

Versioning of infrastructure configuration is the cornerstone of infrastructure of code. It makes it possible to automate processes around making changes, including tests and auditing. It makes changes traceable, reproducible, and reversible.

Approaches to using source control are discussed in the chapter on software practices, and are referred to with many, if not most, of the topics in this book.

Selecting Tools

Too many organizations start with tools. It's appealing for managers to select a vendor that promises to make everything work well, and it's appealing to technical folks to find a shiny new tool to learn. But this leads to failed projects, often expensive ones.

I recommend going through this book and other resources, and deciding what infrastructure management strategy makes sense for you. Then choose the simplest, least expensive tools you can use to explore how to implement the strategy. As things progress and your team learns more, they will inevitably find that many of the tools they've chosen aren't the right fit, and new tools will emerge that may be better.

The best decision you can make on tools is for everyone to agree that tools will change over time.

This suggests that paying a large fee for an expensive tool, or set of tools, is unwise unless you've been using it successfully for a while. I've seen organizations which are struggling to keep up with technically-savvy competition sign ten-year, seven-figure licensing agreements with vendors. This makes as much sense as pouring concrete over your Formula One car at the start of a race.

Select each tool with the assumption that you will need to throw it out within a year. Hopefully this won't be true of every tool in your stack, but avoid having certain tools which are locked in, because you can guarantee those will be the tools that hold you back.

When the Infrastructure Is Finished

Reading through the principles outlined above may give the impression that automation should (and can) be used to create a perfect, factory-like machine. You'll build it, and then sit back, occasionally intervening to oil the gears and replace parts that break. Management can probably cut staffing down to a minimal number of people, maybe less technically skilled ones since all they need to do is watch some gauges and follow a checklist to fix routine problems.

Sadly, this is a fantasy. IT infrastructure automation isn't a standardized commodity yet, but is still at the stage of hand-crafting systems customized for each organization. Once in operation, the infrastructure will need to be continuously improved and adapted. Most organizations will find an ongoing need to support new services and change existing ones, which will need changes to the infrastructure underlying them. Even without changing service requirements, erosion (as described above in "Erosion" on page 7) creates a continuous stream of maintenance work.

When planning to build of an automated infrastructure, make sure the people who will run and maintain it are involved in its design and implementation. Running the infrastructure is really just a continuation of building it, so the team needs to know how it was built, and really needs to have made the decisions about how to build it, so they will be in the best position for continued ownership.

Please don't buy into the idea that an off the shelf, expensive product from even the most reputable vendor will get you around this. Any automation tool requires extensive knowledge to implement, and operating it requires intimate knowledge of its implementation.

Antifragility—Beyond "Robust"

We typically aim to build robust infrastructure, meaning systems will hold up well to shocks - failures, load spikes, attacks, etc. 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, 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 apply to IT architecture.

The key to an antifragile 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 prevent it from happening again.

A team typically handles an incident by first making a quick fix, so service can resume, then working out the changes needed to fix the underlying cause, to prevent the issue from happening again. Tweaking monitoring to alert when the issue happens again is often an afterthought, something nice to have but easily neglected.

A team striving for antifragility will make monitoring, and even automated testing, the second step, after the quick fix and before implementing the long term fix.

This may be counter-intuitive. Some systems administrators have told me it's a waste of time to implement automated checks for an issue that has already been fixed, since by definition it won't happen again. But in reality, fixes don't always work, may not resolve related issues, and can even be reversed by well-meaning team members who weren't involved in the previous incident.

Add a monitoring check to alert the team if the issue happens again. Implement automated tests that run when someone changes configuration related to the parts of the system that broke.

Implement these checks and tests before you implement the fix to the underlying problem, then reproduce the problem and prove that your checks really do catch it. Then implement the fix, re-run the tests and checks, and you will prove that your fix works. This is Test Driven Development (TDD) for infrastructure!

The Software Practices chapter and the Pipeline chapter go into more details on how to do this sort of thing routinely.

The secret ingredient of anti-fragile IT systems

It's people! People are the element that can cope with unexpected situations and adapt 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 we might be able to buy a standard cor-

porate 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 non-technology 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.

Brian L. Troutwin gave a talk at DevOpsDays Ghent in 2014 on Automation with humans in mind. 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.

What Good Looks Like

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 MTTR (Mean Time to Recover) and focuses on ways to improve this. Although MTBF (Mean Time Between

Failure) may also be tracked, the team does not rely on avoiding failures.[5]

- Team members feel their work is adding measurable value to the organization.

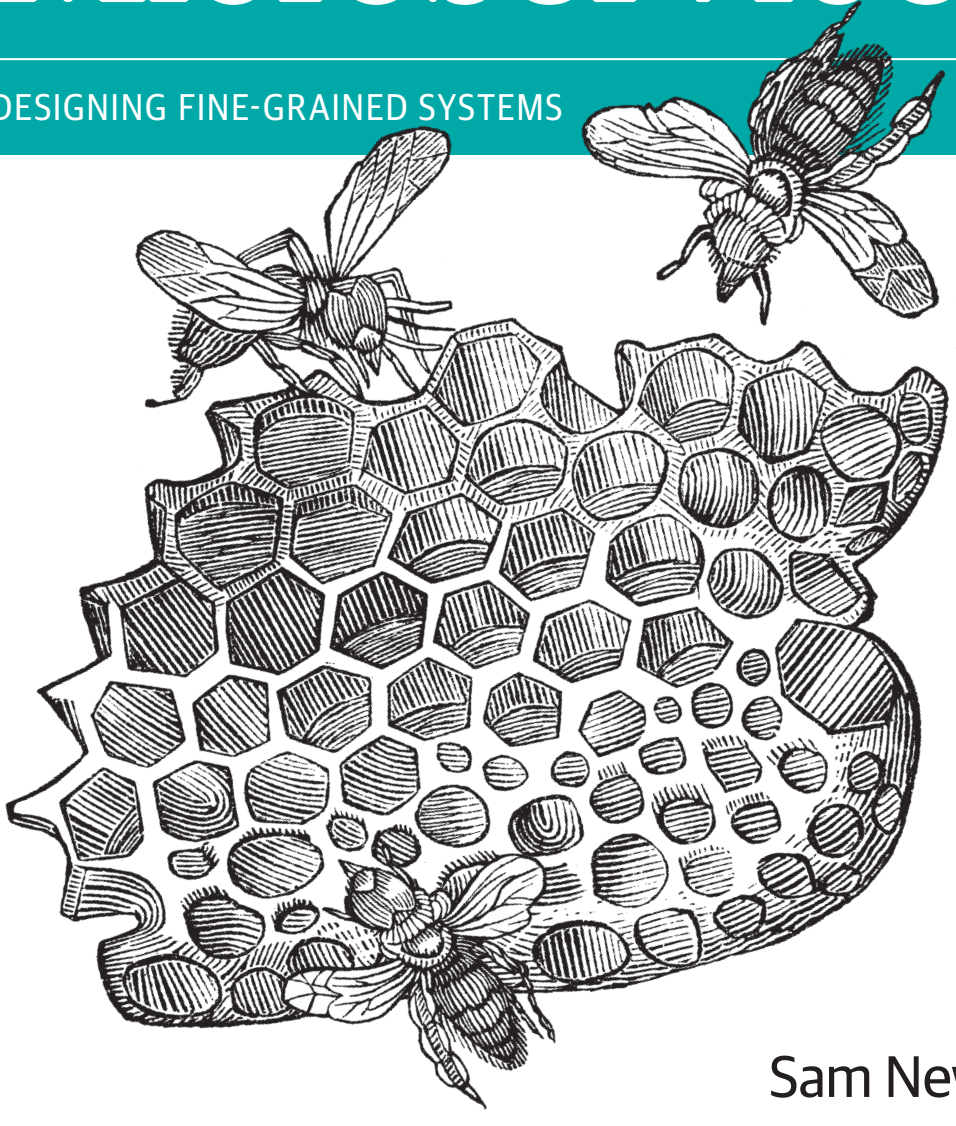
The next two chapters discuss the core toolchains for building infrastructure as code; the infrastructure management platform, and server configuration tools. Part II describes how to use these tools in a way that follows infrastructure as code approaches.

5 See John Allspaw's seminal blog post, MTTR is more important than MTBF (for most types of F).

O'REILLY®

Building Microservices

DESIGNING FINE-GRAINED SYSTEMS



Sam Newman

Deployment

The following content is excerpted from *Building Microservices*, by Sam Newman. Available now.

Deploying a monolithic application is a fairly straightforward process. Microservices, with their interdependence, are a different kettle of fish altogether. If you don't approach deployment right, it's one of those areas where the complexity can make your life a misery. In this chapter, we're going to look at some techniques and technology that can help us when deploying microservices into fine-grained architectures.

We're going to start off, though, by taking a look at continuous integration and continuous delivery. These related but different concepts will help shape the other decisions we'll make when thinking about what to build, how to build it, and how to deploy it.

A Brief Introduction to Continuous Integration

Continuous integration (CI) has been around for a number of years at this point. It's worth spending a bit of time going over the basics, however, as especially when we think about the mapping between microservices, builds, and version control repositories, there are some different options to consider.

With CI, the core goal is to keep everyone in sync with each other, which we achieve by making sure that newly checked-in code properly integrates with existing code. To do this, a CI server detects that

the code has been committed, checks it out, and carries out some verification like making sure the code compiles and that tests pass.

As part of this process, we often create artifact(s) that are used for further validation, such as deploying a running service to run tests against it. Ideally, we want to build these artifacts once and once only, and use them for all deployments of that version of the code. This is in order to avoid doing the same thing over and over again, and so that we can confirm that the artifact we deployed is the one we tested. To enable these artifacts to be reused, we place them in a repository of some sort, either provided by the CI tool itself or on a separate system.

We'll be looking at what sorts of artifacts we can use for microservices shortly, and we'll look in depth at testing in Chapter 7.

CI has a number of benefits. We get some level of fast feedback as to the quality of our code. It allows us to automate the creation of our binary artifacts. All the code required to build the artifact is itself version controlled, so we can re-create the artifact if needed. We also get some level of traceability from a deployed artifact back to the code, and depending on the capabilities of the CI tool itself, can see what tests were run on the code and artifact too. It's for these reasons that CI has been so successful.

Are You Really Doing It?

I suspect you are probably using continuous integration in your own organization. If not, you should start. It is a key practice that allows us to make changes quickly and easily, and without which the journey into microservices will be painful. That said, I have worked with many teams who, despite saying that they do CI, aren't actually doing it at all. They confuse the use of a CI tool with adopting the practice of CI. The tool is just something that enables the approach.

I really like Jez Humble's three questions he asks people to test if they really understand what CI is about:

Do you check in to mainline once per day?
: You need to make sure your code integrates. If you don't check your code together with everyone else's changes frequently, you end up making future integration harder. Even if you are using short-lived branches to manage changes, integrate as frequently as you can into a single mainline branch.

Do you have a suite of tests to validate your changes?
: Without tests, we just know that syntactically our integration has worked, but we don't know if we have broken the behavior of the system. CI without some verification that our code behaves as expected isn't CI.

When the build is broken, is it the #1 priority of the team to fix it?
: A passing green build means our changes have safely been integrated. A red build means the last change possibly did not integrate. You need to stop all further check-ins that aren't involved in fixing the builds to get it passing again. If you let more changes pile up, the time it takes to fix the build will increase drastically. I've worked with teams where the build has been broken for days, resulting in substantial efforts to eventually get a passing build.

Mapping Continuous Integration to Microservices

When thinking about microservices and continuous integration, we need to think about how our CI builds map to individual microservices. As I have said many times, we want to ensure that we can make a change to a single service and deploy it independently of the rest. With this in mind, how should we map individual microservices to CI builds and source code?

If we start with the simplest option, we could lump everything in together. We have a single, giant repository storing all our code, and have one single build, as we see in Figure 2-1. Any check-in to this source code repository will cause our build to trigger, where we will run all the verification steps associated with all our microservices, and produce multiple artifacts, all tied back to the same build.

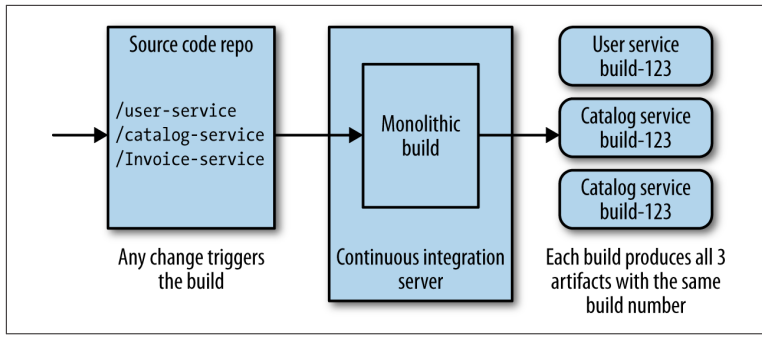


Figure 2-1. Using a single source code repository and CI build for all microservices

This seems much simpler on the surface than other approaches: fewer repositories to worry about, and a conceptually simpler build. From a developer point of view, things are pretty straightforward too. I just check code in. If I have to work on multiple services at once, I just have to worry about one commit.

This model can work perfectly well if you buy into the idea of lock-step releases, where you don't mind deploying multiple services at once. In general, this is absolutely a pattern to avoid, but very early on in a project, especially if only one team is working on everything, this might make sense for short periods of time.

However, there are some significant downsides. If I make a one-line change to a single service—for example, changing the behavior in the user service in Figure 2-1—*all* the other services get verified and built. This could take more time than needed—I'm waiting for things that probably don't need to be tested. This impacts our cycle time, the speed at which we can move a single change from development to live. More troubling, though, is knowing what artifacts should or shouldn't be deployed. Do I now need to deploy all the build services to push my small change into production? It can be hard to tell; trying to guess which services *really* changed just by reading the commit messages is difficult. Organizations using this approach often fall back to just deploying everything together, which we really want to avoid.

Furthermore, if my one-line change to the user service breaks the build, no other changes can be made to the other services until that break is fixed. And think about a scenario where you have multiple teams all sharing this giant build. Who is in charge?

A variation of this approach is to have one single source tree with all of the code in it, with multiple CI builds mapping to parts of this source tree, as we see in Figure 2-2. With well-defined structure, you can easily map the builds to certain parts of the source tree. In general, I am not a fan of this approach, as this model can be a mixed blessing. On the one hand, my check-in/check-out process can be simpler as I have only one repository to worry about. On the other hand, it becomes very easy to get into the habit of checking in source code for multiple services at once, which can make it equally easy to slip into making changes that couple services together. I would greatly prefer this approach, however, over having a single build for multiple services.

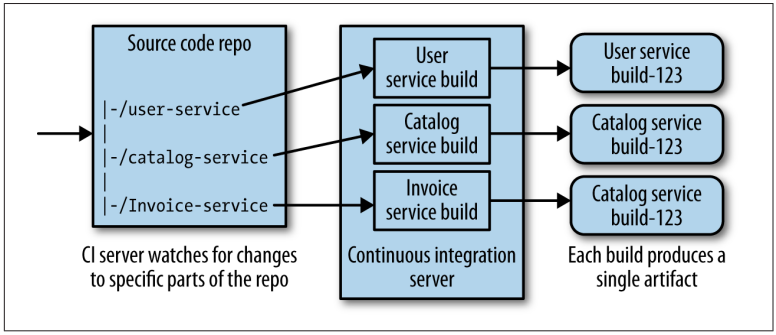


Figure 2-2. A single source repo with subdirectories mapped to independent builds

So is there another alternative? The approach I prefer is to have a single CI build per microservice, to allow us to quickly make and validate a change prior to deployment into production, as shown in Figure 2-3. Here each microservice has its own source code repository, mapped to its own CI build. When making a change, I run only the build and tests I need to. I get a single artifact to deploy. Alignment to team ownership is more clear too. If you own the service, you own the repository and the build. Making changes across repositories can be more difficult in this world, but I'd maintain this is easier to resolve (e.g., by using command-line scripts) than the downside of the monolithic source control and build process.

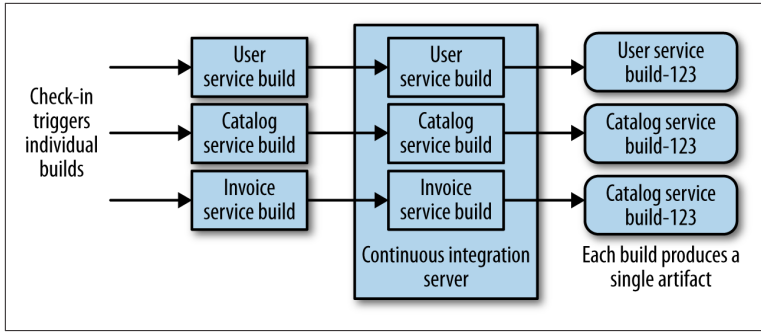


Figure 2-3. Using one source code repository and CI build per microservice

The tests for a given microservice should live in source control with the microservice's source code too, to ensure we always know what tests should be run against a given service.

So, each microservice will live in its own source code repository, and its own CI build process. We'll use the CI build process to create our deployable artifacts too in a fully automated fashion. Now lets look beyond CI to see how continuous delivery fits in.

Build Pipelines and Continuous Delivery

Very early on in using continuous integration, we realized the value in sometimes having multiple stages inside a build. Tests are a very common case where this comes into play. I may have a lot of fast, small-scoped tests, and a small number of large-scoped, slow tests. If we run all the tests together, we may not be able to get fast feedback when our fast tests fail if we're waiting for our long-scoped slow tests to finally finish. And if the fast tests fail, there probably isn't much sense in running the slower tests anyway! A solution to this problem is to have different stages in our build, creating what is known as a *build pipeline*. One stage for the faster tests, one for the slower tests.

This build pipeline concept gives us a nice way of tracking the progress of our software as it clears each stage, helping give us insight into the quality of our software. We build our artifact, and that artifact is used throughout the pipeline. As our artifact moves through these stages, we feel more and more confident that the software will work in production.

Continuous delivery (CD) builds on this concept, and then some. As outlined in Jez Humble and Dave Farley's book of the same name, continuous delivery is the approach whereby we get constant feedback on the production readiness of each and every check-in, and furthermore treat each and every check-in as a release candidate.

To fully embrace this concept, we need to model all the processes involved in getting our software from check-in to production, and know where any given version of the software is in terms of being cleared for release. In CD, we do this by extending the idea of the multistage build pipeline to model each and every stage our software has to go through, both manual and automated. In Figure 2-4, we see a sample pipeline that may be familiar.

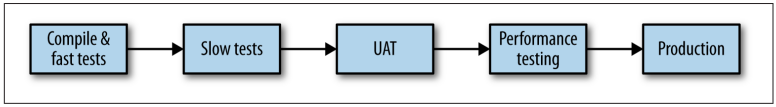


Figure 2-4. A standard release process modeled as a build pipeline

Here we really want a tool that embraces CD as a first-class concept. I have seen many people try to hack and extend CI tools to make them do CD, often resulting in complex systems that are nowhere as easy to use as tools that build in CD from the beginning. Tools that fully support CD allow you to define and visualize these pipelines, modeling the entire path to production for your software. As a version of our code moves through the pipeline, if it passes one of these automated verification steps it moves to the next stage. Other stages may be manual. For example, if we have a manual user acceptance testing (UAT) process I should be able to use a CD tool to model it. I can see the next available build ready to be deployed into our UAT environment, deploy it, and if it passes our manual checks, mark that stage as being successful so it can move to the next.

By modeling the entire path to production for our software, we greatly improve visibility of the quality of our software, and can also greatly reduce the time taken between releases, as we have one place to observe our build and release process, and an obvious focal point for introducing improvements.

In a microservices world, where we want to ensure we can release our services independently of each other, it follows that as with CI, we'll want one pipeline per service. In our pipelines, it is an artifact that we want to create and move through our path to production. As

always, it turns out our artifacts can come in lots of sizes and shapes. We'll look at some of the most common options available to us in a moment.

And the Inevitable Exceptions

As with all good rules, there are exceptions we need to consider too. The "one microservice per build" approach is absolutely something you should aim for, but are there times when something else makes sense? When a team is starting out with a new project, especially a greenfield one where they are working with a blank sheet of paper, it is quite likely that there will be a large amount of churn in terms of working out where the service boundaries lie. This is a good reason, in fact, for keeping your initial services on the larger side until your understanding of the domain stabilizes.

During this time of churn, changes across service boundaries are more likely, and what is in or not in a given service is likely to change frequently. During this period, having all services in a single build to reduce the cost of cross-service changes may make sense.

It does follow, though, that in this case you need to buy into releasing all the services as a bundle. It also absolutely needs to be a transitionary step. As service APIs stabilize, start moving them out into their own builds. If after a few weeks (or a very small number of months) you are unable to get stability in service boundaries in order to properly separate them, merge them back into a more monolithic service (albeit retaining modular separation within the boundary) and give yourself time to get to grips with the domain. This reflects the experiences of our own SnapCI team, as we discussed in Chapter 3.

Platform-Specific Artifacts

Most technology stacks have some sort of first-class artifact, along with tools to support creating and installing them. Ruby has gems, Java has JAR files and WAR files, and Python has eggs. Developers with experience in one of these stacks will be well versed in working with (and hopefully creating) these artifacts.

From the point of view of a microservice, though, depending on your technology stack, this artifact may not be enough by itself. While a Java JAR file can be made to be executable and run an

embedded HTTP process, for things like Ruby and Python applications, you'll expect to use a process manager running inside Apache or Nginx. So we may need some way of installing and configuring other software that we need in order to deploy and launch our artifacts. This is where automated configuration management tools like Puppet and Chef can help.

Another downfall here is that these artifacts are specific to a certain technology stack, which may make deployment more difficult when we have a mix of technologies in play. Think of it from the point of view of someone trying to deploy multiple services together. They could be a developer or tester wanting to test some functionality, or it could be someone managing a production deployment. Now imagine that those services use three completely different deployment mechanisms. Perhaps we have a Ruby Gem, a JAR file, and a nodeJS NPM package. Would they thank you?

Automation can go a long way toward hiding the differences in the deployment mechanisms of the underlying artifacts. Chef, Puppet, and Ansible all support multiple different common technology-specific build artifacts too. But there are different types of artifacts that might be even easier to work with.

Operating System Artifacts

One way to avoid the problems associated with technology-specific artifacts is to create artifacts that are native to the underlying operating system. For example, for a RedHat– or CentOS-based system, I might build RPMs; for Ubuntu, I might build a deb package; or for Windows, an MSI.

The advantage of using OS-specific artifacts is that from a deployment point of view we don't care what the underlying technology is. We just use the tools native to the OS to install the package. The OS tools can also help us uninstall and get information about the packages too, and may even provide package repositories that our CI tools can push to. Much of the work done by the OS package manager can also offset work that you might otherwise do in a tool like Puppet or Chef. On all Linux platforms I have used, for example, you can define dependencies from your packages to other packages you rely on, and the OS tools will automatically install them for you too.

The downside can be the difficulty in creating the packages in the first place. For Linux, the FPM package manager tool gives a nicer abstraction for creating Linux OS packages, and converting from a tarball-based deployment to an OS-based deployment can be fairly straightforward. The Windows space is somewhat trickier. The native packaging system in the form of MSI installers and the like leave a lot to be desired when compared to the capabilities in the Linux space. The NuGet package system has started to help address this, at least in terms of helping manage development libraries. More recently, Chocolatey NuGet has extended these ideas, providing a package manager for Windows designed for deploying tools and services, which is much more like the package managers in the Linux space. This is certainly a step in the right direction, although the fact that the idiomatic style in Windows is still *deploy something in IIS* means that this approach may be unappealing for some Windows teams.

Another downside, of course, could be if you are deploying onto multiple different operating systems. The overhead of managing artifacts for different OSes could be pretty steep. If you're creating software for other people to install, you may not have a choice. If you are installing software onto machines you control, however, I would suggest you look at unifying or at least reducing the number of different operating systems you use. It can greatly reduce variations in behavior from one machine to the next, and simplify deployment and maintenance tasks.

In general, those teams I've seen that have moved to OS-based package management have simplified their deployment approach, and tend to avoid the trap of big, complex deployment scripts. Especially if you're on Linux, this can be a good way to simplify deployment of microservices using disparate technology stacks.

Custom Images

One of the challenges with automated configuration management systems like Puppet, Chef, and Ansible can be the time taken to run the scripts on a machine. Let's take a simple example of a server being provisioned and configured to allow for the deployment of a Java application. Let's assume I'm using AWS to provision the server, using the standard Ubuntu image. The first thing I need to do is install the Oracle JVM to run my Java application. I've seen this sim-

ple process take around five minutes, with a couple of minutes taken up by the machine being provisioned, and a few more to install the JVM. Then we can think about actually putting our software on it.

This is actually a fairly trivial example. We will often want to install other common bits of software. For example, we might want to use collectd for gathering OS stats, use logstash for log aggregation, and perhaps install the appropriate bits of nagios for monitoring (we'll talk more about this software in Chapter 8). Over time, more things might get added, leading to longer and longer amounts of time needed for provisioning of these dependencies.

Puppet, Chef, Ansible, and their ilk can be smart and will avoid installing software that is already present. This does not mean that running the scripts on existing machines will always be fast, unfortunately, as running all the checks takes time. We also want to avoid keeping our machines around for too long, as we don't want to allow for too much configuration drift (which we'll explore in more depth shortly). And if we're using an on-demand compute platform we might be constantly shutting down and spinning up new instances on a daily basis (if not more frequently), so the declarative nature of these configuration management tools may be of limited use.

Over time, watching the same tools get installed over and over again can become a real drag. If you are trying to do this multiple times per day—perhaps as part of development or CI—this becomes a real problem in terms of providing fast feedback. It can also lead to increased downtime when deploying in production if your systems don't allow for zero-downtime deployment, as you're waiting to install all the pre-requisites on your machines even before you get to installing your software. Models like blue/green deployment (which we'll discuss in Chapter 7) can help mitigate this, as they allow us to deploy a new version of our service without taking the old one offline.

One approach to reducing this spin-up time is to create a virtual machine image that bakes in some of the common dependencies we use, as shown in Figure 2-5. All virtualization platforms I've used allow you to build your own images, and the tools to do so are much more advanced than they were even a few years ago. This shifts things somewhat. Now we could bake the common tools into our own image. When we want to deploy our software, we spin up an

instance of this custom image, and all we have to do is install the latest version of our service.

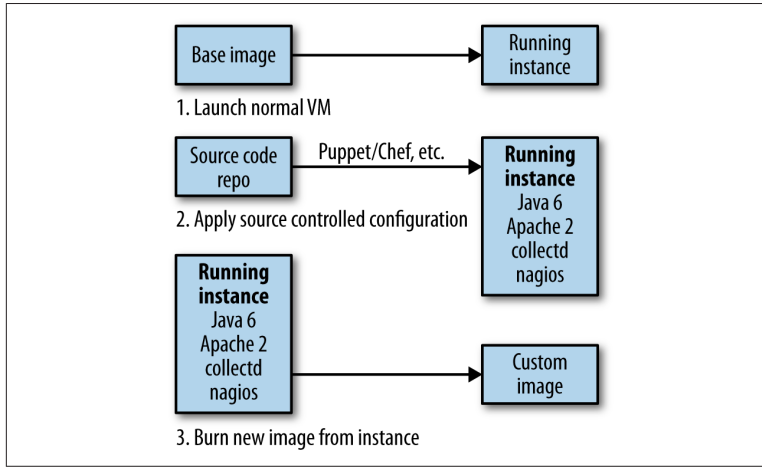


Figure 2-5. Creating a custom VM image

Of course, because you build the image only once, when you subsequently launch copies of this image you don't need to spend time installing your dependencies, as they are already there. This can result in a significant time savings. If your core dependencies don't change, new versions of your service can continue to use the same base image.

There are some drawbacks with this approach, though. Building images can take a long time. This means that for developers you may want to support other ways of deploying services to ensure they don't have to wait half an hour just to create a binary deployment. Second, some of the resulting images can be large. This could be a real problem if you're creating your own VMWare images, for example, as moving a 20GB image around a network isn't always a simple activity. We'll be looking at container technology shortly, and specifically Docker, which can avoid some of these drawbacks.

Historically, one of the challenges is that the tool chain required to build such an image varied from platform to platform. Building a VMWare image is different from building an AWS AMI, a Vagrant image, or a Rackspace image. This may not have been a problem if you had the same platform everywhere, but not all organizations were this lucky. And even if they were, the tools in this space were

often difficult to work with, and they didn't play nicely with other tools you might be using for machine configuration.

Packer is a tool designed to make creation of images much easier. Using configuration scripts of your choice (Chef, Ansible, Puppet, and more are supported), it allows us to create images for different platforms from the same configuration. At the time of writing, it has support for VMWare, AWS, Rackspace Cloud, Digital Ocean, and Vagrant, and I've seen teams use it successfully for building Linux and Windows images. This means you could create an image for deployment on your production AWS environment and a matching Vagrant image for local development and test, all from the same configuration.

Images as Artifacts

So we can create virtual machine images that bake in dependencies to speed up feedback, but why stop there? We could go further, bake our service into the image itself, and adopt the model of our service artifact being an image. Now, when we launch our image, our service is there ready to go. This really fast spin-up time is the reason that Netflix has adopted the model of baking its own services as AWS AMIs.

Just as with OS-specific packages, these VM images become a nice way of abstracting out the differences in the technology stacks used to create the services. Do we care if the service running on the image is written in Ruby or Java, and uses a gem or JAR file? All we care about is that it works. We can focus our efforts, then, on automating the creation and deployment of these images. This also becomes a really neat way to implement another deployment concept, the *immutable server*.

Immutable Servers

By storing all our configuration in source control, we are trying to ensure that we can automatically reproduce services and hopefully entire environments at will. But once we run our deployment process, what happens if someone comes along, logs into the box, and changes things independently of what is in source control? This problem is often called *configuration drift*—the code in source control no longer reflects the configuration of the running host.

To avoid this, we can ensure that no changes are ever made to a running server. Instead, any change, no matter how small, has to go through a build pipeline in order to create a new machine. You can implement this pattern without using image-based deployments, but it is also a logical extension of using images as artifacts. During our image creation, for example, we could actually disable SSH, ensuring that no one could even log onto the box to make a change!

The same caveats we discussed earlier about cycle time still apply, of course. And we also need to ensure that any data we care about that is stored on the box is stored elsewhere. These complexities aside, I've seen adopting this pattern lead to much more straightforward deployments, and easier-to-reason-about environments. And as I've already said, anything we can do to simplify things should be pursued!

Environments

As our software moves through our CD pipeline stages, it will also be deployed into different types of environments. If we think of the example build pipeline in Figure 2-4, we probably have to consider at least four distinct environments: one environment where we run our slow tests, another for UAT, another for performance, and a final one for production. Our microservice should be the same throughout, but the environment will be different. At the very least, they'll be separate, distinct collections of configuration and hosts. But often they can vary much more than that. For example, our production environment for our service might consist of multiple load-balanced hosts spread across two data centers, whereas our test environment might just have everything running on a single host. These differences in environments can introduce a few problems.

I was bitten by this personally many years ago. We were deploying a Java web service into a clustered WebLogic application container in production. This WebLogic cluster replicated session state between multiple nodes, giving us some level of resilience if a single node failed. However, the WebLogic licenses were expensive, as were the machines our software was deployed onto. This meant that in our test environment, our software was deployed on a single machine, in a nonclustered configuration.

This hurt us badly during one release. For WebLogic to be able to copy session state between nodes, the session data needs to be prop-

erly serializable. Unfortunately, one of our commits broke this, so when we deployed into production our session replication failed. We ended up resolving this by pushing hard to replicate a clustered setup in our test environment.

The service we want to deploy is the same in all these different environments, but each of the environments serves a different purpose. On my developer laptop I want to quickly deploy the service, potentially against stubbed collaborators, to run tests or carry out some manual validation of behavior, whereas when I deploy into a production environment I may want to deploy multiple copies of my service in a load-balanced fashion, perhaps split across one or more data centers for durability reasons.

As you move from your laptop to build server to UAT environment all the way to production, you'll want to ensure that your environments are more and more production-like to catch any problems associated with these environmental differences sooner. This will be a constant balance. Sometimes the time and cost to reproduce production-like environments can be prohibitive, so you have to make compromises. Additionally, sometimes using a production-like environment can slow down feedback loops; waiting for 25 machines to install your software in AWS might be much slower than simply deploying your service into a local Vagrant instance, for example.

This balance, between production-like environments and fast feedback, won't be static. Keep an eye on the bugs you find further downstream and your feedback times, and adjust this balance as required.

Managing environments for single-artfact monolithic systems can be challenging, especially if you don't have access to systems that are easily automatable. When you think about multiple environments per microservice, this can be even more daunting. We'll look shortly at some different deployment platforms that can make this much easier for us.

Service Configuration

Our services need some configuration. Ideally, this should be a small amount, and limited to those features that change from one environment to another, such as *what username and password should I*

use to connect to my database? Configuration that changes from one environment to another should be kept to an absolute minimum. The more your configuration changes fundamental service behavior, and the more that configuration varies from one environment to another, the more you will find problems only in certain environments, which is painful in the extreme.

So if we have some configuration for our service that does change from one environment to another, how should we handle this as part of our deployment process? One option is to build one artifact per environment, with configuration inside the artifact itself. Initially this seems sensible. The configuration is built right in; just deploy it and everything should work fine, right? This is problematic. Remember the concept of continuous delivery. We want to create an artifact that represents our release candidate, and move it through our pipeline, confirming that it is good enough to go into production. Let's imagine I build a Customer-Service-Test and Customer-Service-Prod artifacts. If my Customer-Service-Test artifact passes the tests, but it's the Customer-Service-Prod artifact that I actually deploy, can I be sure that I have verified the software that actually ends up in production?

There are other challenges as well. First, there is the additional time taken to build these artifacts. Next, the fact that you need to know at build time what environments exist. And how do you handle sensitive configuration data? I don't want information about production passwords checked in with my source code, but if it is needed at build time to create all those artifacts, this is often difficult to avoid.

A better approach is to create one single artifact, and manage configuration separately. This could be a properties file that exists for each environment, or different parameters passed in to an install process. Another popular option, especially when dealing with a larger number of microservices, is to use a dedicated system for providing configuration, which we'll explore more in Chapter 11.

Service-to-Host Mapping

One of the questions that comes up quite early on in the discussion around microservices is "How many services per machine?" Before we go on, we should pick a better term than *machine*, or even the more generic *box* that I used earlier. In this era of virtualization, the mapping between a single host running an operating system and the

underlying physical infrastructure can vary to a great extent. Thus, I tend to talk about *hosts*, using them as a generic unit of isolation—namely, an operating system onto which I can install and run my services. If you are deploying directly on to physical machines, then one physical server maps to one *host* (which is perhaps not completely correct terminology in this context, but in the absence of better terms may have to suffice). If you're using virtualization, a single physical machine can map to multiple independent hosts, each of which could hold one or more services.

So when thinking of different deployment models, we'll talk about hosts. So, then, how many services per host should we have?

I have a definite view as to which model is preferable, but there are a number of factors to consider when working out which model will be right for you. It's also important to understand that some choices we make in this regard will limit some of the deployment options available to us.

Multiple Services Per Host

Having multiple services per host, as shown in Figure 2-6, is attractive for a number of reasons. First, purely from a host management point of view, it is simpler. In a world where one team manages the infrastructure and another team manages the software, the infrastructure team's workload is often a function of the number of hosts it has to manage. If more services are packed on to a single host, the host management workload doesn't increase as the number of services increases. Second is cost. Even if you have access to a virtualization platform that allows you to provision and resize virtual hosts, the virtualization can add an overhead that reduces the underlying resources available to your services. In my opinion, both these problems can be addressed with new working practices and technology, and we'll explore that shortly.

This model is also familiar to those who deploy into some form of an application container. In some ways, the use of an application container is a special case of the multiple-services-per-host model, so we'll look into that separately. This model can also simplify the life of the developer. Deploying multiple services to a single host in production is synonymous with deploying multiple services to a local dev workstation or laptop. If we want to look at an alternative

model, we want to find a way to keep this conceptually simple for developers.

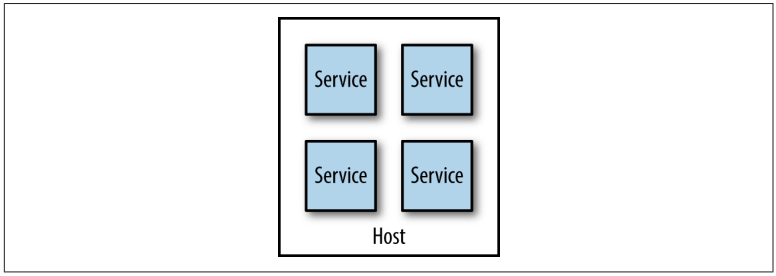


Figure 2-6. Multiple microservices per host

There are some challenges with this model, though. First, it can make monitoring more difficult. For example, when tracking CPU, do I need to track the CPU of one service independent of the others? Or do I care about the CPU of the box as a whole? Side effects can also be hard to avoid. If one service is under significant load, it can end up reducing the resources available to other parts of the system. Gilt, when scaling out the number of services it ran, hit this problem. Initially it coexisted many services on a single box, but uneven load on one of the services would have an adverse impact on everything else running on that host. This makes impact analysis of host failures more complex as well—taking a single host out of commission can have a large ripple effect.

Deployment of services can be somewhat more complex too, as ensuring one deployment doesn't affect another leads to additional headaches. For example, if I use Puppet to prepare a host, but each service has different (and potentially contradictory) dependencies, how can I make that work? In the worst-case scenario, I have seen people tie multiple service deployments together, deploying multiple different services to a single host in one step, to try to simplify the deployment of multiple services to one host. In my opinion, the small upside in improving simplicity is more than outweighed by the fact that we have given up one of the key benefits of microservices: striving for independent release of our software. If you do adopt the multiple-services-per-host model, make sure you keep hold of the idea that each service should be deployed independently.

This model can also inhibit autonomy of teams. If services for different teams are installed on the same host, who gets to configure the host for their services? In all likelihood, this ends up getting

handled by a centralized team, meaning it takes more coordination to get services deployed.

Another issue is that this option can limit our deployment artifact options. Image-based deployments are out, as are immutable servers unless you tie multiple different services together in a single artifact, which we really want to avoid.

The fact that we have multiple services on a single host means that efforts to target scaling to the service most in need of it can be complicated. Likewise, if one microservice handles data and operations that are especially sensitive, we might want to set up the underlying host differently, or perhaps even place the host itself in a separate network segment. Having everything on one host means we might end up having to treat all services the same way even if their needs are different.

As my colleague Neal Ford puts it, many of our working practices around deployment and host management are an attempt to optimize for scarcity of resources. In the past, the only option if we wanted another host was to buy or rent another physical machine. This often had a large lead time to it and resulted in a long-term financial commitment. It wasn't uncommon for clients I have worked with to provision new servers only every two to three years, and trying to get additional machines outside of these timelines was difficult. But on-demand computing platforms have drastically reduced the costs of computing resources, and improvements in virtualization technology mean even for in-house hosted infrastructure there is more flexibility.

Application Containers

If you're familiar with deploying .NET applications behind IIS or Java applications into a servlet container, you will be well acquainted with the model where multiple distinct services or applications sit inside a single application container, which in turn sits on a single host, as we see in Figure 2-7. The idea is that the application container your services live in gives you benefits in terms of improved manageability, such as clustering support to handle grouping multiple instances together, monitoring tools, and the like.

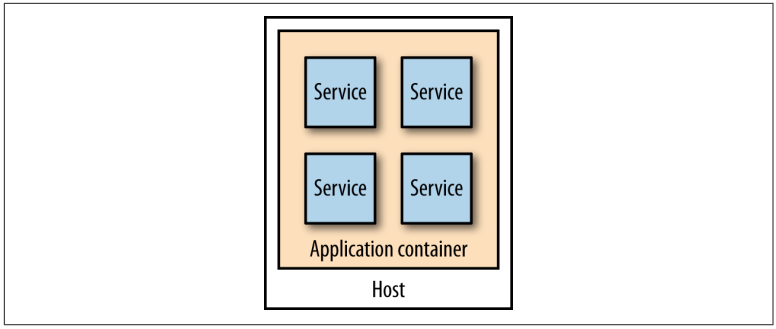


Figure 2-7. Multiple microservices per host

This setup can also yield benefits in terms of reducing overhead of language runtimes. Consider running five Java services in a single Java servlet container. I only have the overhead of one single JVM. Compare this with running five independent JVMs on the same host when using embedded containers. That said, I still feel that these application containers have enough downsides that you should challenge yourself to see if they are really required.

First among the downsides is that they inevitably constrain technology choice. You have to buy into a technology stack. This can limit not only the technology choices for the implementation of the service itself, but also the options you have in terms of automation and management of your systems. As we'll discuss shortly, one of the ways we can address the overhead of managing multiple hosts is around automation, and so constraining our options for resolving this may well be doubly damaging.

I would also question some of the value of the container features. Many of them tout the ability to manage clusters to support shared in-memory session state, something we absolutely want to avoid in any case due to the challenges this creates when scaling our services. And the monitoring capabilities they provide won't be sufficient when we consider the sorts of joined-up monitoring we want to do in a microservices world, as we'll see in Chapter 8. Many of them also have quite slow spin-up times, impacting developer feedback cycles.

There are other sets of problems too. Attempting to do proper lifecycle management of applications on top of platforms like the JVM can be problematic, and more complex than simply restarting a JVM. Analyzing resource use and threads is also much more com-

plex, as you have multiple applications sharing the same process. And remember, even if you do get value from a technology-specific container, they aren't free. Aside from the fact that many of them are commercial and so have a cost implication, they add a resource overhead in and of themselves.

Ultimately, this approach is again an attempt to optimize for scarcity of resources that simply may not hold up anymore. Whether you decide to have multiple services per host as a deployment model, I would strongly suggest looking at self-contained deployable microservices as artifacts. For .NET, this is possible with things like Nancy, and Java has supported this model for years. For example, the venerable Jetty embedded container makes for a very lightweight self-contained HTTP server, which is the core of the Dropwizard stack. Google has been known to quite happily use embedded Jetty containers for serving static content directly, so we know these things can operate at scale.

Single Service Per Host

With a single-service-per-host model shown in Figure 2-8, we avoid side effects of multiple hosts living on a single host, making monitoring and remediation much simpler. We have potentially reduced our single points of failure. An outage to one host should impact only a single service, although that isn't always clear when you're using a virtualized platform. We'll cover designing for scale and failure more in Chapter 11. We also can more easily scale one service independent from others, and deal with security concerns more easily by focusing our attention only on the service and host that requires it.

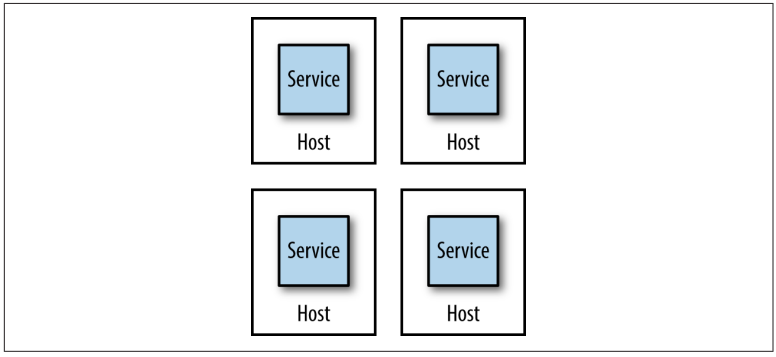


Figure 2-8. A single microservice per host

Just as important is that we have opened up the potential to use alternative deployment techniques such as image-based deployments or the immutable server pattern, which we discussed earlier.

We've added a lot of complexity in adopting a microservice architecture. The last thing we want to do is go looking for more sources of complexity. In my opinion, if you don't have a viable PaaS available, then this model does a very good job of reducing a system's overall complexity. Having a single-service-per-host model is significantly easier to reason about and can help reduce complexity. If you can't embrace this model yet, I won't say microservices aren't for you. But I would suggest that you look to move toward this model over time as a way of reducing the complexity that a microservice architecture can bring.

Having an increased number of hosts has potential downsides, though. We have more servers to manage, and there might also be a cost implication of running more distinct hosts. Despite these problems, this is still the model I prefer for microservice architectures. And we'll talk about a few things we can do to reduce the overhead of handling large numbers of hosts shortly.

Platform as a Service

When using a platform as a service (PaaS), you are working at a higher-level abstraction than at a single host. Most of these platforms rely on taking a technology-specific artifact, such as a Java WAR file or Ruby gem, and automatically provisioning and running it for you. Some of these platforms will transparently attempt to handle scaling the system up and down for you, although a more common (and in my experience less error-prone) way will allow you some control over how many nodes your service might run on, but it handles the rest.

At the time of writing, most of the best, most polished PaaS solutions are hosted. Heroku comes to mind as being probably the gold class of PaaS. It doesn't just handle running your service, it also supports services like databases in a very simple fashion. Self-hosted solutions do exist in this space, although they are more immature than the hosted solutions.

When PaaS solutions work well, they work very well indeed. However, when they don't quite work for you, you often don't have much control in terms of getting under the hood to fix things. This is part

of the trade-off you make. I would say that in my experience the smarter the PaaS solutions try to be, the more they go wrong. I've used more than one PaaS that attempts to autoscale based on application use, but does it badly. Invariably the heuristics that drive these smarts tend to be tailored for the average application rather than your specific use case. The more nonstandard your application, the more likely it is that it might not play nicely with a PaaS.

As the good PaaS solutions handle so much for you, they can be an excellent way of handling the increased overhead we get with having many more moving parts. That said, I'm still not sure that we have all the models right in this space yet, and the limited self-hosted options mean that this approach might not work for you. In the coming decade though I expect we'll be targeting PaaS for deployment more than having to self-manage hosts and deployments of individual services.

Automation

The answer to so many problems we have raised so far comes down to automation. With a small number of machines, it is possible to manage everything manually. I used to do this. I remember running a small set of production machines, and I would collect logs, deploy software, and check processes by manually logging in to the box. My productivity seemed to be constrained by the number of terminal windows I could have open at once—a second monitor was a huge step up. This breaks down really fast, though.

One of the pushbacks against the single-service-per-host setup is the perception that the amount of overhead to manage these hosts will increase. This is certainly true if you are doing everything manually. Double the servers, double the work! But if we automate control of our hosts, and deployment of the services, then there is no reason why adding more hosts should increase our workload in a linear fashion.

But even if we keep the number of hosts small, we still are going to have lots of services. That means multiple deployments to handle, services to monitor, logs to collect. Automation is essential.

Automation is also how we can make sure that our developers still remain productive. Giving them the ability to self-service-provision individual services or groups of services is key to making develop-

ers' lives easier. Ideally, developers should have access to exactly the same tool chain as is used for deployment of our production services so as to ensure that we can spot problems early on. We'll be looking at a lot of technology in this chapter that embraces this view.

Picking technology that enables automation is highly important. This starts with the tools used to manage hosts. Can you write a line of code to launch a virtual machine, or shut one down? Can you deploy the software you have written automatically? Can you deploy database changes without manual intervention? Embracing a culture of automation is key if you want to keep the complexities of microservice architectures in check.

Two Case Studies on the Power of Automation

It is probably helpful to give you a couple of concrete examples that explain the power of good automation. One of our clients in Australia is RealEstate.com.au (REA). Among other things, the company provides real estate listings for retail and commercial customers in Australia and elsewhere in the Asia-Pacific region. Over a number of years, it has been moving its platform toward a distributed, microservices design. When it started on this journey it had to spend a lot of time getting the tooling around the services just right—making it easy for developers to provision machines, to deploy their code, or monitor them. This caused a front-loading of work to get things started.

In the first three months of this exercise, REA was able to move just two new microservices into production, with the development team taking full responsibility for the entire build, deployment, and support of the services. In the next three months, between 10–15 services went live in a similar manner. By the end of the 18-month period, REA had over 60–70 services.

This sort of pattern is also borne out by the experiences of Gilt, an online fashion retailer that started in 2007. Gilt's monolithic Rails application was starting to become difficult to scale, and the company decided in 2009 to start decomposing the system into microservices. Again automation, especially tooling to help developers, was given as a key reason to drive Gilt's explosion in the use of microservices. A year later, Gilt had around 10 microservices live; by 2012, over 100; and in 2014, over 450 microservices by Gilt's own

count—in other words, around three services for every developer in Gilt.

From Physical to Virtual

One of the key tools available to us in managing a large number of hosts is finding ways of chunking up existing physical machines into smaller parts. Traditional virtualization like VMWare or that used by AWS has yielded huge benefits in reducing the overhead of host management. However, there have been some new advances in this space that are well worth exploring, as they can open up even more interesting possibilities for dealing with our microservice architecture.

Traditional Virtualization

Why is having lots of hosts expensive? Well, if you need a physical server per host, the answer is fairly obvious. If this is the world you are operating in, then the multiple-service-per-host model is probably right for you, although don't be surprised if this becomes an ever more challenging constraint. I suspect, however, that most of you are using virtualization of some sort. Virtualization allows us to slice up a physical server into separate hosts, each of which can run different things. So if we want one service per host, can't we just slice up our physical infrastructure into smaller and smaller pieces?

Well, for some people, you can. However, slicing up the machine into ever increasing VMs isn't free. Think of our physical machine as a sock drawer. If we put lots of wooden dividers into our drawer, can we store more socks or fewer? The answer is fewer: the dividers themselves take up room too! Our drawer might be easier to deal with and organize, and perhaps we could decide to put T-shirts in one of the spaces now rather than just socks, but more dividers means less overall space.

In the world of virtualization, we have a similar overhead as our sock drawer dividers. To understand where this overhead comes from, let's look at how most virtualization is done. Figure 2-9 shows a comparison of two types of virtualization. On the left, we see the various layers involved in what is called *type 2 virtualization*, which is the sort implemented by AWS, VMWare, VSphere, Xen, and KVM. (Type 1 virtualization refers to technology where the VMs run directly on hardware, not on top of another operating system.)

On our physical infrastructure we have a host operating system. On this OS we run something called a *hypervisor*, which has two key jobs. First, it maps resources like CPU and memory from the virtual host to the physical host. Second, it acts as a control layer, allowing us to manipulate the virtual machines themselves.

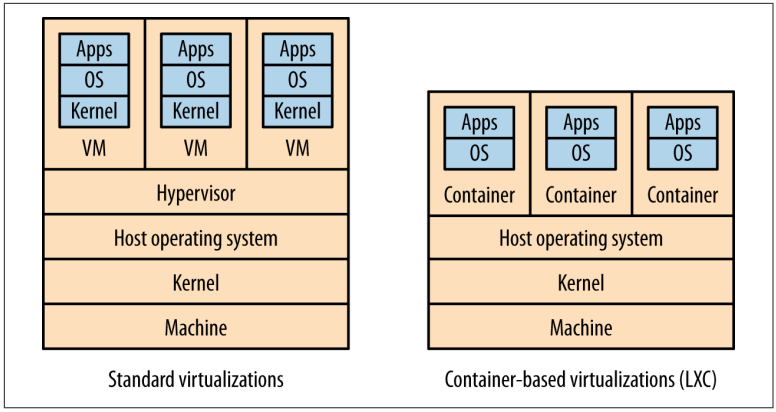


Figure 2-9. A comparison of standard Type 2 virtualization, and light-weight containers

Inside the VMs, we get what looks like completely different hosts. They can run their own operating systems, with their own kernels. They can be considered almost hermetically sealed machines, kept isolated from the underlying physical host and the other virtual machines by the hypervisor.

The problem is that the hypervisor here needs to set aside resources to do its job. This takes away CPU, I/O, and memory that could be used elsewhere. The more hosts the hypervisor manages, the more resources it needs. At a certain point, this overhead becomes a constraint in slicing up your physical infrastructure any further. In practice, this means that there are often diminishing returns in slicing up a physical box into smaller and smaller parts, as proportionally more and more resources go into the overhead of the hypervisor.

Vagrant

Vagrant is a very useful deployment platform, which is normally used for dev and test rather than production. Vagrant provides you with a virtual cloud on your laptop. Underneath, it uses a standard virtualization system (typically VirtualBox, although it can use other

platforms). It allows you to define a set of VMs in a text file, along with how the VMs are networked together and which images the VMs should be based on. This text file can be checked in and shared between team members.

This makes it easier for you to create production-like environments on your local machine. You can spin up multiple VMs at a time, shut individual ones to test failure modes, and have the VMs mapped through to local directories so you can make changes and see them reflected immediately. Even for teams using on-demand cloud platforms like AWS, the faster turnaround of using Vagrant can be a huge boon for development teams.

One of the downsides, though, is that running lots of VMs can tax the average development machine. If we have one service to one VM, you may not be able to bring up your entire system on your local machine. This can result in the need to stub out some dependencies to make things manageable, which is one more thing you'll have to handle to ensure that the development and test experience is a good one.

Linux Containers

For Linux users, there is an alternative to virtualization. Rather than having a hypervisor to segment and control separate virtual hosts, Linux containers instead create a separate process space in which other processes live.

On Linux, process are run by a given user, and have certain capabilities based on how the permissions are set. Processes can spawn other processes. For example, if I launch a process in a terminal, that child process is generally considered a child of the terminal process. The Linux kernel's job is maintaining this tree of processes.

Linux containers extend this idea. Each container is effectively a subtree of the overall system process tree. These containers can have physical resources allocated to them, something the kernel handles for us. This general approach has been around in many forms, such as Solaris Zones and OpenVZ, but it is LXC that has become most popular. LXC is now available out of the box in any modern Linux kernel.

If we look at a stack diagram for a host running LXC in Figure 2-9, we see a few differences. First, we don't need a hypervisor. Second,

although each container can run its own operating system distribution, it has to share the same kernel (because the kernel is where the process tree lives). This means that our host operating system could run Ubuntu, and our containers CentOS, as long as they could both share the same kernel.

We don't just benefit from the resources saved by not needing a hypervisor. We also gain in terms of feedback. Linux containers are *much* faster to provision than full-fat virtual machines. It isn't uncommon for a VM to take many minutes to start—but with Linux containers, startup can take a few seconds. You also have finer-grained control over the containers themselves in terms of assigning resources to them, which makes it much easier to tweak the settings to get the most out of the underlying hardware.

Due to the lighter-weight nature of containers, we can have many more of them running on the same hardware than would be possible with VMs. By deploying one service per container, as in Figure 2-10, we get a degree of isolation from other containers (although this isn't perfect), and can do so much more cost effectively than would be possible if we wanted to run each service in its own VM.

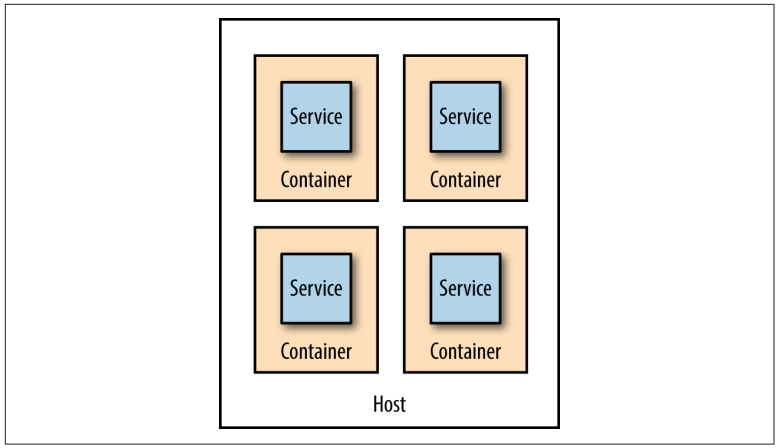


Figure 2-10. Running services in separate containers

Containers can be used well with full-fat virtualization too. I've seen more than one project provision a large AWS EC2 instance and run LXC containers on it to get the best of both worlds: an on-demand

ephemeral compute platform in the form of EC2, coupled with highly flexible and fast containers running on top of it.

Linux containers aren't without some problems, however. Imagine I have lots of microservices running in their own containers on a host. How does the outside world see them? You need some way to route the outside world through to the underlying containers, something many of the hypervisors do for you with normal virtualization. I've seen many a person sink inordinate amounts of time into configuring port forwarding using IPTables to expose containers directly. Another point to bear in mind is that these containers cannot be considered completely sealed from each other. There are many documented and known ways in which a process from one container can bust out and interact with other containers or the underlying host. Some of these problems are by design and some are bugs that are being addressed, but either way if you don't trust the code you are running, don't expect that you can run it in a container and be safe. If you need that sort of isolation, you'll need to consider using virtual machines instead.

Docker

Docker is a platform built on top of lightweight containers. It handles much of the work around handling containers for you. In Docker, you create and deploy *apps*, which are synonymous with images in the VM world, albeit for a container-based platform. Docker manages the container provisioning, handles some of the networking problems for you, and even provides its own registry concept that allows you to store and version Docker applications.

The Docker app abstraction is a useful one for us, because just as with VM images the underlying technology used to implement the service is hidden from us. We have our builds for our services create Docker applications, and store them in the Docker registry, and away we go.

Docker can also alleviate some of the downsides of running lots of services locally for dev and test purposes. Rather than using Vagrant to host multiple independent VMs, each one containing its own service, we can host a single VM in Vagrant that runs a Docker instance. We then use Vagrant to set up and tear down the Docker platform itself, and use Docker for fast provisioning of individual services.

A number of different technologies are being developed to take advantage of Docker. CoreOS is a very interesting operating system designed with Docker in mind. It is a stripped-down Linux OS that provides only the essential services to allow Docker to run. This means it consumes fewer resources than other operating systems, making it possible to dedicate even more resources of the underlying machine to our containers. Rather than using a package manager like debs or RPMs, all software is installed as independent Docker apps, each running in its own container.

Docker itself doesn't solve all problems for us. Think of it as a simple PaaS that works on a single machine. If you want tools to help you manage services across multiple Docker instances across multiple machines, you'll need to look at other software that adds these capabilities. There is a key need for a scheduling layer that lets you request a container and then finds a Docker container that can run it for you. In this space, Google's recently open sourced Kubernetes and CoreOS's cluster technology can help, and it seems every month there is a new entrant in this space. Deis is another interesting tool based on Docker, which is attempting to provide a Heroku-like PaaS on top of Docker.

I talked earlier about PaaS solutions. My struggle with them has always been that they often get the abstraction level wrong, and that self-hosted solutions lag significantly behind hosted solutions like Heroku. Docker gets much more of this right, and the explosion of interest in this space means I suspect it will become a much more viable platform for all sorts of deployments over the next few years for all sorts of different use cases. In many ways, Docker with an appropriate scheduling layer sits between IaaS and PaaS solutions—the term *containers as a service (CaaS)* is already being used to describe it.

Docker is being used in production by multiple companies. It provides many of the benefits of lightweight containers in terms of efficiency and speed of provisioning, together with the tools to avoid many of the downsides. If you are interested in looking at alternative deployment platforms, I'd strongly suggest you give Docker a look.

A Deployment Interface

Whatever underlying platform or artifacts you use, having a uniform interface to deploy a given service is vital. We'll want to trigger

deployment of a microservice on demand in a variety of different situations, from deployments locally for dev and test to production deployments. We'll also want to keep our deployment mechanisms as similar as possible from dev to production, as the last thing we want is to find ourselves hitting problems in production because deployment uses a completely different process!

After many years of working in this space, I am convinced that the most sensible way to trigger any deployment is via a single, parameterizable command-line call. This can be triggered by scripts, launched by your CI tool, or typed in by hand. I've built wrapper scripts in a variety of technology stacks to make this work, from Windows batch, to bash, to Python Fabric scripts, and more, but all of the command lines share the same basic format.

We need to know what we are deploying, so we need to provide the name of a known entity, or in our case a microservice. We also need to know what version of the entity we want. The answer to *what version* tends to be one of three possibilities. When you're working locally, it'll be whatever version is on your local machine. When testing, you'll want the latest *green* build, which could just be the most recent blessed artifact in our artifact repository. Or when testing/diagnosing issues, we may want to deploy an exact build.

The third and final thing we'll need to know is what environment we want the microservice deployed into. As we discussed earlier, our microservice's topology may differ from one environment to the next, but that should be hidden from us here.

So, imagine we create a simple `deploy` script that takes these three parameters. Say we're developing locally and want to deploy our catalog service into our local environment. I might type:

```
$ deploy artifact=catalog environment=local version=local
```

Once I've checked in, our CI build service picks up the change and creates a new build artifact, giving it the build number `b456`. As is standard in most CI tools, this value gets passed along the pipeline. When our test stage gets triggered, the CI stage will run:

```
$ deploy artifact=catalog environment=ci version=b456
```

Meanwhile, our QA wants to pull the latest version of the catalog service into an integrated test environment to do some exploratory testing, and to help with a showcase. That team runs:

```
$ deploy artifact=catalog environment=integrated_qa ver-
sion=latest
```

The tool I've used the most for this is Fabric, a Python library designed to map command-line calls to functions, along with good support for handling tasks like SSH into remote machines. Pair it with an AWS client library like Boto, and you have everything you need to fully automate very large AWS environments. For Ruby, Capistrano is similar in some ways to Fabric, and on Windows you could go a long way using PowerShell.

Environment Definition

Clearly, for this to work, we need to have some way of defining what our environments look like, and what our service looks like in a given environment. You can think of an environment definition as a mapping from a microservice to compute, network, and storage resources. I've done this with YAML files before, and used my scripts to pull this data in. Example 2-1 is a simplified version of some work I did a couple of years ago for a project that used AWS.

Example 2-1. An example environment definition

```
development:
  nodes:
  - ami_id: ami-e1e1234
    size:   t1.micro ❶
    credentials_name: eu-west-ssh ❷
    services: [catalog-service]
    region: eu-west-1

production:
  nodes:
  - ami_id: ami-e1e1234
    size:   m3.xlarge ❶
    credentials_name: prod-credentials ❷
    services: [catalog-service]
    number: 5  ❸
```

❶ We varied the size of the instances we used to be more cost effective. You don't need a 16-core box with 64GB of RAM for exploratory testing!

❷ Being able to specify different credentials for different environments is key. Credentials for sensitive environments were stored in different source code repos that only select people would have access to.

❸ We decided that by default if a service had more than one node configured, we would automatically create a load balancer for it.

I have removed some detail for the sake of brevity.

The `catalog-service` information was stored elsewhere. It didn't differ from one environment to the next, as you can see in Example 2-2.

Example 2-2. An example environment definition

```
catalog-service:
  puppet_manifest : catalog.pp ❶
  connectivity:
    - protocol: tcp
      ports: [ 8080, 8081 ]
      allowed: [ WORLD ]
```

❶ This was the name of the Puppet file to run—we happened to use Puppet solo in this situation, but theoretically could have supported alternative configuration systems.

Obviously, a lot of the behavior here was convention based. For example, we decided to normalize which ports services used wherever they ran, and automatically configured load balancers if a service had more than one instance (something that AWS's ELBs make fairly easy).

Building a system like this required a significant amount of work. The effort is often front-loaded, but can be essential to manage the deployment complexity you have. I hope in the future you won't have to do this yourself. Terraform is a very new tool from Hashicorp, which works in this space. I'd generally shy away from mentioning such a new tool in a book that is more about ideas than technology, but it is attempting to create an open source tool along these lines. It's early days yet, but already its capabilities seem really interesting. With the ability to target deployments on a number of different platforms, in the future it could be just the tool for the job.

Summary

We've covered a lot of ground here, so a recap is in order. First, focus on maintaining the ability to release one service independently from another, and make sure that whatever technology you select supports this. I greatly prefer having a single repository per microservice, but am firmer still that you need one CI build per microservice if you want to deploy them separately.

Next, if possible, move to a single-service per host/container. Look at alternative technologies like LXC or Docker to make managing the moving parts cheaper and easier, but understand that whatever technology you adopt, a culture of automation is key to managing everything. Automate everything, and if the technology you have doesn't allow this, get some new technology! Being able to use a platform like AWS will give you huge benefits when it comes to automation.

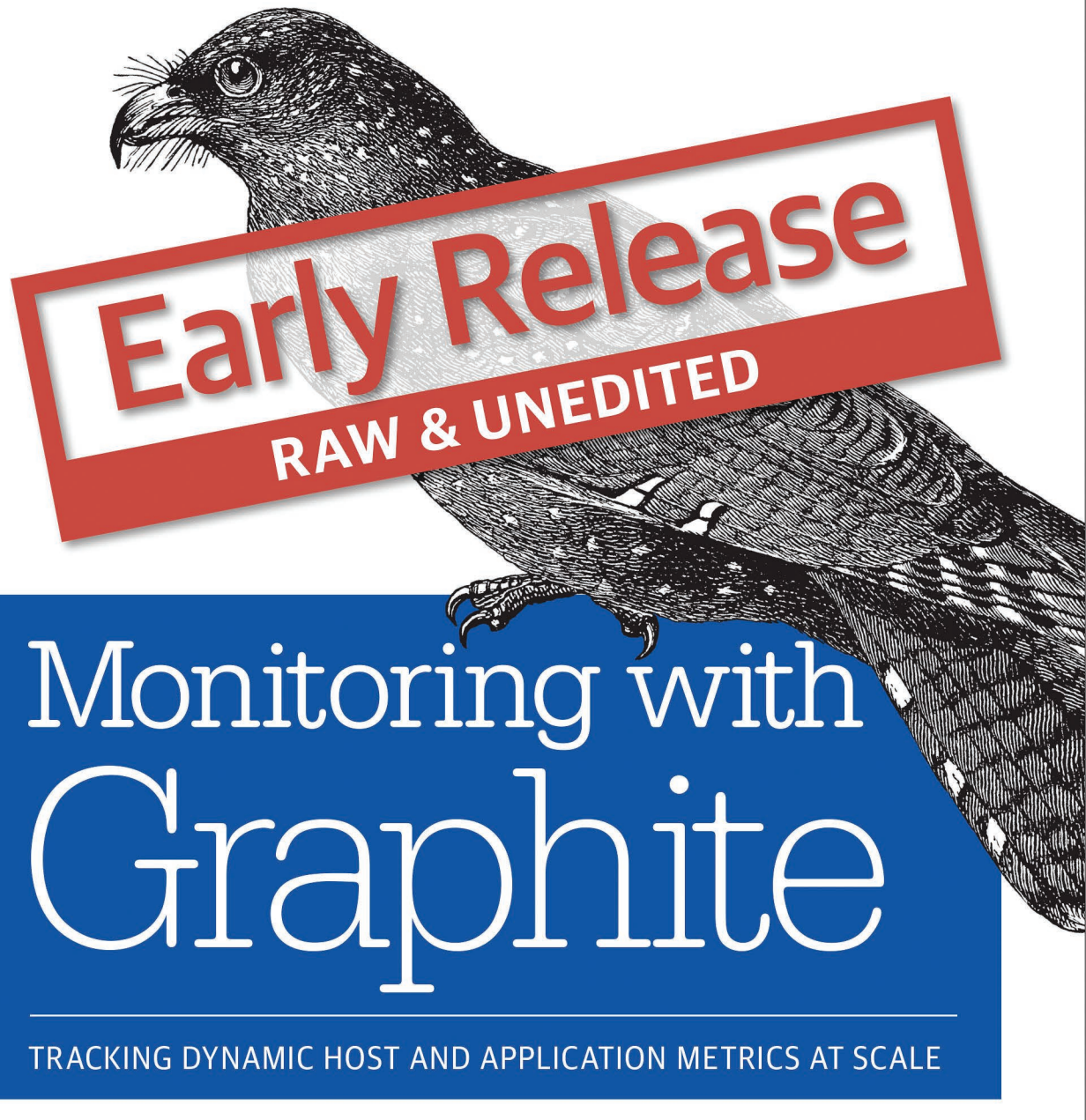
Make sure you understand the impact your deployment choices have on developers, and make sure they feel the love too. Creating tools that let you self-service-deploy any given service into a number

of different environments is really important, and will help developers, testers, and operations people alike.

Finally, if you want to go deeper into this topic, I thoroughly recommend you read Jez Humble and David Farley's *Continuous Delivery* (Addison-Wesley), which goes into much more detail on subjects like pipeline design and artifact management.

In the next chapter, we'll be going deeper into a topic we touched on briefly here. Namely, how do we test our microservices to make sure they actually work?

O'REILLY®



Monitoring with Graphite

TRACKING DYNAMIC HOST AND APPLICATION METRICS AT SCALE

Jason Dixon

Monitoring Conventions

The following content is excerpted from *Monitoring with Graphite*, by Jason Dixon. Available now in Early Release.

Everyone seems to have a different definition for what *Monitoring* means to them. Many folks know it as a convential *polling* system like Nagios. For others it might mean walking their networks with SNMP and Cacti, or perhaps even a bespoke collection of Perl scripts and artisinal cron jobs. Some companies don't run internal monitoring systems at all, perferring to outsource all or part of their monitoring to hosted monitoring services. No matter which tools you cobble together or how you orchestrate them, most of these systems cover a common set of operational responsibilities.

From my experiences, it's important that we, the maintainers and users of these *Monitoring Architectures*, share a common vocabulary and understanding of the logical areas of functionality that make up these systems. Describing to your peers how you "*instrument* your application *telemetry* and *aggregate* the results (because they report irregularly) before firing them off to your *trending* system for *correlation* and *fault detection*" is almost certainly going to explain more about your setup than "we monitor stuff".

Don't get me wrong, I'm all for brevity, but words really do matter. And for better or worse, we've got enough of them to choke a horse. Although those two expressions may in theory be saying roughly the same thing, the former tells us a lot more about what you do and how it adds value to your organization. Above all else, being able to speak lucidly about your monitoring systems can go a long way

towards building trust with your customers, your engineering and operations teams, and your business leadership.

Three Tenets of Monitoring

Monitoring is a generic way to describe the collection of software and processes we use to ensure the availability and health of our IT systems and services. In an abstract sense, monitoring can be broken up into three main categories: *Fault Detection*, *Alerting*, and *Capacity Planning*. Each of these can be dissected even further into specific functional tasks, but we'll get to that later. For now let's take a brief look at these broader concepts since they form the basis for most of the legacy approaches among monitoring vendors and Open Source projects.

Fault Detection

The primary goal of any monitoring system should be to identify when a resource (or collection of resources) becomes unavailable or starts to perform poorly. We use *Fault Detection* techniques to compare the current state of an asset against a known good (or simply *operational*) state. Traditionally we employ *thresholds* to recognize the delta in a system's behavior.

For example, we might want to know when our webserver's "time to first byte" (i.e. the amount of time it takes for a client to download the first byte of content from a web site or application) exceeds 300 milliseconds. Our monitoring software should be able to review the metric - either by polling the service directly or reviewing the data provided from a dedicated collection agent, detect when the webserver becomes sluggish (typically by polling the service directly), classify it as a fault, and escalate the *Alert* to the responsible parties (usually a very tired on-call person at 3am).

With the emergence of Big Data and other data science pursuits, we're starting to see new and exciting *anomaly detection* techniques and algorithms applied to the area of IT monitoring. Basic thresholds are very much a practice in *trial and error*. Conversely, these new approaches attempt to leverage the advances in Machine Learn-

ing to automate fault detection software to be more intelligent and proactive, and ideally, result in fewer false alarms.

False Alarms

You're bound to hear monitoring folks refer to false positives or negatives when talking about fault detection and alerting behavior. A *false positive* is an alert triggered by our monitoring threshold, either by accident or misconfiguration. Not only are these annoying, but they can quickly lead to *pager fatigue* and a general distrust of the monitoring system.

On the other hand, *false negatives* are alerts that our monitoring system fails to detect. These are usually caused by improper thresholds, but can also be triggered by improper check intervals (running at the wrong times or too infrequently), or simply a lack of proper checks. Unfortunately, false negatives are usually identified too late, as a consequence of a host or service outage.

Alerting

> If my answers frighten you then you should cease asking scary questions.
>
> —Jules Winnfield

If you've been in IT for a while, there's a good chance you've had to carry a pager after business hours. Even worse, it's probably woken you (and your significant other) in the middle of a lovely dream, or interrupted a fun night out with your friends. I don't know anyone that enjoys *Alerting*, but I think we can all acknowledge that it's a necessary part of monitoring.

For all practial purposes, alerting constitutes the moment that your monitoring system identifies a fault that demands some further action. In most cases the system is designed to send an email or page to a member of your engineering or operations teams. But getting your alert to its destination is not always as simple as it sounds. Sometimes the recipient may be out of range of cellular service or simply have their ringer turned off (accidentally, I'm sure). This is why we should also include on-call scheduling, notification routing, and escalations in any discussion of alerting.

When your team grows large enough to "pass the pager around", it's a good time to start looking at various on-call scheduling approaches. It's common to see team members trade on-call responsibilities every week, every few days, or even *every day*, depending on the severity and "pain levels" of your rotation. Ideally, you want the handoff to occur during normal business hours when both individuals are awake, alert (excuse the pun), and the person coming off their rotation can pass along valuable information from their shift.

The good news is that as the size of your team increases, the amount of time each person has to cover on-call decreases proportionately. The bad news is that on-call scheduling software has traditionally been pretty awful. Until the last few years, most companies had to resort to manually updating their alerting configuration with the new recipient and any escalation paths. If you were lucky, someone wrote a script to manage this automatically, but these were susceptible to bugs and typos (like all software), often resulting in false negatives.

These days, it's much more common to see companies route their alerts through an external alert service like PagerDuty. A cottage industry of alert management companies have emerged over the last few years, offering notification routing, escalation, and on-call scheduling services. As someone whose lived through the lean years before outsourced alert management, I strongly recommend investing in any of these services or even their Open Source counterparts.

Capacity Planning

Anyone whose ever been asked to predict the growth of their application, customer base, or the budget for next year's IT purchases has been faced with the need for *Capacity Planning*. Heck, even the act of being alerted when your server's disk hits 90% capacity is a simple version of capacity planning (albeit a very poor one). This might not be something you're asked to do very often (or at all), but if you're tracking your server and application metrics in Graphite, there's a good chance you'll be prepared when the opportunity presents itself.

The act of capacity planning is really just the ability to studying trends in your data and use that knowledge to make informed decisions for adding capacity now, or in the near future. Time-series data plays a powerful role in executing your capacity planning strategy successfully. If you intend to use Graphite for this (and why

wouldn't you), at some point you'll face the tradeoff between storing your metrics for as long as possible and the increased disk space requirements needed for supporting a lengthy retention policy.

Obviously, nobody wants to keep their data around for a few years because they think it's going to be useful for current troubleshooting efforts; they use these older metrics to observe annual trends in our systems and business. Although this typically means *downsampling* your metrics to save on disk space, having this data available (even at lower precision) is invaluable for studying long-term trends. In fact, it's not uncommon to see users with ten-year retention policies!

We're not going to cover capacity planning at depth in this book, but it's important to acknowledge its purpose and understand its relationship with the sort of time-series data you'll collect with Graphite. For a proper treatment on the topic, I highly recommend John Allspaw's *The Art of Capacity Planning*, and *Guerrilla Capacity Planning* by Dr. Neil J. Gunther.

In most circumstances you'll hear me refer to *Trending* since we're more interested in the broad application of storing, retrieving and analyzing time-series data, but rest assured Capacity Planning is always hiding in the shadows, ready to pounce at a moment's notice.

Rethinking the Poll/Pull Model

Having a shared vocabulary of logical monitoring systems is important, but one of the biggest advancements in recent history has been the deconstruction of the monolithic monitoring system into discrete functional components. As a result, we understand our functional competencies and responsibilities better than ever before. But perhaps one of the biggest developments is the explosion of new tools and frameworks in the monitoring space.

For years, businesses were resigned to the fact that Nagios was effectively the only choice in Open Source tools for host and service monitoring. Because it was "good enough" for most use cases, there was never enough "operational angst" to drive innovation in the space.

If we look back over the last five years we can see parallels between the development of *Service Oriented Architectures* (more recently recoined as *Microservices*) and the advancement of Open Source monitoring tools. Users and developers began to recognize that you

didn't have to reinvent the entire wheel; it was enough to build and publish small, sharp tools that emulated the legacy interfaces and covered specific functional areas: alerting, notifications, graphs, etc. These days you'll be hard pressed to find many businesses that don't use a variety components to form their monitoring architecture.

While it's true that most CIOs would prefer the simplicity of a single vendor that offers the traditional monolithic "Enterprise Monitoring Suite", the reality is that all businesses are different and there is no such thing as "one size fits all" when it comes to monitoring systems. If you take away nothing else from this book, please remember that axiom. I promise it will save you (and those you care about) plenty of tears and heartache.

Before we dive into the features of a modern monitoring stack, I want you to forget everything you think you know about monitoring systems. Whether you're the CTO of a San Francisco-based startup or a battle-tested Systems Administrator working on government contracts, you probably have a lot of preconceptions about what a monitoring system *should* look like. Try to forget, at least for a bit, that you ever heard of Nagios and NRPE; I'm about to free your mind.

Pull Model

> Then you'll see, that it is not the spoon that bends, it is only yourself.
>
> —Spoon Boy

The traditional approach to IT monitoring centers around a *polling* agent that spends a great deal of time and resources connecting to remote servers or appliances in an effort to determine their current status: are they reachable on the network with *ping*; what does their SNMP output tell me about their *CPU usage*; can I execute a remote command on that host and interpret the results?

This is what we generally refer to as a *pull model* in that we're pulling (or polling) the target systems at regular intervals. Historically, we've asked very simple questions of our systems ("is it alive") and organized our operations staff and monitoring software around the goal of minimizing downtime. Sadly, this characterization reinforced the legacy perception of the IT department as a cost center.

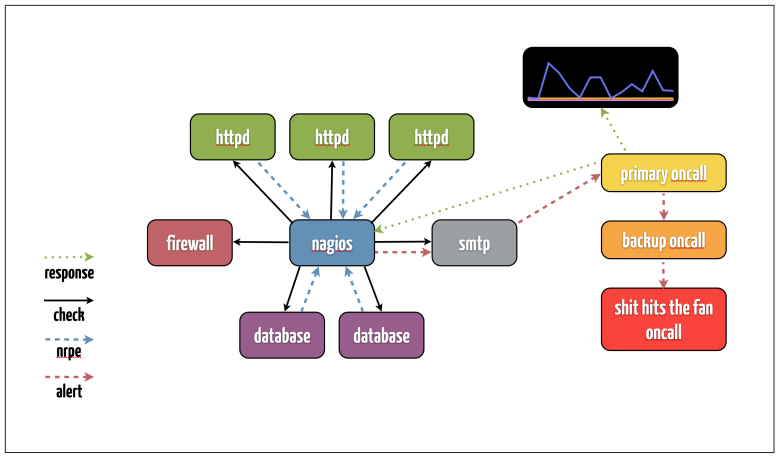


Figure 3-1. Legacy Pull Model

A number of factors have caused IT departments and Operations teams to start being viewed in a much more positive (and lucrative) light. Thanks to the popularity of elastic computing resources ("The Cloud"), improved automation through configuration management, and our increased reliance on Software as a Service (SaaS), businesses have begun leveraging their IT investment into cost efficiencies in other departments. In many ways, the modern Operations team can be as much of a profit center as the traditional sales channel (ok, maybe not *quite as much*, but you get the idea).

Of course, this means that we need to measure our IT performance with vigor and real qualitative numbers. It's not that we aren't already doing this, but in many cases we treat *Monitoring* and *Trending* as two completely distinct stacks of information. Businesses deploy "Enterprise Software X" to monitor uptime and general system health, and then deploy a separate instance of "Enterprise Software Y" to gather SNMP data and render graphs. There's still a massive amount of duplication of effort; not to mention all the Nagios performance data that's frequently discarded (the default behavior for Nagios).

Push Model

Fortunately, many companies are beginning to recognize the folly in this model. The performance data previously lost is now collected and stored at high precision and used to power questions relating to availability and quality of service. Metrics are *pushed* from their

sources to a unified storage repository, arming us with a consolidated set of data that we can use to drive both our IT responses *and* our business decisions. We can use the exact same metrics to measure the health and availability of our systems.

There's a great deal of flexibility that comes with instrumenting our systems to *send* telemetry data, rather than pulling it manually. Collection tasks are decentralized, so we no longer need to scale our collection systems vertically as our architecture scales horizontally (i.e. to many nodes). Systems report in as they're available; no need to deal with timeouts, disconnects or retry attempts. Each system can use the transport mechanism best suited to their design or environment; servers can be tooled to leverage TCP sockets, message queues, or plain old log streams.

But one of my favorite aspects of this *push model* is that we can begin isolating the functional responsibilities of our monitoring systems. We're no longer forced to deal with a monolithic "black box" to manage our IT assets. In fact, as we begin to understand these discrete functional units, we can begin looking to other industries (e.g. airlines, medicine, etc) and apply their best practices to our own.

Where Does Graphite Fit into the Picture?

Glad you asked. Truth be told, Graphite often comes into play at almost every step of the monitoring lifecycle. Does this mean Graphite is one of those terrible monolithic applications I mentioned earlier? Not at all. But if you're in a position to use Graphite as your "source of truth", it is capable of fulfilling different roles at various points of the monitoring architecture, thanks to it's flexible design, well-defined service interfaces and API coverage.

Having a centralized, canonical system of record for all metrics and state is a key part of any responsible and trustworthy monitoring architecture. In legacy systems it was common to store network accounting data in one system, host and service monitoring state in another system, and capacity planning telemetry in yet another. This lack of a unified source of truth often resulted in conflicting information, or worse, the inability (or great difficulty) to correlate disparate data sources.

Now there are inevitably times when it makes sense to have some overlap (or even duplication) of measurement data, particularly when the types of questions asked of the data vary significantly. Be careful not to fall into the trap where you feel you must choose one tool for *all of your data*. For example, it often makes sense to keep rich analytical data in a Hadoop cluster or business-related data in an Excel spreadsheet. From my experience, the questions you need to ask of your data should influence the tools you use to store them, and not the other way around.

Regardless, there will eventually come a time when you need to correlate your IT and business metrics. Having a unified source of truth in a service like Graphite will make quick work of those questions that would often be difficult and time-intensive to answer otherwise. Powering your alert responses and decision-making with time-series data gives you a level of intelligence and perspective that simply isn't possible with "ping" checks.

But please keep in mind that the discussion of pull-vs-push models is completely orthogonal to the need for proper time-series collection and storage. No matter what your monitoring systems resemble, you should *absolutely* have a data visualization service like Graphite at your disposal, for all the reasons I've laid out previously. I feel strongly about designing your monitoring systems properly, and that's why I've dedicated almost half of this chapter to a discussion about the topic. Tough love.

In the next section we'll describe a modern monitoring architecture based on the push model. I'm going to explain each functional area and try to give practical examples for each. But more importantly, we'll discuss how Graphite relates to each in order to give you a better understanding of its versatility and how vital a proper time-series engine is to each layer of the monitoring stack.

Composable Monitoring Systems

If you've been in the IT industry for long, there's a very good chance that you've had to procure or evaluate a commercial vendor's software product. Their glossy brochure or website almost certainly checked off all the requirements on your specification and promised a one-stop comphrehensive "solution". The overly-attached sales rep hounded you incessantly, and you finally relented.

Inevitably, your productivity took a hit because one or more features wasn't quite the right fit, didn't work as described, or worse yet, didn't exist at all (despite claims to the contrary). If you're lucky, you got a direct response from your rep (who's no longer nearly as needy, for some reason) who reached out to their engineering team and got a roadmap estimate for sometime in the next 6-12 months. If you *weren't* lucky, you got stuck in a confusing loop of Knowledge Base articles and community forums.

Weeks turned into months, months grinding into years, and before long you realize that you're fortunate because someone else took over your old role while you transitioned into a nice dark cubicle with a red stapler and a job managing the nightly tape backups.

Now, consider the alternative - an ecosystem of utilities and services that work as building blocks, allowing you to assemble a monitoring architecture that's custom-tailored to the specific needs of your organization. Components can more easily be upgraded, replaced, or enhanced without the lock-in usually associated with single-source vendors or monolithic software products. In many cases, you can run competing "blocks" in parallel for evaluation with the help of load-balancing software and related bridge services.

In short, a Composable Monitoring System offers you a level of *flexibility* that simply can't be matched by any other approach.

Even if you have no desire to deploy and manage a multi-faceted composable monitoring system, it helps to understand the functional areas involved. Most existing monitoring software products are built on these discrete components, even if the user interface completely abstracts the underlying gook from the user.

The components I'm about to introduce are based on well-established patterns common to both Open Source projects and commercial services. Having a solid grasp of these concepts will help you to better understand how all of these different products work and makes you a more educated "consumer".

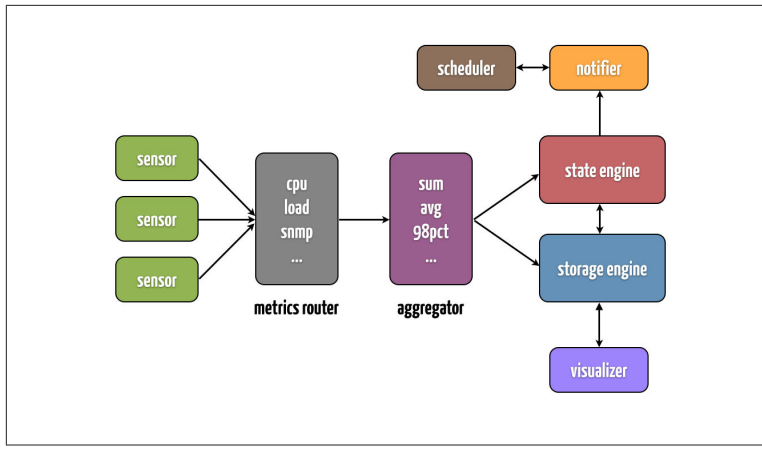


Figure 3-2. Composable Monitoring System

The diagram in Figure 3-2 demonstrates a typical workflow for data and interactions within a modular monitoring system. Metrics generally flow in a unidirectional pattern, from collection of sensor or emitter data (i.e. *telemetry*) to *aggregation*, into a *state engine* for tracking state and thresholds, and eventually to a *storage engine* for long-term archival and retrieval. Along the way we may fire off *notifications*, and users are almost certainly going to use *visualization* for troubleshooting and correlation activities.

Your environment may dictate a different topology or information flow, and that's fine. Your monitoring system is much more likely to be influenced by your existing IT systems design (as it should be) than my incoherent ramblings. Although the monitoring principles I espouse here are heavily influenced by real-life production architectures at scale, it's entirely possible that your environment and circumstances are sufficiently different as to render my suggestions moot. In this case I urge you to listen to your own experiences but study the functional separation of these systems, because the logical components I've described here are *universal*.

Telemetry

> Look, do you wanna play blind man? Go walk with the shepherd. But me, my eyes are wide fucking open.
>
> —Jules Winnfield

Collection Agents and Sensors

For operating systems, network services, and even hardware devices, it's still common to see many companies using SNMP to collect and expose (or even extend, using custom shell scripts) their host-level telemetry. Because they're able to be emitted or polled at regular intervals, it's possible to send them directly to the storage system, bypassing any sort of aggregation or normalization. Unfortunately, SNMP has a negative connotation for many users due to it's archaic configuration, confusing usage, and questionable security record.

NOTE

SNMP is not going anywhere soon, at least with regards to network devices. Manufacturers are starting to offer the ability to export metrics, typically using packet sampling protocols like NetFlow or sFlow. Projects like Evenflow (*https://github.com/obfuscurity/evenflow*) provide a bridge service for accepting data from these types of exports into your metrics stream.

Personally, I'm a big fan of the collectd project (*http://collectd.org/*). It's a highly extensible collection agent that works well on all Linux distributions and BSD systems. Installation and configuration is easy and the default setup provides a great out-of-the-box experience. Written in C, it performs significantly better and more reliably than competing collectors. Best of all, it has a huge collection of supported input and output plugins, including support for sending metrics directly to Graphite's Carbon listener.

Diamond (*https://github.com/BrightcoveOS/Diamond*) is another popular collection agent among the Graphite user community. Because it's written in Python, many Graphite (another Python app) developers and users find it a better fit for writing new plugins from scratch. Regardless of the host-level collection agent you select, it's important that you use one that has a solid breadth of coverage for your particular environment.

Application Instrumentation

Developers, QA engineers and even Operations teams rely on software instrumentation to report telemetry data for tracking performance and identifying bugs in the code they release to production. Without these measurements, there's no simple method for accurately tracing a problem back to its source, or correlating causal relationships between systems or services. Proper instrumentation can

quickly isolate regressions between software versions or deployments.

StatsD (*https://github.com/etsy/statsd/*) is a very popular aggregation daemon created by engineers at Etsy to aggregate and normalize metrics before forwarding them to a time-series backend like Graphite. We'll touch more about it's service-side benefits in the Aggregation section, but it's an important technology to acknowledge in terms of application instrumentation. StatsD libraries are plentiful, offering language bindings for virtually every modern high-level programming language in use on the Web today.

Using a StatsD client library, developers can fire off new measurements with a single line of code. It supports *timers*, *counters*, and *gauges* natively, making it trivial to track durations, cumulative sums, or rates over time. And perhaps best of all, StatsD is designed to use the non-blocking and stateless UDP protocol by default. This means that clients will never have to wait on a StatsD request to complete; it doesn't *block* progress of your code because of e.g. a remote connection timeout or processing request. This is a significant consideration for developers wanting to collect performance data on their applications at "Web Scale".

Example 3-1.

```
StatsD::increment("widgets.sold");
```

Logging

It's worth mentioning that logging is a completely valid alternative to dedicated instrumentation libraries. Many companies already have a robust logging pipeline available; this diminishes the need for a dedicated aggregation service like StatsD or retooling your applications with the libraries needed to speak to it.

In fact, the entire Heroku platform emits telemetry data using a standard log format, leveraging their high-performance distributed log routing service. Although the logs inevitably get stored in a high-volume log archival service further down the line, they "drain" metrics from this "log stream", applying aggregation rules and forwarding it on to their time-series storage system.

If you can print your metrics to `STDOUT` in a predictable and structured format, there are a variety of logging tools that will happily

parse your entries and convert them into Graphite-compatible metrics. As we'll see later in the *Event Stream* and *Aggregation* sections, a number of companies use this approach to leverage their existing logging infrastructure.

Example 3-2.

```
measure#cache.get=4ms measure.db.get=50ms measure#cache.put=4ms
```

Business Telemetry

I frequently get asked by colleagues which metrics they should track in Graphite or related systems. Among the usual suspects of host-level and application-level metrics, I strongly encourage everyone to track their Key Performance Indictators (KPI). Unfortunately, there really is no universal KPI that I can blindly recommend; they're going to be unique to your business or organization.

If you're an online retailer, it's a no-brainer to track metrics related to sales and shipments. These reflect the current health of an online business, but they're not a useful predictor of future trends. Instead, think about the relationships and experiences that may cause your business to falter. Are your customers happy? How can you tell? If you're using an online ticketing service, they almost certainly offer an API where you can track your customer support levels and convert them into time-series data for Graphite. This sort of KPI may one day help you explain a downturn in customer retention levels or even new signups.

Business telemetry is not only useful for predicting trends. Often, we want to be able to measure the effects of a service outage in real dollars. Without metrics that describe the health of your sales pipeline or revenue numbers, there's no *quick* method for correlating these events. The engineer that manages to capture this sort of data and present it on an executive dashboard is going to be viewed in a *very* positive light.

Metrics Router

Depending on the size of your company or the complexity of your production architecture (or lack thereof), you may never find yourself needing anything resembling a metrics "Event Stream". This particular pattern is more common in large distributed systems with a

tremendous volume of telemetry data and the need to support a variety of emitter and consumer interface formats.

It helps to think of this component as something akin to a metrics transport or router. In many cases, the need to publish a measurement and have multiple consumers pull copies of the same data is a primary driver for these services. Apache's Kafka (*http://kafka.apache.org/*), a "high-throughput distributed messaging system", is hugely popular for producing centralized feeds of telemetry data. Projects like Heroku's Logplex log router (*https://github.com/heroku/logplex*) and Mozilla's Heka stream processing service (*http://hekad.readthedocs.org/*) have different approaches but arguably address the same problem set.

Aggregation

There will be times when it's necessary to perform aggregation on your metrics stream. For example, application telemetry tends to be irregular, firing whenever an event or action occurs. This doesn't jibe well with the sequential and predictable nature of time-series data. If our Whisper policy is designed for 10-second resolution, attempting to write a few increments in one interval and no metrics at all in the next interval is going to result in a loss of data in the former and gaps in the latter. Using aggregation, we can normalize our metrics: all of the data in the first interval should be summed up before transmitting, while the second interval should be reported as a zero (rather than nothing at all, resulting in a *null* value).

And although Graphite's render API provides all of the transformative functions you'll probably ever require, asking it to perform a lot of recalculations on the fly for numerous clients can result in significant CPU overhead in the Graphite web application. Many users prefer to apply statistical transforms in the aggregation layer (e.g. summing two metrics to create a new unique metric) before passing it on to the storage layer. Graphite's **carbon-aggregator** is an example of this type of service, and we'll cover it at greater length in the next chapter.

We've already touched on StatsD in terms of the library support it offers developers for easy instrumentation of their applications. But one of my favorite features is its ability to export "value-added" statistics for *timer* measurements. Not only will it tell you the average (*mean*) for all timers in an interval, it will also give you: the *upper*

and *lower* boundaries, the total *count* of timers reported, the *timer_ps* of timers per second, the *sum* and *sum_squares*, and the *median* and *std* standard deviation. It's a veritable smorgasbord of statstical delights.

Last but not least, anyone with a reliance on logging data will want to look at aggregators like Logstash (*http://logstash.net/*) to parse the source logs and convert them into time-series data. Logstash even has support for StatsD output, or you can configure it to fire your metrics directly to Graphite's **carbon-cache**.

State Engine

In many ways, the state engine acts as the brains of a monitoring system. It may perform additional services, but it should at least: track the state of metrics according to their current and recent measurements in relation to defined thresholds (e.g. upper and lower boundaries), open alert incidents when a threshold condition is exceeded, close or resolve alert incidents when a threshold has recovered, and trigger alert or recovery notifications.

Despite all its warts, Nagios is an effective state engine. I have my own personal biases against the way it handles acknowledgements and flapping, but at its core it does a solid job adhering to the basic requirements for this component. Riemann (*http://riemann.io/*) and Sensu (*http://sensuapp.org/*) are both compelling alternatives to Nagios here, designed with automation in mind and the goal of streaming all check results (read: metrics) to a time-series storage engine like Graphite.

There are other Open Source alternatives that are more strictly aligned with the Composable Monitoring System design. Heroku's Umpire (*https://github.com/heroku/umpire*) is one of the more unique designs, specifically built to accept Graphite query definitions and thresholds. It then analyzes the Graphite response, determines whether the response falls within the acceptable threshold boundaries, and finally returns an HTTP status code 200 (success), 500 (out of range), or 404 (no data). This is a particularly ingenious design; with the combination of Umpire and any website monitoring service (e.g. Pingdom), it allows developers to build their own monitoring checks without needing to commit configuration

changes to the monitoring system or burdening other IT teams with routine requests.

Self Service Monitoring

Umpire adheres to the notion of *Self Service Monitoring*. This is an approach to monitoring that aims to remove the traditional hurdles of deploying monitoring system configuration changes. There are certainly parallels to be made to concepts introduced by the *DevOps* movement, but the notion of Self Service Monitoring focuses squarely on the design of our tools in order to remove telemetry deployment friction for engineering teams.

Notification Routers

We've already discussed Alerting earlier in this chapter, so you should already be familiar with the basic concepts here. Notification engines are typically responsible for routing alerts to their desired destination, which means they should be capable of supporting a variety of output transports and protocols: Email, SMS, Webhooks or even other third-party "incident management" services.

Scheduling and escalation management is generally a difficult nut to crack, so there are unfortunately very few Open Source projects that address this problem directly. Companies like PagerDuty, OpsGenie and VictorOps do a very good job solving this concern, and in my opinion, are well worth the investment.

Storage Engine

Storage engines are responsible for long-term storage and retrieval of metrics. Due to the nature of time-series data, they need to support a heavy write load, but be suited for near-realtime retrieval. They should include transformative functions (standard arithmetic and statistical primitives) and support a variety of output formats: JSON, CSV, SVG and some manner of raw data should be considered standard fare these days.

Above all, they should be capable of persisting data to disk for long-term trending. It's not sufficient to store metrics in memory for faster response and then discard or allow them to "fall off" after a predetermined interval (that's cheating!); we should be able to trust

our storage engine to actually write our data to archive storage in a reliable manner.

If you've been paying attention so far (good on ya, mate), you'll probably find it as no great surprise that I recommend using Graphite as your storage engine. Thanks to its powerful API and scalability-conscious design, it's a natural fit as the *source of truth* for any monitoring architecture. Newer versions of Graphite include support for *pluggable storage backends* (such as Whisper), allowing users to choose the storage tradeoffs that matter for their particular use case.

Visualization

For many Graphite users, it might seem unusual to categorize Graphite as a "Storage Engine" but not necessarily as a visualization tool. It's always included support for PNG image output, so why not?

The world of data visualization has moved steadily away from server-side static images, adopting modern frameworks like D3.js (*http://d3js.org/*), built on standard technologies that incorporate Javascript, HTML and CSS. These frameworks support a much better interactive experience and offer a huge variety of dynamic new chart types that even systems like Graphite can't match. This separation of responsibility means that we now have an entire community of web designers and developers participating in the monitoring and visualization communities. If you've ever seen a Systems Administrator design a website, you understand this means a huge advance in usability (wink).

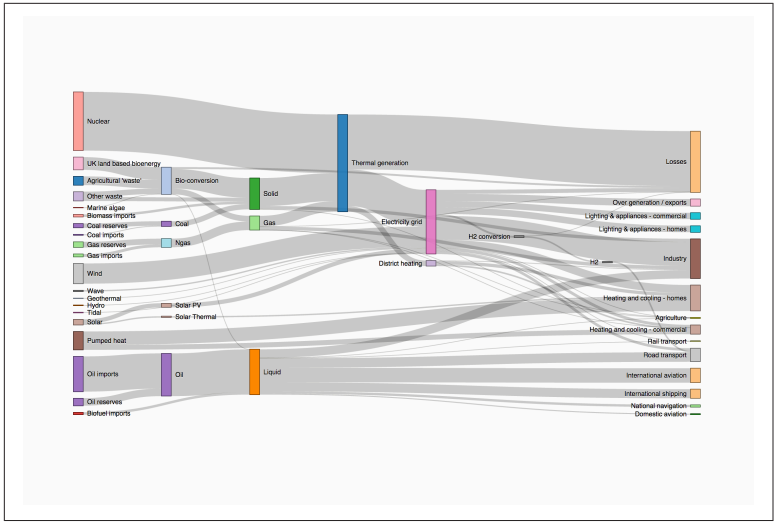


Figure 3-3. Sankey Chart

Dashboards in particular have benefitted from these changes. An enormous ecosystem of dashboard projects and commercial dashboard services now exist, able to consume storage engine APIs and generate useful interfaces for a variety of workflows and organizations. We'll cover some of these dashboards, as well as the client-side development approach, in Chapter 7.

As an aside, there are server-side benefits to decoupling image generation and the storage engine. Data requests from the client browser to the backend storage engine can take advantage of text formats like JSON, allowing for increased compression in transit, resulting in faster data connections and lower bandwidth requirements. Coincidentally, I'm proud to have been ahead of the curve here. I submitted the original patch for adding JSON format support to Graphite back in 2011. Fortunately for us all, the JSON formatting code that exists now bears no resemblence to my original patch.

Conclusion

As you can see, there are a *ton* of functional responsibilities for a modern monitoring system. Assembling yours (or even just buying one) will require a lot of considerations and almost certainly some compromises. I encourage you to take some time to analyze your current strengths and weaknesses, and to determine which gaps are

important for you to address *right now*. For most companies I talk with, time-series collection and visualization is a vital operational need for them, which is also probably why you're reading this book.

Now that you have a solid foundation of monitoring-related vocabulary and conventions, I feel confident that we can dive head-first into learning Graphite without too many distractions. The next chapter is rich with tips and tidbits that will help you get the most out of Graphite as it grows with your organization. There's a lot to absorb, but this knowledge should prepare you for many of the little surprises that trip up most new users (and even some experienced ones).

THE **LEAN** SERIES ERIC RIES, SERIES EDITOR

Jez Humble, Joanne Molesky & Barry O'Reilly

LEAN ENTERPRISE

How High Performance Organizations Innovate at Scale

O'REILLY®

Deploy Continuous Improvement

The following content is excerpted from *Lean Enterprise*, by Jez Humble, Joanne Molesky, and Barry O'Reilly. Availble now.

> The paradox is that when managers focus on productivity, long-term improvements are rarely made. On the other hand, when managers focus on quality, productivity improves continuously.
>
> —John Seddon

In most enterprises, there is a distinction between the people who build and run software systems (often referred to as "IT") and those who decide what the software should do and make the investment decisions (often called "the business"). These names are relics of a bygone age in which IT was considered a cost necessary to improve efficiencies of the business, not a creator of value for external customers by building products and services. These names and the functional separation have stuck in many organizations (as has the relationship between them, and the mindset that often goes with the relationship). Ultimately, we aim to remove this distinction. In high-performance organizations today, people who design, build, and run software-based products are an integral part of business; they are given—and accept—responsibility for customer outcomes. But getting to this state is hard, and it's all too easy to slip back into the old ways of doing things.

Achieving high performance in organizations that treat software as a strategic advantage relies on *alignment* between the IT function and the rest of the organization, along with the ability of IT to *execute*. It

pays off. In a report for the *MIT Sloan Management Review*, "Avoiding the Alignment Trap in Information Technology," the authors surveyed 452 companies and discovered that high performers (7% of the total) spent a little less than average on IT while achieving substantially higher rates of revenue growth.[1]

However, *how* you move from low performance to high performance matters. Companies with poor alignment and ineffective IT have a choice. Should they pursue alignment first, or try to improve their ability to execute? The data shows that companies whose IT capabilities were poor achieve worse results when they pursue alignment with business priorities before execution, even when they put significant additional investment into aligned work. In contrast, companies whose engineering teams do a good job of delivering their work on schedule and simplifying their systems achieve better business results with much lower cost bases, even if their IT investments aren't aligned with business priorities.

The researchers concluded that to achieve high performance, companies that rely on software should focus first and foremost on their ability to execute, build reliable systems, and work to continually reduce complexity. Only then will pursuing alignment with business priorities pay off.

However, in every team we are always balancing the work we do to improve our capability against delivery work that provides value to customers. In order to do this effectively, it's essential to manage both kinds of work at the program and value stream levels. In this chapter we describe how to achieve this by putting in place a framework called *Improvement Kata*. This is the first step we must take to drive continuous improvement in our execution of large scale programs. Once we have achieved this, we can use the tools in the following chapters to identify and remove no-value-add activity in our product development process.

1 Schpilberg, D., Berez, S., Puryear, R., and Shah, S. (2007). "Avoiding the Alignment Trap in Information Technology." *MIT Sloan Management Review* Fall 2007.

The HP LaserJet Firmware Case Study

We will begin with a case study from the HP LaserJet Firmware team, which faced a problem with both alignment and execution.[2] As the name suggests, this was a team working on embedded software, whose customers have no desire to receive software updates frequently. However, it provides an excellent example of how the principles described in the rest of Part III work at scale in a distributed team, as well as of the economic benefits of adopting them.

HP's LaserJet Firmware division builds the firmware that runs all their scanners, printers, and multifunction devices. The team consists of 400 people distributed across the USA, Brazil, and India. In 2008, the division had a problem: they were moving too slowly. They had been on the critical path for all new product releases for years, and were unable to deliver new features: "Marketing would come to us with a million ideas that would dazzle the customer, and we'd just tell them, 'Out of your list, pick the two things you'd like to get in the next 6–12 months.'" They had tried spending, hiring, and outsourcing their way out of the problem, but nothing had worked. They needed a fresh approach.

Their first step was to understand their problem in greater depth. They approached this by using *activity accounting*—allocating costs to the activities the team is performing. Table 4-1 shows what they discovered.

Table 4-1. Activities of the HP LaserJet Firmware team in 2008

% of costs	Activity
10%	Code integration
20%	Detailed planning
25%	Porting code between version control branches
25%	Product support

2 This case study is taken from Gruver, G. (2012). *A Practical Approach to Large-Scale Agile Development: How HP Transformed LaserJet FutureSmart Firmware*. Addison-Wesley., supplemented by numerous discussions with Gary Gruver.

% of costs	Activity
15%	Manual testing
~5%	Innovation

This revealed a great deal of no-value-add activity in their work, such as porting code between branches and detailed upfront planning. The large amount spent on current product support also indicated a problem with the quality of the software being produced. Money spent on support is generally serving *failure demand*, as distinct from *value demand*, which was only driving 5% of the team's costs.[3]

The team had a goal of increasing the proportion of spending on innovation by a factor of 10. In order to achieve that goal, they took the bold but risky decision to build a new firmware platform from scratch. There were two main architectural goals for the new "FutureSmart" platform. The first goal was to increase quality while reducing the amount of manual testing required for new firmware releases (a full manual testing cycle required six weeks). The team hoped that this goal could be achieved through:

- The practice of continuous integration (which we describe in Chapter 8)
- Significant investment in test automation
- Creating a hardware simulator so that tests could be run on a virtual platform
- Reproduction of test failures on developer workstations

Three years into the development of the new firmware, thousands of automated tests had been created.

Second, they wanted to remove the need for the team to spend time porting code between branches (25% of total costs on the existing system). This was caused by the need to create a branch—effectively

3 The distinction between failure demand and value demand comes from John Seddon, who noticed that when banks outsourced their customer service to call centers, the volume of calls rose enormously. He showed that up to 80% of the calls were "failure demand" of people calling back because their problems were not solved correctly the first time (Seddon, J. (1992). *I Want You to Cheat!: The Unreasonable Guide to Service and Quality in Organisations.* Vanguard Consulting.).

a copy of the entire codebase—for every new line of devices under development. If a feature or bug-fix added to one line of devices was required for any others, these changes would need to be merged (copied back) into the relevant code branches for the target devices, as shown in Figure 4-1. Moving away from branch-based development to trunk-based development was also necessary to implement continuous integration. Thus the team decided to create a single, modular platform that could run on any device, removing the need to use version control branches to handle the differences between devices.

The final goal of the team was to reduce the amount of time its members spent on detailed planning activities. The divisions responsible for marketing the various product lines had insisted on detailed planning because they simply could not trust the firmware team to deliver. Much of this time was spent performing detailed re-plans after failing to meet the original plans.

Furthermore, the team did not know how to implement the new architecture, and had not used trunk-based development or continuous integration at scale before. They also understood that test automation would require a great deal of investment. How would they move forward?

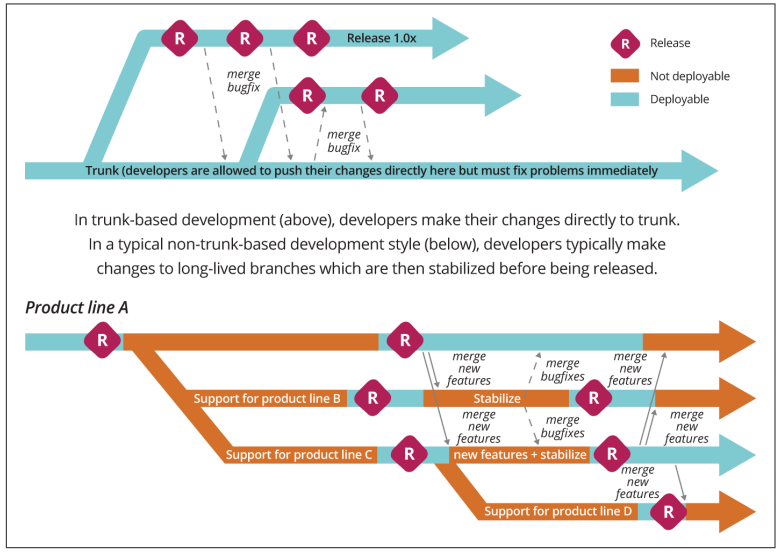


Figure 4-1. Branching versus trunk-based development

It's all too easy to turn a sequence of events into a story in an attempt to explain the outcome—a cognitive bias that Nassim Taleb terms the *narrative fallacy*. This is, arguably, how methodologies are born. What struck us when studying the FutureSmart case were the similarities between the program management method of FutureSmart's engineering management team and the approach Toyota uses to manage innovation as described in Mike Rother's *Toyota Kata: Managing People for Improvement, Adaptiveness, and Superior Results*.[4]

Drive Down Costs Through Continuous Process Innovation Using the Improvement Kata

The Improvement Kata, as described by Mike Rother, is a general-purpose framework and a set of practice routines for reaching goals where the path to the goal is uncertain. It requires us to proceed by iterative, incremental steps, using very rapid cycles of experimentation. Following the Improvement Kata also increases the capabilities and skills of the people doing the work, because it requires them to solve their own problems through a process of continuous experimentation, thus forming an integral part of any learning organization. Finally, it drives down costs through identifying and eliminating waste in our processes.

The Improvement Kata needs to be first adopted by the organization's management, because it is a management philosophy that focuses on developing the capabilities of those they manage, as well as on enabling the organization to move towards its goals under conditions of uncertainty. Eventually, everybody in the organization should be practicing the Improvement Kata habitually to achieve goals and meet challenges. This is what creates a culture of continuous improvement, experimentation, and innovation.

To understand how this works, let's examine the concept of *kata* first. A kata is "a routine you practice deliberately, so its pattern becomes a habit."[5] Think of practicing scales to develop muscle memory and digital dexterity when learning the piano, or practicing the basic patterns of movement when learning a martial art (from which the term derives), or a sport. We want to make continuous

4 Rother, M. (2010). *Toyota Kata: Managing People for Improvement, Adaptiveness, and Superior Results.* McGraw-Hill.

5 Rother, M. (2014).

improvement a habit, so that when faced with an environment in which the path to our goal is uncertain, we have an instinctive, unconscious routine to guide our behavior.

In Toyota, one of the main tasks of managers is to teach the Improvement Kata pattern to their teams and to facilitate running it (including coaching learners) as part of everyday work. This equips teams with a method to solve their own problems. The beauty of this approach is that if the goal or our organization's environment changes, we don't need to change the way we work—if everybody is practicing the Improvement Kata, the organization will automatically adapt to the new conditions.

The Improvement Kata has four stages that we repeat in a cycle, as shown in Figure 4-2.

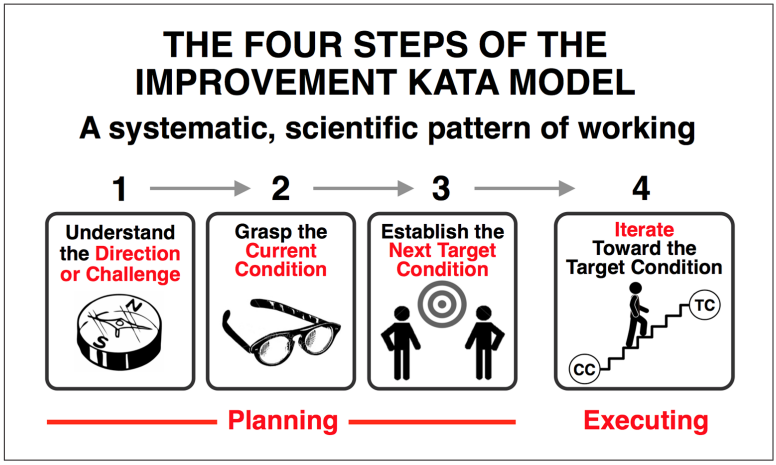


Figure 4-2. The Improvement Kata, courtesy of Mike Rother

Understand the Direction

We begin with understanding the direction. Direction is derived from the vision set by the organization's leadership. A good vision is one that is inspiring—and, potentially, unattainable in practice. For example, the long-term vision for Toyota's production operations is "One-piece flow at lowest possible cost." In *Leading Lean Software Development*, Mary and Tom Poppendieck describe Paul O'Neill set-

ting the objective for Alcoa to be "Perfect safety for all people who have anything to do with Alcoa" when he became CEO in 1987.[6]

People need to understand that they must always be working towards the vision and that they will never be done with improvement. We will encounter problems as we move towards the vision. The trick is to treat them as obstacles to be removed through experimentation, rather than objections to experimentation and change.

Based on our vision and following the Principle of Mission, we must understand the direction we are working in, at the level of the whole organization and at the value stream level. This challenge could be represented in the form of a future-state value stream map (see Chapter 7 for more on value stream mapping). It should result in a measurable outcome for our customers, and we should plan to achieve it in six months to three years.

Planning: Grasp the Current Condition and Establish a Target Condition

After we have understood the direction at the organizational and value stream levels, we incrementally and iteratively move towards it at the process level. Rother recommends setting target conditions with a horizon between one week and three months out, with a preference for shorter horizons for beginners. For teams that are using iterative, incremental methods to perform product development, it makes sense to use the same iteration (or sprint) boundaries for both product development and Improvement Kata iterations. Teams that use flow-based methods such as the Kanban Method (for which see Chapter 7) and continuous delivery (described in Chapter 8) can create Improvement Kata iterations at the program level.

As with all iterative product development methods, Improvement Kata iterations involve a planning part and an execution part. Here, planning involves grasping the current condition at the process level and setting a target condition that we aim to achieve by the end of the next iteration.

Analyzing the current condition "is done to obtain the facts and data you need in order to then describe an appropriate next target condition. What you're doing is trying to find the current pattern of oper-

6 Poppendieck, M. and Poppendieck, T. (2009). *Leading Lean Software Development: Results Are Not the Point*. Addison-Wesley., Frame 13, "Visualize Perfection."

ation, so you can establish a desired pattern of operation (a target condition)." The target condition "describes in measurable detail how you want a process to function…[It is] a description and specification of the operating pattern you want a process or system to have on a future date."[7]

The team grasps the current condition and establishes a target condition together. However, in the planning phase the team does not plan how to *move* to the target condition. In the Improvement Kata, people doing the work strive to achieve the target condition by performing a series of experiments, not by following a plan.

A target condition identifies the process being addressed, sets the date by which we aim to achieve the specified condition, and specifies measurable details of the process as we want it to exist. Examples of target conditions include WIP (work in progress) limits, the implementation of Kanban or a continuous integration process, the number of good builds we expect to get per day, and so forth.

Getting to the Target Condition

Since we are engaging in process innovation in conditions of uncertainty, we cannot know in advance how we will achieve the target condition. It's up to the people doing the work to run a series of experiments using the Deming cycle (plan, do, check, act), as described in Chapter 3. The main mistakes people make when following the Deming cycle are performing it too infrequently and taking too long to complete a cycle. With Improvement Kata, everybody should be running experiments on a daily basis.

Each day, people in the team go through answering the following five questions:[8]

1. What is the target condition?
2. What is the actual condition now?
3. What obstacles do you think are preventing you from reaching the target condition? Which one are you addressing now?
4. What is your next step? (Start of PDCA cycle.) What do you expect?

7 Rother, M. (2014). *Improvement Kata Handbook.* Available from *http://bit.ly/11iBzlY*.

8 Rother, M. (2014). *Improvement Kata Handbook.* Available from *http://bit.ly/11iBzlY*.

5. When can we go and see what we learned from taking that step?

As we continuously repeat the cycle, we reflect on the last step taken to introduce improvement. What did we expect? What actually happened? What did we learn? We might work on the same obstacle for several days.

This experimental approach is already central to how engineers and designers work. Designers who create and test prototypes to reduce the time taken by a user to complete a task are engaged in exactly this process. For software developers using test-driven development, every line of production code they write is essentially part of an experiment to try and make a unit test pass. This, in turn, is a step on the way to improving the value provided by a program—which can be specified in the form of a target condition, as we describe in Chapter 9.

The Improvement Kata is simply a generalization of this approach to improvement, combined with applying it at multiple levels of the organization, as we discuss when presenting strategy deployment in Chapter 15.

How the Improvement Kata Differs from Other Methodologies

You can think of the Improvement Kata as a *meta-methodology* since it does not apply to any particular domain, nor does it tell you what to do. It is not a playbook; rather, as with the Kanban Method, it teaches teams how to *evolve* their existing playbook. In this sense, it differs from other agile frameworks and methodologies. With the Improvement Kata, there is no need to make existing processes conform to those specified in the framework; process and practices you use are *expected* to evolve over time. *This* is the essence of agile: teams do not become agile by adopting a methodology. Rather, true agility means that teams are constantly working to evolve their processes to deal with the particular obstacles they are facing at any given time.

NOTE

Single-Loop Learning and Double-Loop Learning

Changing the way we think and behave in reaction to a failure is crucial to effective learning. This is what distinguishes *single-loop learning* from *double-loop learning* (see Figure 4-3). These terms were coined by management theorist Chris Argyris, who summarizes them as follows: "When the error detected and corrected permits the organization to carry on its present policies or achieve its present objectives, then that error-and-correction process is single-loop learning. Single-loop learning is like a thermostat that learns when it is too hot or too cold and turns the heat on or off. The thermostat can perform this task because it can receive information (the temperature of the room) and take corrective action. Double-loop learning occurs when error is detected and corrected in ways that involve the modification of an organization's underlying norms, policies and objectives."[9] Argyris argues that the main barrier to double-loop learning is defensiveness when confronted with evidence that we need to change our thinking, which can operate at both individual and organizational levels. We discuss how to overcome this anxiety and defensiveness in Chapter 11.

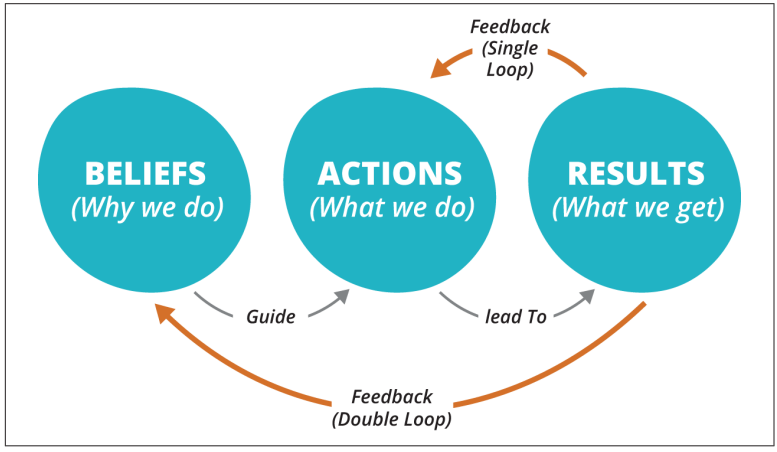


Figure 4-3. Single-loop and double-loop learning

9 Argyris, C. and Schön, D. (1978). *Organizational Learning: A Theory of Action Perspective.* Addison Wesley., pp. 2–3.

When you practice the Improvement Kata, process improvement becomes planned work, similar to building product increments. The key is that we don't plan *how* we will achieve the target condition, nor do we create epics, features, stories, or tasks. Rather, the team works this out through experimentation over the course of an iteration.

Deploying the Improvement Kata

Rother's work on the Improvement Kata was a direct result of his enquiry into how people become managers at Toyota. There is no formal training program, nor is there any explicit instruction. However, to become a manager at Toyota, one must have first worked on the shop floor and therefore participated in the Improvement Kata. Through this process, managers receive implicit training in how to manage at Toyota.

This presents a problem for people who want to learn to manage in this way or adopt the Improvement Kata pattern. It is also a problem for Toyota—which is aiming to scale faster than is possible through what is effectively an apprenticeship model for managers.

Consequently, Rother presents the Coaching Kata in addition to the Improvement Kata. It is part of deploying the Improvement Kata, but it is also as a way to grow people capable of working with the Improvement Kata, including managers.

Rother has made a guide to deploying the Improvement Kata, *The Improvement Kata Handbook*, available for free on his website at *http://bit.ly/11iBzlY*.

How the HP LaserJet Team Implemented the Improvement Kata

The direction set by the HP LaserJet leadership was to improve developer productivity by a factor of 10, so as to get firmware off the critical path for product development and reduce costs.[10] They had three high-level goals:

1. Create a single platform to support all devices

10 Gruver, G. (2012). *A Practical Approach to Large-Scale Agile Development: How HP Transformed LaserJet FutureSmart Firmware.* Addison-Wesley., p. 144.

2. Increase quality and reduce the amount of stabilization required prior to release
3. Reduce the amount of time spent on planning

They did not know the details of the path to these goals and didn't try to define it. The key decision was to work in iterations, and set target conditions for the end of each four-week iteration. The target conditions for Iteration 30 (about 2.5 years into the development of the FutureSmart platform) are shown in Figure 4-4.

The first thing to observe is that the target conditions (or "exit criteria" as they are known in FutureSmart) are all measurable conditions. Indeed, they fulfill all the elements of SMART objectives: they are specific, measurable, achievable, relevant, and time bound (the latter by virtue of the iterative process). Furthermore, many of the target conditions were not focused on features to be delivered but on attributes of the system, such as quality, and on activities designed to validate these attributes, such as automated tests. Finally, the objectives for the entire 400-person distributed program for a single month was captured in a concise form that fit on a single piece of paper—similar to the standard A3 method used in the Toyota Production System.

How are the target conditions chosen? They are "aggressive goals the team feels are possible and important to achieve in 4 weeks…We typically drive hard for these stretch goals but usually end up hitting around 80% of what we thought we could at the beginning of the month."[11] Often, target conditions would be changed or even dropped if the team found that the attempt to achieve them results in unintended consequences: "It's surprising what you learn in a month and have to adjust based on discovery in development."[12]

11 Gruver, G. (2012). *A Practical Approach to Large-Scale Agile Development: How HP Transformed LaserJet FutureSmart Firmware.* Addison-Wesley., p. 40.

12 Ibid.

Rank	Theme	Exit Criteria Objective met / *Objective not met*
0	Quality threshold	P1 issues open < 1 week L2 test failure 24 hour response
1	Quarterly bit release	*A) Final P1 change requests fixed* B) Reliability error rate at release criteria
2	New platform stability and test coverage	A) Customer Acceptance Test 100% passing B) All L2 test pillars 98% passing C) L4 test pillars in place D) L4 test coverage for all Product Turn On requirements E) 100% execution of L4 tests on new products
3	Product Turn On dependencies and key features	A) Print for an hour at speed to finisher with stapling B) Copy for an hour *at speed* *C) Enable powersave mode* D) Manufacturing nightly test suite execution E) Common Test Library support for four-line control panel display
4	Build for next-gen products	*A) End-to-end system build on new processor* *B) High-level performance analysis on new processor*
5	Fleet integration plan	Align on content and schedule for "slivers" of end-to-end agile test with system test lab

Figure 4-4. Target conditions for iteration 30[13]

13 Gruver, Gary, Young, Mike, Fulghum, Pat. *A Practical Approach to Large-Scale Agile Development: How HP Transformed LaserJet FutureSmart Firmware*, 1st Edition, (c) 2013. Reprinted by permission of Pearson Education, Inc. Upper Saddle River, NJ.

What Happens When We Do Not Achieve Our Target Conditions?

In bureaucratic or pathological organizational cultures, not achieving 100% of the specified target conditions is typically considered a failure. In a generative culture, however, we *expect* to not be able to achieve all our target conditions. The purpose of setting aggressive target conditions is to reveal obstacles so we can overcome them through further improvement work. Every iteration should end with a retrospective (described in Chapter 11) in which we investigate how we can get better. The results form part of the input for the next iteration's target conditions. For example, if we fail to achieve a target condition for the number of good builds of the system per day, we may find that the problem is that it takes too long to provision test environments. We may then set a target condition to reduce this in the next iteration.

This approach is a common thread running through all of Lean Thinking. The subtitle of Mary and Tom Poppendieck's book *Leading Lean Software Development* reads: "Results are not the point." This is a provocative statement that gets to the heart of the lean mindset. If we achieve the results by ignoring the process, we do not learn how to improve the process. If we do not improve the process, we cannot repeatably achieve better results. Organizations that put in place unmodifiable processes that everybody is required to follow, but which get bypassed in a crisis situation, fail on both counts.

This adaptive, iterative approach is not new. Indeed it has a great deal in common with what Tom Gilb proposed in his 1988 work *Principles of Software Engineering Management*:[14]

> We must set measurable objectives for each next small delivery step. Even these are subject to constant modification as we learn about reality. It is simply not possible to set an ambitious set of multiple quality, resource, and functional objectives, and be sure of meeting them all as planned. We must be prepared for compromise and trade-off. We must then design (engineer) the immediate technical solution, build it, test it, deliver it—

14 Gilb, T. (1988). *Principles of Software Engineering Management.* Addison-Wesley., p. 91.

and get feedback. This feedback must be used to modify the immediate design (if necessary), modify the major architectural ideas (if necessary), and modify both the short-term and the long-term objectives (if necessary).

Designing for Iterative Development

In large programs, demonstrating improvement within an iteration requires ingenuity and discipline. It's common to feel we can't possibly show significant progress within 2–4 weeks. Always try to find something small to bite off to achieve a little bit of improvement, instead of trying to do something you think will have more impact but will take longer.

This is not a new idea, of course. Great teams have been working this way for decades. One high-profile example is the Apple Macintosh project where a team of about 100 people—co-located in a single building—designed the hardware, operating system, and applications for what was to become Apple's breakthrough product.

The teams would frequently integrate hardware, operating system, and software to show progress. The hardware designer, Burrell Smith, employed programmable logic chips (PALs) so he could prototype different approaches to hardware design rapidly in the process of developing the system, delaying the point at which it became fixed—a great example of the use of optionality to delay making final decisions.[15]

After two years of development, the new firmware platform, FutureSmart, was launched. As a result, HP had *evolved* a set of processes and tools that substantially reduced the cost of no-value-add activities in the delivery process while significantly increasing productivity. The team was able to achieve "predictable, on-time, regular releases so new products could be launched on time."[16] Firmware moved off the critical path for new product releases for the first time in twenty years. This, in turn, enabled them to build up trust with the product marketing department.

As a result of the new relationship between product marketing and the firmware division, the FutureSmart team was able to considerably reduce the time spent on planning. Instead of "committing to a final feature list 12 months in advance that we could never deliver

15 *http://folklore.org/StoryView.py?story=Macintosh_Prototypes.txt*

16 Gruver, G. (2012). *A Practical Approach to Large-Scale Agile Development: How HP Transformed LaserJet FutureSmart Firmware.* Addison-Wesley., p. 89.

due to all the plan changes over the time,"[17] they looked at each planned initiative once every 6 months and did a 10-minute estimate of the number of months of engineering effort required for a given initiative, broken down by team. More detailed analysis would be performed once work was scheduled into an iteration or mini-milestone. An example of the output from one of these exercises is shown in Figure 4-5.

		High-Level Estimate – FW Engineering Months											
Rank	*Initiative*	*Component 1* (25-30)	*Component 2* (20-25)	*Component 3* (30-40)	*Component 4* (30-40)	*Component 5* (20-30)	*Component 6* (20-30)	*Component 7* (20-30)	*Component 8* (15-25)	*Component 10* (40-50)	*Component 11* (20-30)	*Component 12* (20-30)	*other teams*	**TOTAL**
1	Initiative A			*21*			*5*	*3*		*1*				***30***
2	Initiative B	*3*							*4*				*17*	***24***
3	Initiative C		*5*							*2*	*1*	*1*		***9***
4	Initiative D							*10*		*2*	*2*	*2*		***16***
5	Initiative E					*20*						*3*	*5*	***28***
6	Initiative F	*23*							*5*	*6*			*2*	***36***
7	Initiative G									*2*				***2***
8	Initiative H											*5*		***5***
9	Initiative I												*3*	***3***
10	Initiative J		*20*	*27*			*17*			*39*	*17*	*21*	*9*	***150***
11	Initiative K			*3*	*30*		*3*		*3*	*14*			*12*	***65***
12	Initiative L									*2*				***2***
13	Initiative M	*3*						*10*		*6*	*6*	*6*		***31***
		29	*25*	*51*	*30*	*20*	*25*	*23*	*12*	*74*	*26*	*38*	*59*	***401***

Figure 4-5. Ballpark estimation of upcoming initiatives[18]

This is significantly different from how work is planned and estimated in large projects that often create detailed functional and architectural epics which must be broken down into smaller and smaller pieces, analyzed in detail, estimated, and placed into a prioritized backlog *before* they are accepted into development.

17 Gruver (2012), p. 67.

18 Gruver, Gary, Young, Mike, Fulghum, Pat. *A Practical Approach to Large-Scale Agile Development: How HP Transformed LaserJet FutureSmart Firmware*, 1st Edition, (c) 2013. Reprinted by permission of Pearson Education, Inc. Upper Saddle River, NJ.

Ultimately the most important test of the planning process is whether we are able to keep the commitments we make to our stakeholders, including end users. As we saw, a more lightweight planning process resulted in firmware development moving off the critical path, while at the same time reducing both development costs and failure demand. Since we would expect failure demand to *increase* as we increase throughput, this is doubly impressive.

Three years after their initial measurements, a second activity-accounting exercise offered a snapshot of the results the FutureSmart team had achieved with their approach, shown in Table 4-2.

Table 4-2. Activity of the HP LaserJet Firmware Team in 2011

% of costs	Activity	Previously
2%	Continuous integration	10%
5%	Agile planning	20%
15%	One main branch	25%
10%	Product support	25%
5%	Manual testing	15%
23%	Creating and maintaining automated test suites	0%
~40%	Innovation	~5%

Overall, the HP LaserJet Firmware division changed the economics of the software delivery process by adopting continuous delivery, comprehensive test automation, an iterative and adaptive approach to program management, and a more agile planning process.

Economic Benefits of HP FutureSmart's Agile Transformation

- Overall development costs were reduced by ~40%.
- Programs under development increased by ~140%.
- Development costs per program went down 78%.

- Resources driving innovation increased eightfold.

The most important point to remember from this case study is that the enormous cost savings and improvements in productivity were only possible on the basis of a large and ongoing *investment* made by the team in test automation and continuous integration. Even today, many people think that Lean is a management-led activity and that it's about simply *cutting costs*. In reality, it requires *investing* to remove waste and reduce failure demand—it is a worker-led activity that, ultimately, can continuously drive down costs and improve quality and productivity.

Managing Demand

Up to now, we've been discussing how to improve the throughput and quality of the delivery process. However, it is very common for this kind of improvement work to get crowded out by business demands, such as developing new features. This is ironic, given that the whole purpose of improvement work is to increase the rate at which we can deliver as well as the quality of what gets delivered. It's often hard to make the outcome of improvement work tangible—which is why it's important to make it visible by activity accounting, including measuring the cycle time and the time spent serving failure demand such as rework.

The solution is to use the same mechanism to manage both demand and improvement work. One of the benefits of using the Improvement Kata approach is that it creates alignment to the outcomes we wish to achieve over the next iteration across the whole program. In the original Improvement Kata, the target conditions are concerned with process improvement, but we can use them to manage demand as well.

There are two ways to do this. In organizations with a generative culture (see Chapter 1), we can simply specify the desired business goals as target conditions, let the teams come up with ideas for features, and run experiments to measure whether they will have the desired impact. We describe how to use impact mapping and hypothesis-driven development to achieve this in Chapter 9. However, more traditional enterprises will typically have a backlog of

work prioritized at the program level by its lines of business or by product owners.

We can take a few different approaches to integrating a program-level backlog with the Improvement Kata. One possibility is for teams working within the program to deploy the Kanban Method, as described in Chapter 7. This includes the specification of work in process (WIP) limits which are owned and managed by these teams. New work will only be accepted when existing work is completed (where "completed" means it is at least integrated, fully tested with all test automation completed, and shown to be deployable).

Managing Cross-Cutting Work

Implementing some features within a program will involve multiple teams working together. To achieve this, the HP FutureSmart division would set up a small, temporary "virtual" feature team whose job is to coordinate work across the relevant teams.

The HP FutureSmart program, some of whose teams were using Scrum, took the approach of specifying a target velocity at the *program* level. Work adding up to the target velocity was accepted for each iteration, approximating a WIP limit. In order to implement this approach, all work was analyzed and estimated at a high level before being accepted. Analysis and estimation was kept to the bare minimum required to be able to consistently meet the overall program-level target conditions, as shown in Figure 4-5.

Do Not Use Team Velocity Outside Teams

It is important to note that specifying a target velocity at the program level does *not* require that we attempt to measure or manage velocity at the team level, or that teams must use Scrum. Program-level velocity specifies the expected work capacity of all teams based on high-level estimates, as shown in Figure 4-5. If a team using Scrum accepts work based on these high-level feature specifications, they then create lower-level stories with which to work.

Scrum's team-level velocity measure is not all that meaningful outside of the context of a particular team. Managers should *never* attempt to compare velocities of different teams or aggregate estimates across teams. Unfortunately, we have seen team velocity used as a measure to compare productivity between teams, a task for which it is neither designed nor suited. Such an approach may lead teams to "game" the metric, and even to stop collaborating effectively with each other. In any case, it doesn't matter how many stories we complete if we don't achieve the business outcomes we set out to achieve in the form of program-level target conditions.

In this and the next chapter, we describe a much more effective way to measure progress and manage productivity—one that does not require all teams to use Scrum or "standardize" estimates or velocity. We use activity accounting and value stream mapping (described in Chapter 7) to measure productivity, and we use value stream mapping combined with the Improvement Kata to increase it—crucially, at the value stream level rather than at the level of individual teams. We measure and manage progress through the use of target conditions at the program level, and if we need to increase visibility, we reduce the duration of iterations.

Creating an Agile Enterprise

Many organizations look to try and adopt agile methods to improve the productivity of their teams. However, agile methods were originally designed around small, cross-functional teams, and many organizations have struggled to use these methods at scale. Some

frameworks for scaling agile focus on creating such small teams and then adding structures to coordinate their work at the program and portfolio level.

Gary Gruver, Director of Engineering for FutureSmart, contrasts this approach of "trying to enable the efficiencies of small agile teams in an enterprise" with the FutureSmart team's approach of "trying to make an enterprise agile using the basic agile principles."[19] In the FutureSmart approach, while the teams ran within tight guide rails in terms of engineering practices (which we discuss in more detail in Chapter 8), there was relatively little attention paid to whether they had, for example, implemented Scrum at the team level. Instead, teams have relative autonomy to choose and evolve their own processes, provided they are able to meet the program-level target conditions for each iteration.

This required that engineering management had the freedom to set their own program-level objectives. That is, they didn't have to get budget approval to pay for process improvement work such as test automation or building out the toolchain for continuous integration. Indeed, the business wasn't even consulted on this work. All business demand was also managed at the program level. Notably, product marketing requests always went through the program-level process, without feeding work directly to teams.

Another important consideration is the way enterprises treat metrics. In a control culture, metrics and targets are often set centrally and never updated in response to the changes in behavior they produce. Generative organizations don't manage by metrics and targets. Instead, the FutureSmart management "use[s] the metrics to understand where to have conversations about what is not getting done."[20] This is part of the strategy of "Management by Wandering Around" pioneered by HP founders Bill Hewlett and Dave Packard.[21] Once we discover a problem, we ask the team or person having a hard time what we can do to help. We have discovered an opportunity to improve. If people are punished for failing to meet targets or metrics, one of the fallouts is that they start manipulating work and

19 Gruver (2012), Chapter 15.

20 Gruver (2012), p. 38.

21 Perhaps it's better characterized as "Management by Wandering Around and Asking Questions." In the Toyota Production System, this is known as a *gemba* walk.

information to look like they are meeting the targets. As FutureSmart's experience shows, having good real-time metrics is a better approach than relying on scrums, or scrums of scrums, or Project Management Office reporting meetings to discover what is going on.

Conclusion

The Improvement Kata provides a way to align teams and, more generally, organizations by taking goals and breaking them down into small, incremental outcomes (target conditions) that get us closer to our goal. The Improvement Kata is not just a meta-methodology for continuous improvement at the enterprise and program level; it is a way to push ownership for achieving those outcomes to the edges of the organization, following the Principle of Mission. As we show in Chapter 9, it can also be used to run large programs of work.

The key characteristics of the Improvement Kata are its iterativeness and the ability to drive an experimental approach to achieve the desired target conditions, which makes it suitable for working in conditions of uncertainty. The Improvement Kata is also an effective way to develop the capabilities of people throughout the enterprise so they can self-organize in response to changing conditions.

The FutureSmart case study shows how a large, distributed team applied the Improvement Kata meta-method to increase productivity eightfold, improving quality and substantially reducing costs. The processes and tools the team used to achieve this transformation changed and evolved substantially over the course of the project. This is characteristic of a truly agile organization.

Implementing an enterprise-level continuous improvement process is a prerequisite for any ongoing large-scale transformation effort (such as adopting an agile approach to software delivery) at scale. True continuous improvement never ends because, as our organization and environment evolve, we find that what works for us today will not be effective when conditions change. High-performance organizations are constantly evolving to adapt to their environment, and they do so in an organic way, not through command and control.

Questions for readers:

- Do you know how much time your engineering organization is spending on no-value-add activities and servicing failure demand versus serving value demand, and what the major sources of waste are?
- Must engineering teams get permission to invest in work that reduces waste and no-value-add activity across the value stream as a whole, such as build, test, and deployment automation and refactoring? Are such requests denied for reasons such as "there is no budget" or "we don't have time"?
- Does everyone within the organization know the short- and long-term outcomes they are trying to achieve? Who decides these outcomes? How are they set, communicated, reviewed, and updated?
- Do teams in your organization regularly reflect on the processes they use and find ways to experiment to improve them? What feedback loops are in place to find out which ideas worked and which didn't? How long does it take to get this feedback?

O'REILLY®

Designing Delivery

Rethinking IT in the Digital Service Economy

Jeff Sussna

IT as Conversational Medium

The following content is excerpted from *Designing Delivery*, by Jeff Sussna. Available now in Early Release.

Serving as a medium for digital conversation is IT's new mandate. Responding to this mandate means radically transforming IT's understanding of its purpose, as well as its approach to fulfilling that purpose. It means replacing information systems management with empathic conversation as both the driving force and the ultimate goal by which post-industrial IT measures itself.

IT has traditionally taken an industrial approach to building and operating systems. It has concentrated on implementing large-scale solutions with high up-front costs and equally large presumed return-on-investment. Development has proceeded in linear phases, from initial conception, through design, development, and testing, to completion. Project management has used sophisticated tools and processes which have required specialized expertise.

IT took this approach under the impression that it could, and must, predict needs and capabilities in advance. Business requirements were seen to be highly complicated, and thus to need highly complicated solutions and solution implementations. IT organizations invested millions of dollars in software solutions such as Enterprise Resource Planning (ERP) systems that were intended to model the entire business.

All-encompassing enterprise projects took years to deploy, and often cost as much to implement as they did to buy. Sadly, more projects

failed than succeeded. IT developed a reputation for an inability to reliably deliver on its claims. Large-scale IT projects generally failed for three reasons:

- The project plan failed to accurately predict or account for the challenges that inevitably accompany complicated system implementations
- The requirements failed to accurately model users' real needs
- Over the course of a lengthy implementation project, users' needs changed

From a cybernetic perspective, the reasons for failure are obvious. Traditional IT project planning and management had no mechanism for incorporating feedback. It had no way to self-correct, other than by throwing away months or years of work and starting over.

IT's inability to reliably or efficiently deploy useful solutions is just the beginning of its troubles. It also struggles to operate those systems. Once an IT organization has managed to deploy a complicated, expensive software system, it needs to keep that system running. Change introduces uncertainty and risk; IT therefore prefers to minimize change.

IT traditionally minimizes risk and manages uncertainty through formal processes. Methodologies such as ITIL give IT a handle on change; unfortunately, it does so by introducing bureaucracy and friction. In addition to its reputation for outright failure, IT has thus gained an unfortunate reputation for slowness and lack of responsiveness. As a result, industrial IT lacks important characteristics that define a conversational medium. It has neither the ability to move quickly, nor to change direction quickly. It lacks a mechanism for incorporating feedback from the rest of the company, and more importantly, for helping the company do the same with its customers.

Some IT organizations, though, have begun adopting an interrelated set of new practices. Together, these practices have the potential to compose the digital medium post-industrial businesses need in order to self-steer through empathic conversations. Chief among them are:

- Agile

- DevOps
- Cloud Computing
- Design Thinking

These practices all share a cybernetic model of control. This chapter examines them in turn, exploring the specific ways each one helps digital businesses improve service through feedback-based adaptation. It then describes how they come together to create a unified, empathic, conversational medium.

Agile

In February of 2001, seventeen early adopters of so-called "light" software development methodologies gathered at the Snowbird ski resort in Utah. They came together to seek common ground between their various approaches. The result is generally considered to be the formal birth of the Agile Software movement. It took the form of a manifesto that expressed four main values:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

Since then, the Agile movement has manifested through a variety of specific practices. Some, such as Extreme Programming and Scrum, were represented at Snowbird, and have continued through to today. Others have evolved in the intervening years. They all, though, share key characteristics that make Agile the grandfather of cybernetic approaches to delivering software.

The word "agile" means "quick and light in movement". Agile arose out of frustration with traditional, heavyweight, "waterfall" software development methodologies. The software industry had become infamous for projects that ran over budget and schedule, while failing to deliver what customers really needed. Software project management generally followed an approach one might describe as anti-cybernetic. It was considered important to "get the requirements right up front". Development shouldn't start until you understood, and expressed, the project's requirements in full detail. Testing shouldn't start until development was believed to be complete.

The waterfall model suffered from key conceptual shortcomings. Requirements are very difficult to "get right" without feedback. First, it's difficult for users to understand and articulate their own needs. Second, it's hard for designers to accurately understand what users are trying to say. The design industry's use of prototyping reflects their recognition of these realities. Finally, even if a user understands what they need, and can communicate it to a designer, that need is likely to change to some degree by the time it's been fully implemented.

Waterfall projects are thus almost inevitably doomed to build things that don't properly match users' needs. Even in the case where an up-front design succeeds, the sequential approach to development and testing generates waste, adds complexity, and compounds errors. Testing can't actually happen "after" development. Bugs must be fixed and then retested. Fixing bugs is part of development. Testing is by definition an iterative and circular process.

Not starting testing until late in the process bunches up large numbers of bugs. Those bugs apply to months worth of code changes. Debugging and fixing them is complex and time-consuming. Some of this complexity comes from trying to analyze code written a long time ago, while some comes from trying to determine which of many code changes introduced the bug.

Agile methodologies combat the shortcomings of waterfall by breaking development into shorter iterations. Each iteration contains a design/development/testing cycle. It ends with presentation of working software to users. The benefits of this approach all have to do with speeding up the feedback loop. Testing happens more quickly after code has been written. Bug-fixing addresses smaller, more manageable change sets. Users see the results of development sooner and more often. They have the chance to say "that's not quite what I meant" with less painful impact. Development teams are thus less shy about getting feedback and making changes. The result is a process that lets a software organization steer its way to genuine customer value.

Continuous Integration

Some Agile teams go even further in integrating testing deeply into the development process. Unit testing makes test-writing part of the developer's job. With unit tests, tests get run every time the software

gets built, rather than either at the end of the development cycle or the end of an iteration. Continuous Integration (CI) drives this approach to its ultimate conclusion by ensuring the software gets built, and tested, every time any developer commits a change. CI shrinks the distance between change and feedback to the smallest possible increment. It makes identifying, diagnosing, and repairing bugs as simple as possible by drastically constraining the amount of change being tested at one time.

Agile understands the importance of testing as a feedback mechanism. Without continuous, specific information about the gap between expected and actual in the past, steering into the future can't happen. Scrum uses the practice of retrospectives to take team-level self-steering even further. A retrospective is a regularly-scheduled pause in the iterative development cadence to gather feedback about how well the team's process is working. Retrospectives give every team member the chance to reflect on, and discuss what went well, what didn't go well, and what could or should be adjusted in future iterations.

Self-Organization

In order to maximize user feedback, Agile emphasizes integrated teams that foster communication and collaboration across disciplines. Testers and product owners, both of whom represent customers' needs and priorities, participate as team members rather than third parties. Some technical teams even integrate design and marketing representatives. Incorporating diverse perspectives lets agile teams understand and resolve gaps between desired and actual results more quickly, thus enabling them to accomplish better work more efficiently.

Agile also emphasizes the power of self-organizing teams. Breaking from the Taylorist industrial tradition, self-organizing teams rely on their members to cooperatively define as well as execute work. Teamwork happens through conversation rather than via an externally managed assembly line. Just as with any cybernetic process, intra-team feedback improves agile teams' ability to cooperate in the face of change. It lets them self-steer by reconfiguring themselves as needed, without losing coherency or effectiveness.

Being vs. Doing Agile

The core of Agile is agility. Done right, it lets teams leverage the power of cybernetics to continuously deliver value in an environment of uncertainty and change. Techniques such as sprint demos and CI maximize feedback. Iterations and backlogs maximize the ability to change direction in response to feedback. Cross-functional collaboration maximizes the ability to listen accurately.

At their best, agile teams function as foundational elements of self-steering organizations. Their cybernetic practices let them flex with minimal friction. Their cross-functional structure lets them converse with the rest of the corporation. It also helps them have the internal empathic conversations they need to participate in the corporation's unity of purpose.

Unfortunately, IT organizations used to industrial thinking often unconsciously fall prey to the temptation to coopt Agile's post-industrial operating model. Daily standups, instead of being a technique for quick feedback and problem-solving, become a rote reporting tool. Burndown charts become a micromanagement hammer, shifting the focus away from delivering useful value and towards finishing assigned tasks on time. Sprint demos become an opportunity to tell users what the team has done, rather than a chance to listen to feedback in order to validate the accuracy and continued usefulness of the team's work.

Retrospectives have the potential to be a wonderful, 2nd-order cybernetic mechanism. When used properly, they can help teams avoid getting locked into habitual methods. They keep everyone's focus on the question of whether the team is accomplishing its underlying goal: continuously responding to changing customer needs.

Unfortunately, more often than not, retrospectives fall victim to the "not enough muffins" syndrome. Instead of soliciting feedback about the efficiency with which the team is pursuing its goals, and exploring ways to self-correct, participants content themselves with complaining about random obstacles. For a team with a tradition of bringing in food for team breakfasts every Monday morning, not having enough muffins may be a valid complaint. It doesn't, however, address questions about the team's deeper purpose.

In order for Agile to successfully support self-steering, its practitioners must adopt a post-industrial worldview. Agile's challenges are are instructive for all of the would-be components of the digital medium. IT needs to keep its focus on the goal, which is self-steering through conversation. It must continually evaluate and improve its ability to support that goal. It must put the ends before the needs. It must learn to view all techniques as provisional and subject to adaptation through their own cybernetic process. "Are we being agile" must always remain primary over "are we doing Agile?"

DevOps

Agile emphasizes delivering value to customers above all else. By itself, though, it doesn't fully address the problem of getting the potential value generated by development into customers' hands. This gap reflects the pre-service era, when software was primarily delivered as a product. A software company would build and ship an application, with the expectation that the customer was responsible for operating it. The software was 'done' when it had been designed, developed, tested,and burned onto a CD. Installing the software from that CD onto a server, or adding memory or upgrading operating systems in order to be able to run it, or figuring out what had gone wrong when it failed, were all things left to the customer.

Post-industrialism has impacted software just as dramatically as any other area of the economy. Software-as-a-Service (SaaS) means that the same company that builds an application also operates it on their customers' behalf. Functionality and operability become inseparable in customers' minds. They judge a software service as much on the basis of performance, availability, and security as they do on whether it helps them accomplish their desired tasks.

DevOps seeks to complete the equation started by Agile, and to address the challenges SaaS poses. The name is a portmanteau of "development" and "operations". It reflects an understanding of the need to unify functional and operational concerns.

The industrial approach to IT worked by creating silos between specialties, and using rigid mechanisms to govern communication between them. Not only did development and operations live different lives; operations sub-specialties such as networking, database management, and security did as well. Delivering functionality and operability required navigating multiple layers of separation. Each

layer added time, effort, and misunderstanding. Each sub-organization viewed the others with suspicion: "they don't understand the importance of …"; "they expect us to figure out …'; and so on.

Deeply siloed IT creates waste by requiring bureaucratic communications processes that don't add value. Forcing development to navigate multi-layered approval processes in order to deploy code slows that code's availability to customers. Silos within operations exacerbate the problem. Since each sub-specialty is measured based on the same metric - stability - they are incented to resist change, not just coming from development, but also from each other. The result is friction built upon friction.

DevOps recognizes that change-averse IT operations is incompatible with post-industrial business. It seeks to dissolve the dichotomy between quality and speed, and to overcome the impedance mismatch between development and operations. Unlike operations, development is incented to generate change, not avoid it. Companies pay developers to write code. When you write code, you are either adding, updating, or removing something. Every line of code you write changes something.

SaaS drives a similar focus on speed into operations. New applications require new infrastructure. User traffic that spikes by the day, hour, or minute requires elastic infrastructure. Customer-visible operational problems require fast resolution. SaaS replaces the need not to break things with the need to make things happen in response to environmental change.

The Three Ways

In their book "The Phoenix Project" [1], Gene Kim, Kevin Behr, and George Spafford describe DevOps as consisting of "Three Ways". Tim Hunter concisely expresses the Three Ways of DevOps as "Flow", "Feedback", and "Continual Experimentation and Learning" [2]. This triad clearly expresses DevOps as an extension of Agile's cybernetic model to the end-to-end software value stream.

1 *http://www.amazon.com/The-Phoenix-Project-Helping-Business/dp/0988262592*

2 *http://itrevolution.com/a-personal-reinterpretation-of-the-three-ways/*

Flow

Flow comprises techniques to remove waste from the process of delivering concrete value to customers. It redefines "done" in the language of service rather than products. Many organizations that call themselves agile define "done" as "sprint tasks completed", or "unit tests passing", or "signed off by testing and product owner". DevOps redefines "done" as "available to customers and operating satisfactorily".

Flow includes three key practices:

- Cross-silo collaboration
- Automation
- Continuous delivery

Cross-silo collaboration uses human communication to improve quality and reduce waste. When developers, system and database administrators, and security engineers all talk to each other, they can see problems and find solutions in the spaces between their domains. The result is improved quality, achieved more efficiently. On a more subtle level, cross-silo collaboration also improves flow by generating empathy and dissolving suspicion. People are naturally more prone to remove roadblocks and help each other when they have mutual trust and understanding.

Automation seeks to reduce manual tasks that unnecessarily consume time and cause errors. There's no good reason to manually configure five hundred servers in the exact same manner. There's even less reason to spend hours or days debugging a production problem that was caused by one server having been configured slightly differently from the other 499.

Automation leverages lessons from agile development. It uses high-level configuration languages that bring the benefits of encapsulation, abstraction, and reuse to system administration. It treats "infrastructure as code", turning computing environments into abstract configurations that can be version-controlled, unit tested, and continually integrated. As a result, they can be created, changed, and replicated quickly and safely, with guaranteed consistency.

IT normally delivers changes to customers in batches called "releases". So-called "big-bang" releases delay value delivery by weeks, months, or even years. Just as with any other large-granularity activ-

ity, batched releases increase the complexity, difficulty, and risk of deploying changes to production environments. Risk-averse IT organizations respond counterproductively by trying to reduce the number of releases, thus locking themselves into a vicious cycle by making them even more risky.

Continuous Delivery (CD) does for deployment what CI does for integration testing. It uses comprehensive automation and rigorous testing to enable immediate production deployment of any change deemed acceptable, regardless of how small. A release might be a small as a one-line bug fix.

By reducing batch size, release latency, and errors due to manual processes, and by guaranteeing comprehensive testing, CD reduces risk and increases confidence. It leads to fearless releases. With the confidence to release any change, any time, the organization gains the flow needed to let it respond immediately to its customers. That flow lets companies have the most intimate conversations at the most valuable times.

Many companies with seasonal business cycles implement production freezes: times during the year when only emergency changes can be deployed to business-critical systems. The rationale for this approach is understandable: if you're an online retailer, the day after Thanksgiving is the day you least want to break your website. Unfortunately, it also minimizes your ability to respond to your customers when it's most important. Fixing a bug or releasing a desired new feature on January 10th instead of November 28th doesn't benefit anyone.

When they first hear about CD, marketing and business operations often respond with alarm. They envision their customers being inundated with uncontrolled, unchaperoned change. Decoupling technical deployment from customer visibility is a key component of CD. Through mechanisms such as feature flags and segmented releases, CD frees the business to truly control its conversation with customers.

Feature flags make it possible to precisely control which features are visible and when. If marketing wants to reveal a new feature at 12:01 AM on Christmas morning, they don't need IT to deploy that code at 12:01. They can simply flip a virtual switch. Segmented releases turn what used to be known as "final" or "golden" into controlled experimentation. Marketing can expose a feature to specific demo-

graphics, or use A-B testing to release multiple versions of a feature simultaneously.

Feedback

Flow lets IT organizations efficiently deliver functionality and operability to customers. Cybernetic conversations require equally efficient feedback. In the context of DevOps, feedback seeks to provide information back to development and operations as continuously as possible. Its goal is to generate a frictionless loop that lets IT hear its customers quickly and accurately across silos.

Monitoring provides the visibility necessary to process feedback. IT should treat it as an integral part of application and system design, implementation, and operations. It must be possible to ask questions on multiple levels. What is the state of the infrastructure? What is the state of the application? What is the state of the users' behavior?

From a DevOps perspective, teams across silos need the ability to listen to each others' signals. Multi-level monitoring lets them see things from each others' perspectives. For example:

- Users are abandoning our site because it's too slow
- It's too slow because we don't have sufficient infrastructure
- We don't have sufficient infrastructure because adding more is expensive and time-consuming
- An on-demand cloud might make infrastructure cheaper, easier, and faster to provision
- …

The delivery lifecycle itself is a cybernetic process. As such, it also requires the ability to process feedback. Information radiators offer global visibility into flow:

- What is the state of the current build?
- Why did a given build fail?
- Where in the pipeline is a given feature?
- Which customer segments have access to a given feature?

Continuous Experimentation and Learning

The combination of flow and feedback creates an efficient cybernetic process that lets IT shift its focus from maintaining the status quo out of fear, to fearlessly enabling continuous change and experimentation. DevOps lets IT unfreeze digital systems, transforming them into a continually flowing fluid. It lets IT unleash software as a dynamic business tool that digitally infused companies can use to deliver service and respond to disruption.

DevOps frees post-industrial businesses to fully integrate software into their self-steering conversations. Marketing and business operations can use CD to turn features on and off with minimal latency, and multi-level monitoring to understand the relationship between their changes and their customers' responses. They can thus treat software, not as something to which they must commit, but rather as something they can continually reshape through experimentation and learning.

Cloud Computing

Part of IT's challenge stems from the fact that provisioning computing resources is expensive and slow. As with any physical object, servers, storage arrays, and network devices have long lead times. They must be specified, ordered, built, shipped, and installed. Deploying a new application, or scaling infrastructure to support increased usage, requires high-latency physical intervention. The high cost of modifying infrastructure contributes to IT's resistance to change.

Cloud computing makes it possible to treat computing resources as an on-demand utility. It works by virtualizing physical resources, and providing web and API-based provisioning interfaces. By abstracting the underlying physical reality, cloud lets users dynamically provision and de-provision resources for themselves. They can even integrate cloud provisioning API's into their automation pipelines to create an elastic computing substrate that responds to change in real-time.

Cloud lets organizations consume IT based on need, pay for it based on consumption, and delegate its management to the provider. In the process, it transforms IT from a relatively static capital expenditure to a highly dynamic operational expenditure. In both technical

and financial terms, cloud more closely aligns IT with the ebb and flow of the business.

Cloud computing addresses multiple layers of IT. Infrastructure-as-a-Service (IaaS) turns physical infrastructure into a utility. Platform-as-a-Service (PaaS) turns application execution environments into a utility. SaaS turns applications themselves into a utility.

All three layers share the "as-a-Service" moniker. This common name reflects the impact of service, infusion, and disruption on IT itself. While cloud makes IT more agile by increasing provisioning speed and reducing sunk costs and management complexity, it also has a more subtle and profound effect. When IT resources are accessible through the same digital interfaces as any other application, it becomes feasible for non-operations groups to manage their own resources. IaaS and PaaS let development teams provision their own application substrates. SaaS lets non-technical organizations provision their own applications.

Cloud's ultimate effect is to remove IT as a bottleneck to self-steering. On one level, it lets IT flex in response to feedback at the same rate as the organizations using those resources. On a deeper level, it begins to dissolve the separation between IT and its users. As part of self-steering, organizations need access to the digital medium in order to converse with each other and with customers. The more flexible and direct that access is, the more efficient those conversations can be.

Microservices

By providing low-friction access to the digital medium, cloud computing allows systems and applications to proliferate. Microservices are the ultimate expression of this effect. Microservices decompose large applications into small, loosely coupled grains of functionality.

By holding units of functionality at arms length from each other, microservices enable more continuous, lower-risk change. Agile and continuous delivery use smaller batch sizes to increase speed and quality simultaneously. Microservices provide a similar effect at the level of application architecture and organizational structure.

Microservices work by reducing the scope of concern. Developers have to worry about fewer lines of code, fewer features, and fewer interfaces. They can deliver functional value more quickly and often,

with less fear of breaking things, and rely on higher-order emergent processes to incorporate their work into a coherent global system.

In order for microservices to work, though, operations needs a similar conceptual framework. Trying to manage an entire universe of microservices from the outside increases the scope of concern instead of reducing it. The solution is to take the word "service" seriously. Each microservice is just that: a software service. The team that builds **and** operates it need only worry about it and its immediate dependencies. Dependent services are customers; services upon which a given microservice depends are vendors.

Microservices thus represent a new IT organizational model as much as a new architectural model. Seen this way, they leverage the power of Conway's Law, named after Melvin Conway. Conway was a computer programmer who observed that "organizations which design systems … are constrained to produce designs which are copies of the communication structures of these organizations" [3].

Conway's Law tells us that software architectures and the organizations that make them mirror each other. IT can leverage this effect to everyone's benefit. Microservices map well to so-called "two pizza-sized", interdisciplinary teams. These teams take responsibility for the entire service delivery lifecycle for their particular microservice.

Microservice-oriented organizations shift IT architectures, processes, and inter-relationships from a complicated-systems model to a complex-systems model. Making that transition lets IT better respond to post-industrial business challenges. It lets different parts of the organization, as represented by different microservices, fluidly adapt to each other and to external change. By taking the "service" in "microservice" seriously, it also weaves an empathic, conversational mentality deeply into the fabric of IT.

Digital infusion makes the relationship between IT and the business it supports all the more important. One could extend Conway's Law to state that digital businesses are constrained to deliver service in ways which reflect the structure and activity of their IT organizations. Using microservices to give itself a more organic structure increases IT's ability to self-steer through flexible, scalable, internal

3 *http://en.wikipedia.org/wiki/Conway%27s_law*

conversations. That ability in turn drives improved self-steering capabilities for the organization as a whole.

Design Thinking

Digital infusion is having a profound affect on the role of design in IT and software services. As the digital realm becomes ever more central to our lives, the design of its interfaces becomes ever more important to people's quality of life. Design brings its own cybernetic sensibility to delivering products and services. The design community has encoded this sensibility into a set of practices and principles known as *Design Thinking*.

Design Thinking is built upon four foundational practices:

- Empathy
- Ethnography
- Abductive thinking
- Iterative user testing

Empathy makes the customer's perspective on a problem the starting point for all design activity. It reflects the philosophy of user-centeredness. No matter how beautiful a design solution, it doesn't actually solve anything if it doesn't work from the user's point of view. An elegant chair that no one can sit in is a user-centered design failure.

Ethnography is a disciplined process of non-judgmentally observing users within their own realms. Without ethnography, designers risk unconsciously imposing their own biases instead of truly seeing the problem from the customer's perspective. Many design teams have described their experience of starting a project believing they were trying to solve a certain problem. After conducting ethnographic research, they realized the real problem was completely different. Having engaged in that research saved them from wasting everyone's time building the wrong solution.

Abductive thinking is the process of finding creative solutions where there are no *correct* or *best* ones. Abductive thinking succeeds in situations where analytical engineering fails. It strives for designs that are practical as well as beautiful and inspiring.

Iterative user testing forces designers to repeatedly test and revise their beliefs about a solution. Design thinking views the development of a solution to a problem as the starting point, not the conclusion. User testing exposes proposed designs to the harsh reality of usage in the form of prototypes. It treats users' experience of those prototype as feedback.

Repeated revision and retesting leads to successively better designs. Ethnography, user testing, and iterative solution discovery incorporate feedback into the essential process of design. As design thinking evangelist Elina Zheleva puts it, design follows a circular "understand-act" process [4].

Service Design

Design thinking expresses designers' sensibilities in a form that can be applied to problems beyond traditional disciplines such as graphic and product design. In particular, design thinking has tremendous insights to offer to the creation of services. By engaging in empathy, ethnography, abductive thinking, and iterative user testing, designers can create services that genuinely help customers accomplish their goals.

Service Design applies the principles of design thinking to the design of services. It centers its practice around the customer journey, which represents customer's unfolding experiences over time across all of a service's touchpoints. Applying design thinking to the customer journey helps create experiences that are satisfying and coherent rather than challenging and disjoint.

Service design also addresses service fulfillment. It uses a technique called service blueprinting to chart the relationship between so-called "front-stage" and "back-stage" activities. However well-designed a kiosk interface is, it can't provide a satisfying experience if it doesn't properly integrate needed back-office information. All of the component human and computer processes needed to generate that information must mesh with each other.

Customer journey maps and service blueprints help service organizations understand themselves and their customer interactions across silos and layers, across physical and virtual interfaces, and

4 *https://medium.com/@ellie_zheleva/the-two-step-design-thinking-process-955e6087020*

across human and computerized processes. Digital infusion necessitates seamless integration across of these dimensions.

A self-steering organization needs to think in terms of relationships and systems. It needs a way to visualize its internal conversations and how they enable or distort its customer conversations. Through customer journey maps and service blueprints, service design has the potential to provide such a mirror.

Unifying Design and Operations

Post-industrial business involves operating a continuous "listening loop" through which companies can respond efficiently and accurately to customer needs. In order to maintain their viability through self-steering, organizations need to map that external listening loop to a similar set of internal conversations. Digital infusion means that internal and external conversations all happen through software-enabled service. 21st-century business thus relies on IT to enable the continuous design and operation of responsive digital services. These services provide the medium through which cybernetic conversations flow, both between companies and their customers, and among employees and groups within a company.

The purpose of the digital conversational medium is not merely to deliver software, but rather to enable continuous, adaptive value co-creation. Whether it involves a company operating a website on behalf of its customers, or the finance department operating a microservice on behalf of the project management department, "service" is the key word in all cases. In the post-industrial era, service unfolds through the unification of design and operations.

In order to fulfill its purpose, a software service must work on multiple levels. It must provide suitable functionality. That functionality must be usable, whether through an interactive interface or through an API. It must be operable, so that its customers can access it when they need it, and rely on it for stability, security, and so forth. It must meets customers' needs throughout their journey. Finally, it must adapt to meet changing needs. If, for example, the project management department needs finer-grained project cost information, the finance department has to be able to update their service to provide that new information.

These requirements apply to all services, whether internally or externally facing. To fulfill its role as a conversational medium, IT therefore must address all aspects of fitness for purpose. It needs to incorporate the capabilities provided by methods such as Service Design, Agile, DevOps, and cloud computing into a coherent practice. This practice uses a unified set of principles to guide itself in finding comprehensive answers to the full suite of questions that define digital service needs.

The fundamental principles that guide IT as a digital conversational medium include:

- Design for service, not just software
- Minimize latency, maximize feedback
- Use operations as input to design
- Seek empathy

Designing for service means designing both for the whole customer and the whole organization. Designing for the whole customer starts by understanding their goals. It identifies the entire journey through which they interact with the service in order to accomplish those goals. It also identifies the larger context that surrounds their interaction with a given service. Finally, understanding the whole customer requires considering their needs beyond the obvious, utilitarian level. Productivity arises, not just from efficiency, but also from satisfaction and happiness. Making people's lives better through service thus contributes to meeting practical goals.

The co-creative nature of service means that the service organization also must be a design target. Those designing a service need to ask the same questions about internal, operational users as they do about customers. Most importantly, they must address the interrelationships between internal and external goals and journeys.

Conversational quality depends on efficient information exchange. Communicating by letter across continents, for example, is harder and slower than speaking in person. The digital conversational medium needs to minimize latency in the exchange between service providers and customers. Together, Agile, DevOps, and cloud computing serve this purpose. Agile minimizes the latency between discovering a need and implementing it. DevOps minimizes latency in the end-to-end delivery, understanding, and discovery process.

Cloud computing minimizes computing-resource deployment latency.

Continuously delivering functionality is only half of the conversational process. Talking without listening doesn't contribute to responsive service. When done properly, with a truly cybernetic sensibility, Design Thinking, Agile and DevOps complete the conversational equation. They continuously expose the service organization to accurate, thorough feedback, and provide mechanisms such as user testing, sprint demos, and information radiators that maximize the organization's ability to internalize and process that feedback.

Businesses normally treat operations as an output of design. The job of IT operations is to run the code produced by design and development. In order to empathize, though, one must be able to hear. In order to hear, one needs information from operations. Operations thus becomes an input to design.

Operational feedback comes from multiple sources on multiple levels, including:

- Infrastructure and application monitoring
- User behavior monitoring
- A/B, canary, and demographically targeted testing
- Analytics
- Customer support
- Social media

An effective conversational medium incorporates them all. A user interface change could annoy users, either because it degrades performance by increasing server load, or because it makes the application harder to use. Problems can become apparent through monitoring dashboards, or through users complaining on Twitter. Correlating feedback across business and technical layers is key to accurately diagnosing and fixing problems, whether they be caused by infrastructure problems or undiscovered customer needs.

Empathy is a cybernetic process of understanding through conversation. It needs to be both the foundation and the goal of the conversational medium. On the one hand, empathy should inform the pursuit of each of the other foundational principles. On the other hand,

those principles should all be approached as opportunities to develop further empathy.

Continuous Design

IT's new purpose is to help companies and their components parts design and operate software services. No longer can it content itself with running systems and responding to technical requests. Where IT used to be in the business of running things, and maintaining stability, it now must enter the business of enabling change. Ultimately, post-industrial IT's new mandate consists of delivering the capability for continuous design.

Design typically concerns itself with what comes next. It focuses on conceiving new solutions to current problems. Operations, on the other hand, concerns itself with what's happening now. Its purpose is to run and maintain whatever solution was created to a previously understood problem.

The cybernetic model of control breaks down the divisions these two modes of work. It unifies them through feedback. Feedback continually exposes gaps between the actual and the desired. In the process, it creates never-ending opportunities to co-create value by helping people solve problems.

Empathy drives digital businesses to use conversation as a basis for action. It gives them the ability to hear the feedback that operations provides, and exhorts them to respond to what they hear through re-design. Far from being something soft or weak, empathy drives economic sustainability by creating the impetus to design - or in other words, responsively operate - truly useful service capabilities.

The industrial product model uses design to generate solutions. Marketing then convinces customers of the usefulness of those solutions. Operations produces artifacts to meet the demand marketing has generated. A post-industrial approach to design, by contrast, uses it to generate conversations rather than complete them.

As design consultant and researcher Thomas Wendt explains in his book 'Designing for Dasein', the meaning of a digital service depends, not just on the intentions of its creators, but also on how people use it. Facebook was designed as a way for college students to connect with their friends. It has evolved into, among other things, a platform for catalyzing political action. At the same time, designed

solutions change the very problems they were created to solve. Facebook has deeply influenced the way people see themselves as individuals and as social beings. It has transcended its role as a medium for sharing experiences, to become a medium for having them.

Complexity further compromises the ability to create finished solutions. We can never fully know how our designs will work until real customers use them in real operating environments. Facebook has to grapple with social and political issues that Mark Zuckerberg never could have conceived of while writing the initial version in his college dorm room [5].

Post-industrialism transforms design into a circular process of continual learning and repair. Repeatedly responding to the next gap between actual and expected becomes the essence of what it means to be in business. Operations becomes design, and design becomes operations.

Design as a continuous, circular process is actually inherent in its very definition. Nobel Laureate and pioneering cognitive scientist Herbert Simon, in his seminal book 'The Sciences of the Artificial', defined design as "changing current situations into preferred ones". This definition has several important implications:

1. Design is not restricted to visual disciplines
2. Design fundamentally involves change
3. Design operates in the gap between actual ("current") and expected ("preferred")
4. Design does not strive for right, or even good solutions, but only relatively better ones from the user's perspective
5. Design's lack of objective finality maintains the openness that allows and encourages further change (what was preferred is now current)

Self-Steering as Continuous Design

Digital businesses design and operate service capabilities to help customers accomplish their goals. In order to co-create value, they must align their internal structure and activity with that of their cus-

5 *http://www.theguardian.com/technology/2015/feb/16/facebook-real-name-policy-suspends-native-americans*

tomers. Value co-creation arises from structural coupling between the service provider and its environment. If customers need help using an application, for example, the service organization must structure and manage itself to be able to provide help when and how it's needed.

This structural coupling must, of course, continually adapt to environmental change. Change comes from ever-evolving customer needs and market realities. Companies also contribute to environmental change through their own design process. Customers, markets, and companies continuously perturb each other.

In order to maintain their viability by adapting to environmental perturbation through self-steering, digital businesses must design and operate themselves just as they design and operate the services through which they interact with their customers. Continuous design is an autopoietic process. As part of designing/operating a service, a digital business must design/operate itself. To understand how to design/operate itself, a business must understand the design/operations of the service through which it maintains its viability.

In order to deliver a digitally infused taxi service, for example, Uber must do more than just design and operate a mobile app. It also must design and operate itself as an organization with capabilities such as calculating driver tips and conducting background checks. In the process of operating its service, Uber learns how well the service **and** its internal systems work.

Feedback from internal and external operations leads to further internal and external design. Drivers may complain about being under-tipped. Customers may complain about not feeling safe due to inadequate background checks, or about having to wait because of inaccurate arrival estimates. In order to address these complaints, Uber may need to redesign its service, its mobile applications, and its internal operational procedures.

Cybernetics challenges the solidity of the boundaries between inner and outer, production and consumption, and acting and responding. Autopoiesis defines business viability as continuous organizational adaptation to markets and customers. Continuous design reflects the cybernetic model on two levels. First, it treats design and operations as an inseparable, mutually influencing pair. Second, it treats the design of internal and external service relationships as a similar process of mutual influence and reflection.

From Design Thinking to DevOps and Back Again

Simon's definition of design as "changing current situations into preferred ones" reveals it as something that goes beyond traditional visual disciplines such as graphic, industrial, or web design. His definition gives us a lever we can use to reimagine IT itself as a form of design. Cybernetic methods such as Agile, DevOps, and cloud computing free IT from the need to map dynamic business needs to static technical systems and processes. As a result, IT can transform its view of itself from a source of friction into an agent of responsive change.

The capability to deliver continuous, empathic change is the defining characteristic of a digital conversational medium. The word "medium" means "an intervening substance, as air, through which a force acts or an effect is produced" [6]. Post-industrial IT's purpose is to make digital conversations between companies and their customers natural and effortless, the way water makes it possible for fish to swim. As an autopoietic medium, fulfilling its purpose also means simultaneously enabling equally effortless internal conversations.

IT as conversational medium transcends specific techniques. It represents more than just designing and operating services for customers or employees. It is not defined by the specific tools and practices that make it up. Its deepest value comes from infusing entire organizations with design thinking.

When the capacity for responsive change becomes frictionless and universally available, everyone can approach their work as continuous service design. Service characterizes all the relationships that define digital business. Post-industrial companies co-create value with customers through service. That co-creation relies on mutual internal service. Designing and operating service for others' benefit becomes the common driver for all activity at all levels of the organization.

IT's ultimate goal is, like water for the fish, to disappear. In a highly effective digital business, employees focus on continuously transforming empathic listening into acting. They conduct their daily work in order to change current situations into preferred ones. They

6 *http://dictionary.reference.com/browse/medium?s=t*

take it for granted that identifying the gap between current and preferred, and closing that gap, both happen via digital means. They no longer need to step out of the continuous design mindset in order to translate between service design thinking as a process and IT as the means for accomplishing that process.