VOLUME 1

THE PRACTICE OF SYSTEM AND NETWORK ADMINISTRATION

THIRD EDITION

THOMAS A. LIMONCELLI • CHRISTINA J. HOGAN • STRATA R. CHALUP

The Practice of System and Network Administration

Volume 1

Third Edition

This page intentionally left blank

The Practice of System and Network Administration

Volume 1

Third Edition

Thomas A. Limoncelli
Christina J. Hogan
Strata R. Chalup

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the United States, please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Catalog Number: 2016946362

ISBN-13: 978-0-321-91916-8
ISBN-10: 0-321-91916-5
Text printed in the United States of America.

1 16

Contents at a Glance

This page intentionally left blank

Contents

Preface

This is an unusual book. This is not a technical book. It is a book of strategies and frameworks and anecdotes and tacit knowledge accumulated from decades of experience as system administrators.

Junior SAs focus on learning which commands to type and which buttons to click. As you get more advanced, you realize that the bigger challenge is understanding why we do these things and how to organize our work. That's where strategy comes in.

This book gives you a framework—a way of thinking about system administration problems—rather than narrow how-to solutions to particular problems. Given a solid framework, you can solve problems every time they appear, regardless of the operating system (OS), brand of computer, or type of environment. This book is unique because it looks at system administration from this holistic point of view, whereas most other books for SAs focus on how to maintain one particular product. With experience, however, all SAs learn that the big-picture problems and solutions are largely independent of the platform. This book will change the way you approach your work as an SA.

This book is Volume 1 of a series. Volume 1 focuses on enterprise infrastructure, customer support, and management issues. Volume 2, *The Practice of Cloud System Administration* (ISBN: 9780321943187), focuses on web operations and distributed computing.

These books were born from our experiences as SAs in a variety of organizations. We have started new companies. We have helped sites to grow. We have worked at small start-ups and universities, where lack of funding was an issue. We have worked at midsize and large multinationals, where mergers and spin-offs gave rise to strange challenges. We have worked at fast-paced companies that do business on the Internet and where high-availability, high-performance, and scaling issues were the norm. We have worked at slow-paced companies at which "high tech" meant cordless phones. On the surface, these are very different environments with diverse challenges; underneath, they have the same building blocks, and the same fundamental principles apply.

Who Should Read This Book

This book is written for system administrators at all levels who seek a deeper insight into the best practices and strategies available today. It is also useful for managers of system administrators who are trying to understand IT and operations.

Junior SAs will gain insight into the bigger picture of how sites work, what their roles are in the organizations, and how their careers can progress. Intermediate-level SAs will learn how to approach more complex problems, how to improve their sites, and how to make their jobs easier and their customers happier.

Whatever level you are at, this book will help you understand what is behind your day-to-day work, learn the things that you can do now to save time in the future, decide policy, be architects and designers, plan far into the future, negotiate with vendors, and interface with management.

These are the things that senior SAs know and your OS's manual leaves out.

Basic Principles

In this book you will see a number of principles repeated throughout:

- **Automation:** Using software to replace human effort. Automation is critical. We should not be doing tasks; we should be maintaining the system that does tasks for us. Automation improves repeatability and scalability, is key to easing the system administration burden, and eliminates tedious repetitive tasks, giving SAs more time to improve services. Automation starts with getting the process well defined and repeatable, which means documenting it. Then it can be optimized by turning it into code.
- **Small batches:** Doing work in small increments rather than large hunks. Small batches permit us to deliver results faster, with higher quality, and with less stress.
- **End-to-end integration:** Working across teams to achieve the best total result rather than performing local optimizations that may not benefit the greater good. The opposite is to work within your own silo of control, ignoring the larger organization.
- **Self-service systems:** Tools that empower others to work independently, rather than centralizing control to yourself. Shared services should be an enablement platform, not a control structure.
- **Communication:** The right people can solve more problems than hardware or software can. You need to communicate well with other SAs and with your customers. It is your responsibility to initiate communication. Communication ensures that everyone is working toward the same goals. Lack of

communication leaves people concerned and annoyed. Communication also includes documentation. Documentation makes systems easier to support, maintain, and upgrade. Good communication and proper documentation also make it easier to hand off projects and maintenance when you leave or take on a new role.

These principles are universal. They apply at all levels of the system. They apply to physical networks and to computer hardware. They apply to all operating systems running at a site, all protocols used, all software, and all services provided. They apply at universities, nonprofit institutions, government sites, businesses, and Internet service sites.

What Is an SA?

If you asked six system administrators to define their jobs, you would get seven different answers. The job is difficult to define because system administrators do so many things. An SA looks after computers, networks, and the people who use them. An SA may look after hardware, operating systems, software, configurations, applications, or security. An SA influences how effectively other people can or do use their computers and networks.

A system administrator sometimes needs to be a business-process consultant, corporate visionary, janitor, software engineer, electrical engineer, economist, psychiatrist, mindreader, and, occasionally, bartender.

As a result, companies give SAs different titles. Sometimes, they are called network administrators, system architects, system engineers, system programmers, operators, and so on.

This book is for "all of the above."

We have a very general definition of system administrator: one who manages computer and network systems on behalf of another, such as an employer or a client. SAs are the people who make things work and keep it all running.

System Administration Matters

System administration matters because computers and networks matter. Computers are a lot more important than they were years ago.

Software is eating the world. Industry after industry is being taken over by software. Our ability to make, transport, and sell real goods is more dependent on software than on any other single element. Companies that are good at software are beating competitors that aren't.

All this software requires operational expertise to deploy and keep it running. In turn, this expertise is what makes SAs special.

For example, not long ago, manual processes were batch oriented. Expense reports on paper forms were processed once a week. If the clerk who processed them was out for a day, nobody noticed. This arrangement has since been replaced by a computerized system, and employees file their expense reports online, 24/7.

Management now has a more realistic view of computers. Before they had PCs on their desktops, most people's impressions of computers were based on how they were portrayed in films: big, all-knowing, self-sufficient, miracle machines. The more people had direct contact with computers, the more realistic people's expectations became. Now even system administration itself is portrayed in films. The 1993 classic *Jurassic Park* was the first mainstream movie to portray the key role that system administrators play in large systems. The movie also showed how depending on one person is a disaster waiting to happen. IT is a team sport. If only Dennis Nedry had read this book.

In business, nothing is important unless the CEO feels that it is important. The CEO controls funding and sets priorities. CEOs now consider IT to be important. Email was previously for nerds; now CEOs depend on email and notice even brief outages. The massive preparations for Y2K also brought home to CEOs how dependent their organizations have become on computers, how expensive it can be to maintain them, and how quickly a purely technical issue can become a serious threat. Most people do not think that they simply "missed the bullet" during the Y2K change, but rather recognize that problems were avoided thanks to tireless efforts by many people. A CBS Poll shows 63 percent of Americans believe that the time and effort spent fixing potential problems was worth it. A look at the news lineups of all three major network news broadcasts from Monday, January 3, 2000, reflects the same feeling.

Previously, people did not grow up with computers and had to cautiously learn about them and their uses. Now people grow up using computers. They consume social media from their phones (constantly). As a result they have higher expectations of computers when they reach positions of power. The CEOs who were impressed by automatic payroll processing are being replaced by people who grew up sending instant messages all day long. This new wave of management expects to do all business from their phones.

Computers matter more than ever. If computers are to work, and work well, system administration matters. We matter.

Organization of This Book

This book is divided into the following parts:

- **Part I, "Game-Changing Strategies."** This part describes how to make the next big step, for both those who are struggling to keep up with a deluge of work, and those who have everything running smoothly.

- **Part II, "Workstation Fleet Management."** This part covers all aspects of laptops and desktops. It focuses on how to optimize workstation support by treating these machines as mass-produced commodity items.
- **Part III, "Servers."** This part covers server hardware management—from the server strategies you can choose, to what makes a machine a server and what to consider when selecting server hardware.
- **Part IV, "Services."** This part covers designing, building, and launching services, converting users from one service to another, building resilient services, and planning for disaster recovery.
- **Part V, "Infrastructure."** This part focuses on the underlying infrastructure. It covers network architectures and operations, an overview of datacenter strategies, and datacenter operations.
- **Part VI, "Helpdesks and Support."** This part covers everything related to providing excellent customer service, including documentation, how to handle an incident report, and how to approach debugging.
- **Part VII, "Change Processes."** This part covers change management processes and describes how best to manage big and small changes. It also covers optimizing support by centralizing services.
- **Part VIII, "Service Recommendations."** This part takes an in-depth look at what you should consider when setting up some common services. It covers monitoring, nameservices, email, web, printing, storage, backups, and software depositories.
- **Part IX, "Management Practices."** This part is for managers and non-managers. It includes such topics as ethics, organizational structures, perception, visibility, time management, communication, happiness, and hiring and firing SAs.
- **Part X, "Being More Awesome."** This part is essential reading for all managers. It covers how to assess an SA team's performance in a constructive manner, using the Capability Maturity Model to chart the way forward.
- **Part XI, "Appendices."** This part contains two appendices. The first is a checklist of solutions to common situations, and the second is an overview of the positive and negative team roles.

What's New in the Third Edition

The first two editions garnered a lot of positive reviews and buzz. We were honored by the response. However, the passing of time made certain chapters look passé. Most of our bold new ideas are now considered common-sense practices in the industry.

The first edition, which reached bookstores in August 2001, was written mostly in 2000 before Google was a household name and modern computing meant a big Sun multiuser system. Many people did not have Internet access, and the cloud was only in the sky. The second edition was released in July 2007. It smoothed the rough edges and filled some of the major holes, but it was written when DevOps was still in its embryonic form.

The third edition introduces two dozen entirely new chapters and many highly revised chapters; the rest of the chapters were cleaned up and modernized. Longer chapters were split into smaller chapters. All new material has been rewritten to be organized around choosing strategies, and DevOps and SRE practices were introduced where they seem to be the most useful.

If you've read the previous editions and want to focus on what is new or updated, here's where you should look:

- Part I, "Game-Changing Strategies" (Chapters 1–4)
- Part II, "Workstation Fleet Management" (Chapters 5–12)
- Part III, "Servers" (Chapters 13–15)
- Part IV, "Services" (Chapters 16–20 and 22)
- Chapter 23, "Network Architecture," and Chapter 24, "Network Operations"
- Chapter 32, "Change Management"
- Chapter 35, "Centralization Overview," Chapter 36, "Centralization Recommendations," and Chapter 37, "Centralizing a Service"
- Chapter 43, "Data Storage"
- Chapter 45, "Software Repositories," and Chapter 46, "Web Services"
- Chapter 55, "Operational Excellence," and Chapter 56, "Operational Assessments"

Books, like software, always have bugs. For a list of updates, along with news and notes, and even a mailing list you can join, visit our web site:

`www.EverythingSysAdmin.com`

What's Next

Each chapter is self-contained. Feel free to jump around. However, we have carefully ordered the chapters so that they make the most sense if you read the book from start to finish. Either way, we hope that you enjoy the book. We have learned a lot and had a lot of fun writing it. Let's begin.

Thomas A. Limoncelli
Stack Overflow, Inc.
tom@limoncelli.com

Christina J. Hogan
chogan@chogan.com

Strata R. Chalup
Virtual.Net, Inc.
strata@virtual.net

Register your copy of *The Practice of System and Network Administration, Volume 1, Third Edition,* at informit.com for convenient access to downloads, updates, and corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780321919168) and click Submit. Once the process is complete, you will find any available bonus content under "Registered Products."

Acknowledgments

For the Third Edition

Everyone was so generous with their help and support. We have so many people to thank!

Thanks to the people who were extremely generous with their time and gave us extensive feedback and suggestions: Derek J. Balling, Stacey Frye, Peter Grace, John Pellman, Iustin Pop, and John Willis.

Thanks to our friends, co-workers, and industry experts who gave us support, inspiration, and cool stories to use: George Beech, Steve Blair, Kyle Brandt, Greg Bray, Nick Craver, Geoff Dalgas, Michelle Fredette, David Fullerton, Dan Gilmartin, Trey Harris, Jason Harvey, Mark Henderson, Bryan Jen, Gene Kim, Thomas Linkin, Shane Madden, Jim Maurer, Kevin Montrose, Steve Murawski, Xavier Nicollet, Dan O'Boyle, Craig Peterson, Jason Punyon, Mike Rembetsy, Neil Ruston, Jason Shantz, Dagobert Soergel, Kara Sowles, Mike Stoppay, and Joe Youn.

Thanks to our team at Addison-Wesley: Debra Williams Cauley, for her guidance; Michael Thurston, our developmental editor who took this sow's ear and made it into a silk purse; Kim Boedigheimer, who coordinated and kept us on schedule; Lori Hughes, our LaTeX wizard; Julie Nahil, our production editor; Jill Hobbs, our copy editor; and Ted Laux for making our beautiful index!

Last, but not least, thanks and love to our families who suffered for years as we ignored other responsibilities to work on this book. Thank you for understanding! We promise this is our last book. Really!

For the Second Edition

In addition to everyone who helped us with the first edition, the second edition could not have happened without the help and support of Lee Damon, Nathan Dietsch, Benjamin Feen, Stephen Harris, Christine E. Polk, Glenn E. Sieb, Juhani Tali, and many people at the League of Professional System Administrators (LOPSA). Special 73s and 88s to Mike Chalup for love, loyalty, and support, and especially for the mountains of laundry done and oceans of dishes washed so Strata could write. And many cuddles and kisses for baby Joanna Lear for her patience.

Thanks to Lumeta Corporation for giving us permission to publish a second edition.

Thanks to Wingfoot for letting us use its server for our bug-tracking database.

Thanks to Anne Marie Quint for data entry, copyediting, and a lot of great suggestions.

And last, but not least, a big heaping bowl of "couldn't have done it without you" to Mark Taub, Catherine Nolan, Raina Chrobak, and Lara Wysong at Addison-Wesley.

For the First Edition

We can't possibly thank everyone who helped us in some way or another, but that isn't going to stop us from trying. Much of this book was inspired by Kernighan and Pike's *The Practice of Programming* and John Bentley's second edition of *Programming Pearls*.

We are grateful to Global Networking and Computing (GNAC), Synopsys, and Eircom for permitting us to use photographs of their datacenter facilities to illustrate real-life examples of the good practices that we talk about.

We are indebted to the following people for their helpful editing: Valerie Natale, Anne Marie Quint, Josh Simon, and Amara Willey.

The people we have met through USENIX and SAGE and the LISA conferences have been major influences in our lives and careers. We would not be qualified to write this book if we hadn't met the people we did and learned so much from them.

Dozens of people helped us as we wrote this book—some by supplying anecdotes, some by reviewing parts of or the entire book, others by mentoring us during our careers. The only fair way to thank them all is alphabetically and to apologize in advance to anyone whom we left out: Rajeev Agrawala, Al Aho, Jeff Allen, Eric Anderson, Ann Benninger, Eric Berglund, Melissa Binde, Steven Branigan, Sheila Brown-Klinger, Brent Chapman, Bill Cheswick, Lee Damon, Tina Darmohray, Bach Thuoc (Daisy) Davis, R. Drew Davis, Ingo Dean, Arnold de Leon, Jim Dennis, Barbara Dijker, Viktor Dukhovni, Chelle-Marie Ehlers, Michael Erlinger, Paul Evans, Rémy Evard, Lookman Fazal, Robert Fulmer, Carson Gaspar, Paul Glick, David "Zonker" Harris, Katherine "Cappy" Harrison, Jim Hickstein, Sandra Henry-Stocker, Mark Horton, Bill "Whump" Humphries, Tim Hunter, Jeff Jensen, Jennifer Joy, Alan Judge, Christophe Kalt, Scott C. Kennedy, Brian Kernighan, Jim Lambert, Eliot Lear, Steven Levine, Les Lloyd, Ralph Loura, Bryan MacDonald, Sherry McBride, Mark Mellis, Cliff Miller, Hal Miller, Ruth Milner, D. Toby Morrill, Joe Morris, Timothy Murphy, Ravi Narayan, Nils-Peter Nelson, Evi Nemeth, William Ninke, Cat Okita, Jim Paradis, Pat Parseghian, David Parter, Rob Pike, Hal Pomeranz, David Presotto, Doug Reimer, Tommy Reingold, Mike Richichi, Matthew F. Ringel, Dennis Ritchie, Paul D. Rohrigstamper, Ben Rosengart, David Ross, Peter Salus, Scott Schultz, Darren Shaw, Glenn Sieb, Karl Siil, Cicely Smith, Bryan Stansell, Hal Stern, Jay Stiles, Kim Supsinkas, Ken Thompson, Greg Tusar, Kim Wallace, The Rabbit Warren, Dr. Geri Weitzman, Glen Wiley, Pat Wilson, Jim Witthoff, Frank Wojcik, Jay Yu, and Elizabeth Zwicky.

Thanks also to Lumeta Corporation and Lucent Technologies/Bell Labs for their support in writing this book.

Last, but not least, the people at Addison-Wesley made this a particularly great experience for us. In particular, our gratitude extends to Karen Gettman, Mary Hart, and Emily Frey.

This page intentionally left blank

About the Authors

Thomas A. Limoncelli is an internationally recognized author, speaker, and system administrator. During his seven years at Google NYC, he was an SRE for projects such as Blog Search, Ganeti, and internal enterprise IT services. He now works as an SRE at Stack Overflow. His first paid system administration job was as a student at Drew University in 1987, and he has since worked at small and large companies, including AT&T/Lucent Bell Labs and Lumeta. In addition to this book series, he is known for his book *Time Management for System Administrators* (O'Reilly, 2005). His hobbies include grassroots activism, for which his work has been recognized at state and national levels. He lives in New Jersey.

Christina J. Hogan has 20 years of experience in system administration and network engineering, from Silicon Valley to Italy and Switzerland. She has gained experience in small start-ups, midsize tech companies, and large global corporations. She worked as a security consultant for many years; in that role, her customers included eBay, Silicon Graphics, and SystemExperts. In 2005, she and Tom shared the USENIX LISA Outstanding Achievement Award for the first edition of this book. Christina has a bachelor's degree in mathematics, a master's degree in computer science, a doctorate in aeronautical engineering, and a diploma in law. She also worked for six years as an aerodynamicist in a Formula 1 racing team and represented Ireland in the 1988 Chess Olympiad. She lives in Switzerland.

Strata R. Chalup has been leading and managing complex IT projects for many years, serving in roles ranging from project manager to director of operations. She started administering VAX Ultrix and Unisys Unix in 1983 at MIT and spent the dot-com years in Silicon Valley building Internet services for clients like iPlanet and Palm. She joined Google in 2015 as a technical project manager. She has served on the BayLISA and SAGE boards. Her hobbies include being a master gardener and working with new technologies such as Arduino and 2D CAD/CAM devices. She lives in Santa Clara County, California.

This page intentionally left blank

Chapter 3

Pets and Cattle

This chapter is about improving our efficiency by minimizing variation. We mass-produce our work by unifying like things so that they can be treated the same. As a result we have fewer variations to test, easier customer support, and less infrastructure to maintain. We scale ourselves. We can't eliminate all variation, but the more we can unify, the more efficient we can be. Managing the remaining variation is the topic of the next chapter. For now, let's focus on unification itself.

We can't spend hours custom-building every machine we install. Instead, we make our machines generic so that they can all be treated as similarly as possible. Likewise, we are more efficient when we treat related tasks the same way. For example, the process of onboarding new employees usually involves creating accounts and supplying hardware to the new hires. If we invent the process anew with each employee, it not only takes longer but also looks unprofessional as we stumble through improvising each step as the new hires wait. People appreciate a process that is fast, efficient, and well executed.

It is difficult to get better at a process when we never do the same thing more than once. Improvement comes from repetition; practice makes perfect. The more we can consolidate similar things so they can be treated the same, the more practice we get and the better we get at it.

3.1 The Pets and Cattle Analogy

The machines that we administer range from highly customized to entirely generic. The analogy commonly used is "pets and cattle." Pets are the highly customized machines and cattle are the generic machines.

This analogy is generally attributed to Yale computer scientist David Gelernter, who used it in reference to filesystems. Gelernter wrote, "If you have three pet dogs, give them names. If you have 10,000 head of cattle, don't bother."

The analogy gained in popularity when Joshua McKenty, co-founder of Piston Cloud, explained it this way in a press release (McKenty 2013):

> The servers in today's datacenter are like puppies—they've got names and when they get sick, everything grinds to a halt while you nurse them back to health. . . . Piston Enterprise OpenStack is a system for managing your servers like cattle—you number them, and when they get sick and you have to shoot them in the head, the herd can keep moving. It takes a family of three to care for a single puppy, but a few cowboys can drive tens of thousands of cows over great distances, all while drinking whiskey.

A pet is a unique creature. It is an animal that we love and take care of. We take responsibility for its health and well-being. There is a certain level of emotional attachment to it. We learn which food it likes and prepare special meals for it. We celebrate its birthdays and dress it up in cute outfits. If it gets injured, we are sad. When it is ill, we take it to the veterinarian and give it our full attention until it is healed. This individualized care can be expensive. However, since we have only one or two pets, the expense is justified.

Likewise, a machine can be a pet if it is highly customized and requires special procedures for maintaining it.

A herd of cattle is a group of many similar animals. If you have a herd of cows each one is treated the same. This permits us the benefits of mass-production. All cattle receive the same living conditions, the same food, the same medical treatment, the same everything. They all have the same personality, or at least are treated as if they do. There are no cute outfits. The use of mass-production techniques keeps maintenance costs low and improves profits at scale: Saving a dollar per cow can multiply to hundreds of thousands in total savings.

Likewise, machines can be considered cattle when they are similar enough that they can all be managed the same way. This can be done at different levels of abstraction. For example, perhaps the OS is treated generically even though the hardware may comprise any number of virtual or physical machine configurations. Or perhaps the machine hardware, OS, and applications are all the same, but the data they access is different. This is typical in a large web hosting farm, where the only difference is which specific web site is being served by each machine.

Preferably the systems we deal with are fungible resources: Any one unit can substitute for any other.

A related metaphor is the snowflake. A snowflake is even more unique than a pet. It is one of a kind. A system may have started out similar to others, but it was customized, modified, and eventually becomes unlike any other system. Or maybe it started out unique and had very little chance of being properly brought into line with the others. A snowflake requires special operational procedures. Rebooting it requires extra care. Upgrades require special testing. As Martin Fowler (2012) wrote, a snowflake is "good for a ski resort, bad for a datacenter."

A snowflake server is a business risk because it is difficult to reproduce. If the hardware fails or the software becomes corrupted, it would be difficult to build a new machine that provides the same services. It also makes testing more difficult because you cannot guarantee that you have replicated the host in your testing environment. When a bug is found in production that can't be reproduced in the test environment, fixing it becomes much more difficult.

Alternative Analogies

There are other analogies that people use, especially in countries where cattle ranching is less common. One is the analogy of fine porcelain plates and paper plates. You take good care of fine porcelain plates because they are expensive and difficult to replace. In contrast, if a paper plate starts to lose structural integrity, you simply bolster it by putting another paper plate underneath it. If it becomes completely unusable, you replace it.

Another analogy is that modern system administration treats machines like blood cells, not limbs. Blood cells are constantly dying off and being replaced. Limbs, however, are difficult to replace and are protected.

3.2 Scaling

Cattle-like systems give us the ability to grow and shrink our system's scale. In cloud computing a typical architecture pattern has many web server replicas behind a load balancer. Suppose each machine can handle 500 simultaneous users. More replicas are added as more capacity is needed.

Cloud providers such as Amazon Elastic Compute Cloud (Amazon EC2), Google Cloud Platform, and Microsoft Azure have autoscale features where they will spin up and tear down additional replicas as demand requires. This kind of scaling is possible only when machines are cattle. If setting up each new machine required individual attention, the autoscale feature would not be possible.

In such systems we no longer are concerned with the uptime of a particular machine. If one machine fails, the autoscaler will build a new one. If a machine gets sick, we delete it and let the autoscaler do its job. Per-machine uptime was cool in the 1990s but now we measure total system health and availability.

Scale-out architectures are discussed further in Section 16.6.3 and in Volume 2 of this book series.

3.3 Desktops as Cattle

The concept of generic, replaceable machines was first used in desktop environments, long before the cattle and pets analogy was coined. We already discussed the importance of unifying workstation configurations in Chapter 1, "Climbing Out of the Hole," and we'll discuss it in greater detail in Chapter 8, "OS Installation Strategies."

The benefits of generic desktops are manifold. Users benefit from improved customer support, as SAs are no longer struggling to learn and adapt to an infinite number of variations. Repairs happen faster because the IT staff has a single vendor repair procedure to navigate.

Contrast this to an environment where each PC is fully customized. Fixing a software problem is difficult because any change may break something else. It is difficult to know what "working" means when there is no understanding of what is on the machine. Support for older operating systems depends on finding someone on the IT team who remembers that OS.

Creating an environment where cattle are the norm is the primary focus of chapters in Part II, "Workstation Fleet Management," and Part III, "Servers." Chapter 11, "Workstation Standardization," focuses on taking a fleet of workstations that are pets and bringing about unification.

Resetting to a More Uniform State

One of the things that made Apple iPads such a success is that they reset the clock on variation.

PCs had become so customizable that variations had gotten out of control. One of the downsides of competition is that companies compete by differentiating their products, which means making them unique and different. Hardware vendors had many variations and choices, each trying to appeal to different customer segments. Each new market that Microsoft addressed resulted in adding customizability features to attract those users. As a result, by 2005 the complexity of supporting a fleet of Windows machines required a fleet of IT professionals.

Apple iPads took us back to having one particular configuration with curated applications. The uniformity made them more stable and consistent, which then permitted us to focus on the applications, not the infrastructure. Apple retains tight control over the iPad environment so that when the company repeats Microsoft's mistake, it will play out much more slowly.

3.4 Server Hardware as Cattle

Server hardware and software in a datacenter is another situation where we have pets and cattle. At some companies each machine in the datacenter is specified to meet the exact needs of the applications it will run. It has the right amount of RAM and disk, and possibly even additional external storage peripherals or other hardware. Each machine may run a different operating system or OS release. This ensures that each application is maximally optimized to the best of the system administration team's ability.

However, these local optimizations cause inefficiencies at the macro scale. Each machine requires special maintenance procedures. Each operating system in use, and possibly each version of each operating system, requires individual attention. A security patch that must be tested on ten OS versions is a lot more work than one that has to be tested on only one or two versions. This kind of cost eventually outweighs the optimizations one can do for individual applications.

As a result, in large companies it often takes six months or more to deploy a new server in a datacenter. A consultant working at a U.S. bank said it takes 18 months from the initial request to having a working server in their datacenter. If you aren't sure why banks have such lousy interest rates and service, imagine if a phone app you wanted didn't start to run until a year after you bought it.

Contrast this to an environment that has a cattle strategy for its datacenter. Some companies standardize on two or three hardware variations and one or two OS releases. You might not receive the exact hardware you want, but you receive it quickly. Perfect is the enemy of good: Would you rather be up and running this week with hardware that is good enough, or wait a year and have the exact hardware you dreamed of, which is now obsolete?

Case Study: Google's Two Hardware Types

For many years Google standardized on two types of machines. Diskful machines maximized the amount of hard disk storage that could be packed into a single machine. Index machines (so called because they stored the search index) maximized the amount of RAM that could fit in a 1U configuration. Teams that requested machines in the datacenter could choose between one or the other and receive them within minutes because they were preloaded and ready for use.

This setup made handling future orders easier. The purchase department collected orders from all teams and tallied the number of diskful and index machines requested. This was considerably easier than if each month department members had to manage requests for thousands of different bespoke configurations.

Software was designed to fit best with one model or the other. Most services were big enough that they required many (often thousands) machines, some of each type. For example, an application would be split to run the application's web frontend on index machines while storing application data on diskful machines. If the hardware being offered was 10 percent slower than the ideal machine, employees would simply request additional machines to compensate for the lagging performance.

This evolved into a pattern that was, actually, the opposite. Engineers didn't think in terms of spec'ing out the perfect machine. Instead, they designed applications such that scaling was done by adding a certain number of machines per unit of workload. They performed tests to see how many machines (of the type currently offered) would be required to process the number of users or the workload expected. Employees then could request that number of machines. They no longer thought of applications in terms of which machine would be best suited to a particular application, but rather how much generic capacity was required. As faster models were introduced into the datacenter, benchmarks would be run to develop new capacity planning models and the process would repeat.

Not every environment can standardize down to one machine type, but we can provide a few standard configurations (small, medium, and large) and guide people to them. We can minimize the number of vendors, so that there is one firmware upgrade process, one repair workflow, and so on.

Offering fixed sizes of virtual machines (VMs) results in less isolated or stranded capacity. For example, we can make the default VM size such that eight fit on a physical machine with no waste. We can offer larger sizes that are multiples of the default size. This means we are never left with a physical machine that has unused capacity that is too small for a new machine. It also makes it easier to plan future capacity and reorganize placement of existing VMs within a cluster.

By offering standardized sizes we enable an environment where we no longer look at machines individually, but rather treat them as scaling units to be used when sizing our deployments. This is a better fit for how distributed computing applications are designed and how most applications will be built in the future.

We can also standardize at the software level. Each machine is delivered to the user with a standard OS installation and configuration. The defaults embody the best practices we wish all users would follow. Modifications made after that are the application administrator's responsibility. We'll discuss better ways to handle this responsibility in the next chapter.

The Power of Defaults

Defaults are powerful. If you announce an OS configuration change that all IT subteams are required to make, you'll get angry push-back from your loudest and most vocal co-workers. You will get very little participation. In fact, there may be enough push-back that you withdraw the request. Often a tyranny of a few loud complainers prevents the majority from receiving a beneficial change.

In contrast, if you make that change or setting part of the default configuration that is delivered with each new server (thanks to your automated OS install), you may be surprised at how little noise it generates. Most people will live with the change. The people who previously would have made noise will still complain, but now you can work with them to address their concerns. See the anecdote in Section 7.3.1.

3.5 Pets Store State

Another way of describing pets is to note that they contain a lot of irreproducible state. Cattle are stateless, or contain only reproducible state.

State is, essentially, data or information. That information may be data files, configuration, or status. For example, when running MS Excel, the spreadsheet currently loaded is the state. In a video game, the player's score, position, and status are state. In a web-based application, there is the application itself plus the database that is used to store the user's data. That database is state.

The more state a machine holds, the more irreplaceable it is—that is, the more pet-like it is. Cattle are generic because we can rebuild one easily thanks to the fact that cattle contain no state, or only state that can be copied from elsewhere.

A web server that displays static content (web pages and images) is stateless if that static content is a copy from a master stored elsewhere. The web server can be wiped and reloaded, but as long as the content can be copied from the primary source, the new server is functionally the same as the original.

But suppose a web application has a database. If the machine is wiped and reloaded, the database is lost. We can restore it from backups, but then we will have lost any new data accumulated since the last backup was done. This web application is stateful.

Configuration data is also state, but it can usually be regenerated. Which software packages were installed and how they were configured are state, even though the contents of the software packages themselves are not state; they come from a master repository. The state can be reproduced either manually or via automation.

Irreproducible configuration state can be particularly insidious. In this case the state is not a particular configuration file but rather how the system was made that makes it a snowflake server. We've seen important servers that could be rebuilt only by installing an old version of the software and then installing an upgrade package; installing the final version directly did not work. Unknown and unidentifiable state was being generated during the upgrade process that somehow was not reproduced via the direct installation. This is the kind of unexplained state that makes you want to cry.

Irreproducible Laptops

When Tom arrived at Cibernet, the company depended on an application that had been installed on a set of laptops many years ago. By then, no one working there could figure out which combination of Windows release, patches, and installation packages would create a new laptop with a working version of the software. Each time one of the original laptops died, the company moved one step closer to insolvency.

The company was in the process of creating a replacement for the software. If the new software was ready before the last laptop died, the company would survive. If not, the company would literally not be able to perform the financial processing it did for customers. It would have to go out of business. One of the laptops was kept in a safe as a precaution. The others were used carefully and only when needed.

When there were only four working laptops remaining, VMware introduced a product that took a snapshot of a physical hard drive and created a virtual machine image (physical to virtual, or p2v). Luckily it worked and soon a virtual laptop could be run on any other machine. This reduced the risk of the replacement project being late, and probably saved the company.

3.6 Isolating State

We can turn pets into cattle by isolating the state. Optimally this is done during the design process, but sometimes we find ourselves doing it after the fact.

Imagine a typical web application running entirely on a single machine. The machine includes the Apache HTTP server, the application software, a MariaDB database server, and the data that the database is storing. This is the architecture used by many small web-based applications.

The problem with this architecture is that the single machine stores both the software and the state. It is a pet. This situation is depicted in Figure 3.1a.

We can improve the situation by separating out the database. As depicted in Figure 3.1b, we can move the MariaDB database software and the data it stores to another machine. The web server is now cattle-like because it can be reproduced easily by simply installing the software and configuring it to point to the database on the other machine. The database machine is a pet. However, having a cattle + pet situation is an improvement over having one big pet. If the cattle-like server becomes sick, we can easily replace it. The pet, since it has a single function, can be more easily backed up to prepare for an emergency. We can also lock out users so there is less chance of human-caused problems, and we can use more reliable (and more expensive) hardware. By identifying and isolating the state, we are putting all our eggs in one basket, but we can make it a very good basket—one to which we give special care and attention.

The state that remains is the data stored in the database. We can move this data to an external storage to further isolate the state. For example, rather than storing the data on local disk, we can allocate a data volume on our storage area network (SAN) server, as depicted in Figure 3.1c. Now the database machine is stateless.

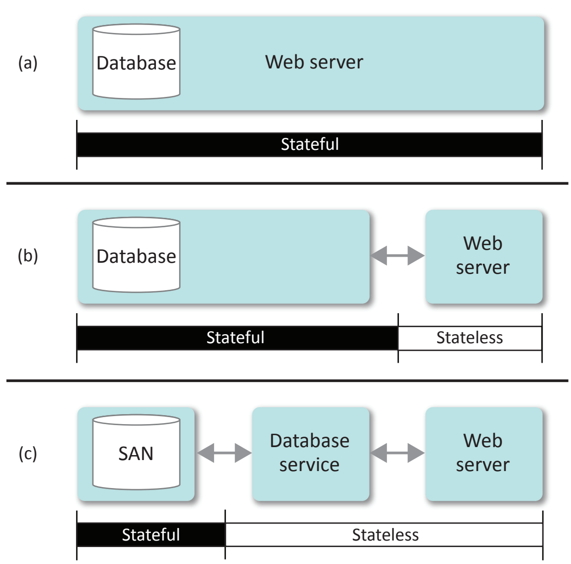

Figure 3.1: Evolving a web service to isolate state

It can be wiped and reloaded without losing the data. It is simply configured to attach to the right SAN volume to access the state.

Many systems go through this kind of evolution. Sometimes these evolutions happen during the design stage, resulting in a design that isolates state or minimizes the number of places in which state is stored. For example, we might consolidate state into a single database instead of storing some in a SQL database, some in local files, and some in an external application service. At other times this kind of evolution happens after the fact. System administrators spend a lot of time reconfiguring and reengineering older systems to evolve them as needed, often because they were designed by predecessors who have not read this book. Lucky you.

This process is also called decoupling state. The all-in-one design tightly couples the application to the data. The last design decouples the data from the software entirely. This decoupling permits us to scale the service better. For example, the web server can be replicated to add more capacity.

Decoupling state makes it easier to scale systems. Many scaling techniques involve replicating services and dividing the workload among those replicas. When designing a system, it is generally easier to replicate components that are stateless. If we administer these components as cattle, we can easily generate and destroy them as demand increases and decreases. Figure 3.2 is similar to Figure 3.1c, but the web server component has been replicated to scale front-end capacity. A replicated database cache was added to off-load read-only queries, improving database performance. This kind of scaling is discussed further in Chapter 18, "Service Resiliency and Performance Patterns."

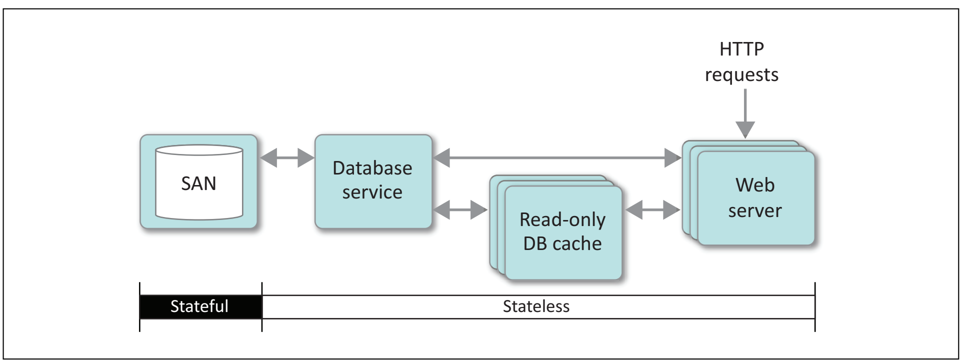

Figure 3.2: A scalable web application service

Blogs and State

State can also be moved to external services. Originally blog platforms were made up of software that generated each page on demand by reading data from locally stored files and an SQL database. This meant state was in three places (the software, the SQL server, and local files). Scaling such systems is very difficult.

In 2016, a new generation of blogging platforms arrived that required no server-side state. In this case, the site was a set of static files that could be uploaded to any web server—even ones without a database or the ability to execute code. Such platforms used client-side JavaScript for all interactive features.

Blog site generators like Hugo and Jekyll typically work as follows. The blog owner creates a Git file repository that stores everything related to the site: images, the text of blog posts, metadata that describes what the web site should look like, and so on. The site generator uses this information to generate the entire site as a set of static files. These files are uploaded to a web server. If a new blog post is created in the Git repository, the entire site is regenerated and uploaded again to the web host.

Highly stateful content such as user comments is handled by external services such as Disqus. While the comments appear to be dynamically updating on the site, they are really loading from the Disqus servers using HTML5 code that does not change. This eliminates most of the infrastructure the blog owner must maintain.

Because the files are static and require no server-side state, they can be served from nearly anywhere. This includes a directory on a file server, a Dropbox account, or a massive multiserver web hosting infrastructure.

3.7 Generic Processes

We can also make processes more generic to improve efficiency. For example, onboarding new employees is a complex process. In some companies each division or team has a different onboarding process. In some places engineers have a different onboarding process than non-engineers. Each of these processes is petlike. It takes extra effort to reinvent each process again and again. Improvements made for one process may not propagate to the others. However, this situation often arises because different teams or departments do not communicate.

In contrast, some companies have a unified onboarding process. The common aspects such as paperwork and new employee training are done first. The variations required for different departments or roles are saved to the end. You would think this is a no-brainer and every company would do this, but you'd be surprised at how many companies, both large and small, have a pet-like onboarding process, or unified the process only after losing money due to a compliance failure that required the company to clean up its act.

Onboarding

Onboarding is the process by which a new employee is brought into the company. While it is not usually the responsibility of the IT team, much of the process involves IT: creating accounts; delivering the employee's computer, phone, and other technology; and so on. See Chapter 12, "Onboarding."

Another example is the process for launching and updating applications in production. Large companies often have hundreds of internal and external applications. A retailer like Target has thousands of applications ranging from inventory management to shipping and logistics, forecasting, electronic data interchange (EDI), and the software that handles the surprisingly complex task of generating price labels.

In many organizations each such application has been built using different software technologies, languages, and frameworks. Some are written in Java; others in Go, Python, or PHP. One requires a particular web framework. Another requires a particular version of an operating system. One requires a certain OS patch; another won't work on a machine *with* that patch. Some are delivered as an installable package; with others the developer emails a ZIP file to the system administrators.

As a result the process of deploying these applications in production is very complex. Each new software release requires the operations team to follow a unique or bespoke process. In some cases the process is full of new and different surprises each time, often based on which developer led that particular release. Joe sends ZIP files; Mary sends RAR files. Each variation requires additional work and additional knowledge, and adds complexity and risk to the production environment. Each variation makes automation more difficult.

In other words, each of these processes is a pet. So how can we turn them into cattle?

Around 2012 a number of organizations identified the need to unify these processes. Many new technologies appeared, one of which was the Docker Container format. It is a format for software distribution that also unifies how production environments deploy applications. This format not only includes all the files required for an application or service, but also includes a standard way to connect and control them. Docker Containers includes meta-information such as which TCP port the service runs on. As a consequence, in a service hosting environment nearly all applications can be deployed the same way. While not every application can work in the Docker Container system, enough can to greatly reduce the number of pets in the environment.

The Docker system includes a number of elements. The Docker Container image is an archive file (like ZIP or TAR) that includes all the files required for a particular service. A Dockerfile is a file that describes how to build an image in an automated fashion, thereby enabling a repeatable process for building images. A Docker compose file defines a complex application made up of many containers, and describes how they talk to each other.

Listing 3.1 is a Dockerfile that describes how to create an image that includes the Apache HTTP server and related files. The `EXPOSE 80` statement indicates that the software this image runs needs exclusive access to TCP port 80.

Listing 3.1: A Dockerfile describing how to build a Docker image

```
FROM ubuntu:12.04

RUN apt-get update && apt-get install -y apache2 \
    && apt-get clean && rm -rf /var/lib/apt/lists/*

ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
ENV APACHE_LOG_DIR /var/log/apache2

EXPOSE 80

CMD ["/usr/sbin/apache2", "-D", "FOREGROUND"]
```

Listing 3.2 shows a Docker compose file for an application that consists of two services: one that provides a web-based application and another that provides API access. Both require access to a MySQL database and a Redis cache.

Listing 3.2: A Docker compose file for a simple application

```
services:
  web:
    git_url: git@github.com:example/node-js-sample.git
    git_branch: test
    command: rackup -p 3000
```

```
    build_command: rake db:migrate
    deploy_command: rake db:migrate
    log_folder: /usr/src/app/log
    ports: ["3000:80:443", "4000"]
    volumes: ["/tmp:/tmp/mnt_folder"]
    health: default
  api:
    image: quay.io/example/node
    command: node test.js
    ports: ["1337:8080"]
    requires: ["web"]
databases:
  - "mysql"
  - "redis"
```

With a standardized container format, all applications can be delivered to production in a form so sufficiently self-contained that IT doesn't need to have a different procedure for each application. While each one is wildly different internally, the process that IT follows to deploy, start, and stop the application is the same.

Containers can be used to build a beta environment. Ideally, the test environment will be as similar to the production environment as possible. Anytime a bug is found in production, it must be reproduced in the test environment to be investigated and fixed. Sometimes a bug can't be reproduced this way, and fixing it becomes much more difficult.

The reality is that at most companies the beta and production environments are very different: Each is built by a different group of people (developers and SAs) for their own purposes. A story we hear time and time again is that the developers who started the project wrote code that deploys to the beta environment. The SAs were not involved in the project at the time. When it came time to deploy the application into production, the SAs did it manually because the deployment code for the beta environment was unusable anywhere else. Later, if the SAs automated their process, they did it in a different language and made it specific to the production environment. Now two code bases are maintained, and changes to the process must be implemented in code twice. Or, more likely, the changes are silently made to the beta deploy code, and no one realizes it until the next production deployment breaks. This sounds like a silly company that is the exception, but it is how a surprisingly large number of teams operate.

Not only do containers unify the production environment and make it more cattle-like, but they also improve developer productivity. Developers can build a sandbox environment on their personal workstations by selecting the right combination of containers. They can create a mini-version of the production environment that they can use to develop against. Having all this on their laptops is better than sharing or waiting their turn to use a centrally administered test stage environment.

In June 2015 the Open Container Initiative (OCI) was formed to create a single industry-wide standard for container formats and run-times. Docker, Inc., donated its container format and runtime to serve as the basis of this effort.

Containers are just one of many methods for unifying this process.

Shipping Containers

The concept of Docker Containers comes from the shipping industry. Before shipping containers were introduced, individual items were loaded and unloaded from ships, usually by hand. Each item had different dimensions and therefore had to be handled differently. An individual lamp needed to be carefully handled, while a large sack of wheat could be tossed about.

That changed in April 1956, when Malcom McLeans organized the first shipment using standardized containers.

Standardized shipping containers revolutionized how products move around the world. Because each shipping container was the same shape and size, loading and unloading could be done much faster. Cranes and automation had to be built to handle only one shape, with a standardized maximum weight and lifting points.

A single container held many individual items, all with the same destination. Customs officials could approve all the items in a particular container and seal it, eliminating the need for customs checks at transit points as long as the seal remained unbroken.

Intermodal shipping was born. A single container would be loaded at a factory and remain as a unit whether it was on a truck, train, or ship. Standard shipping containers are accepted everywhere.

3.8 Moving Variations to the End

Operational science teaches us to move variations in a process to the end. Burger King restaurants make a generic hamburger, waiting until the last minute to add toppings such as ketchup, mustard, and pickles. Unsold inventory can be kept generic so that it can be quickly customized when the order is placed. Otherwise, a restaurant might end up with a surplus of burgers with pickles sitting unsold while Christina waits for her pickle-less order to be made from scratch.

Auto manufacturers also delay variation to the last possible moment. Option packages are added at the end, where demand is better understood. Unusual items like special audio systems or fancy tires are added by the dealer only after a particular customer requests them.

As long as the WIP stays generic, the process is simple and easier to streamline. You can mass-produce dozens of generic burgers with a single process and a single focus, improving it constantly to be more efficient. Once they are customized, everything becomes a special snowflake process. Our ability to improve the process is not impossible, though it is deterred.

This strategy also works in IT. Design systems and processes to keep WIP generic for as long as possible. Save variations until the end. This reduces the combinations of configurations and variables to be tested, makes it easier to verify completeness and accuracy, and makes it easier to improve the process.

We've already seen this in our discussion of the onboarding process, where common tasks were done first.

Another example relates to laptop distribution. Imagine a company where all new employees receive the same laptop, with the same OS, the same configuration, and the same applications. However, when a user logs in for the first time, specific applications are installed depending on whether the employee is an engineer, salesperson, or executive. After that customers can customize the workstation to their liking. This enables the entire laptop deployment process to be generic until the last possible moment.

Now imagine instead that such customizations were done at the start. If there was a burst of new engineers starting at the company, the IT department might find itself with no engineering laptops left but plenty of sales laptops. If the hardware was the same they could at least rework the laptops to be engineering laptops. This would double the effort expended on each laptop, but it would solve the immediate problem. If the hardware models were different, however, the engineers would have to wait for laptops since the units are not fungible resources. Alternatively, the engineers could be retrained to work as salespeople, but that would be silly since people are not fungible resources.

When things are different in software, we can treat them generically by choosing the right level of abstraction. Containers permit all services to be treated generically because no matter what is on the inside, the SAs can simply deal with them at generic touch points that are common for all.

Some software frameworks permit plug-ins or drivers to be written so that the framework deals with generic "things" but the differences are mediated by the plug-in.

3.9 Automation

Consistency makes it easier to automate a process. It is easier to write automation for cattle than for pets because there are fewer surprises and variations to be aware of and fewer permutations to test. Automation brings about opportunities for self-service system administration. Web sites and other tools can empower users to get their needs met without human intervention.

You can also look at this another way: Before we can improve things, we must make things consistent. Making improvements to something inconsistent is like wrestling a pig: It's messy and you probably won't win. Once things are consistent, we can make them better—optimize them—and we gain the freedom to experiment and try new things. Our experiments may fail, but if we do not try, there is no way to improve. At least with each failure we learn something. This is not a rationalization that makes us feel better about our failures: The experiments that are a success are valuable because the system has been improved (optimized); the experiments we revert are learning experiences that guide us as we make future improvements.

You'll see this pattern of chaos⇒defined⇒repeatable⇒optimizing throughout this book. It is also the basis of "The Three Ways of Operational Improvement" described in Section 12.3.5, and is the basis of the assessment levels in Section 55.3.2.

3.10 Summary

Pets are machines that are irreproducible because they are highly customized over a long period of time with no record of how to exactly replicate the process. They must be managed individually. If a pet becomes damaged or corrupted, it must be carefully brought back into the desired state just as a doctor tends to a sick patient.

Cattle are machines that can be reproduced programmatically and are therefore disposable. If one of these cattle gets damaged or corrupted, it is wiped and rebuilt. To complete the analogy, when a single animal in a cattle drive is sick, it is killed so that the herd can keep moving.

Cattle-like systems make it easier to manage large numbers of machines. It is easier to mass-produce IT when machines are generic.

Desktops can be made cattle-like by starting them all the same via automation, and using directory services and other techniques to maintain their sameness. We can also reduce the number of vendors and models to make the repair processes more generic.

Servers have different challenges. The software each runs is usually very different. We can use containers and configuration management systems to automate the setup of these differences so that they can be reproduced by running the code again. More importantly, pet-like servers store irreproducible state: information that is not stored elsewhere (other than backups). We can design our services to separate out our state to specific machines so as to increase the number of cattle-like systems. State can be stored on a separate file server, database server, or external service.

We can also improve efficiency by making processes more cattle-like. A process should save any variations until the last possible moment. By keeping things generic at the start, we can mass-produce the start of the process.

Exercises

1. Explain the pets and cattle analogy for computers.
2. What is a snowflake server? Why are they a bad idea?
3. If a snowflake server is risky, how can we reduce risk through repetition?
4. How do cattle-like systems help us be more efficient?
5. How do cattle-like systems help us scale services?
6. According to this chapter, why do banks have lousy interest rates?
7. A laptop and a desktop PC are very different. In what way could we treat them both as cattle of the same herd?
8. What is state? What is irreproducible state?
9. Why is isolating state to particular machines a good thing?
10. How can beta and production environments end up being different? How can we make them as similar as possible?
11. How is mass-production aided by moving variations to the end?
12. Sometimes bad customer service is described as being treated like cattle. Yet, some of the best companies have practices that assure that everyone receives extremely high-quality service in an efficient and mass-produced way. These companies are also managing people like cattle. How are the latter companies able to achieve this without offending their customers?
13. Pick a service in your organization that stores a lot of state. Describe how it could be implemented using an architecture that isolates state.
14. What are the benefits of moving variations to the end of the process?
15. Pick a process in your organization that has a lot of variation. How can it be restructured to move the variation to the end? What benefits would be achieved by doing this?