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

The Practice of System and Network Administration

DevOps and Other Best Practices for Enterprise IT

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

Library of Congress Catalog Number: 2016946362

ISBN-13: 978-0-321-91916-8
ISBN-10: 0-321-91916-5

4 17

Contents at a Glance

Contents

Chapter 2

The Small Batches Principle

One of the themes you will see in this book is the small batches principle: It is better to do work in small batches than in big leaps. Small batches permit us to deliver results faster, with higher quality and less stress.

This chapter begins with an example that has nothing to do with system administration to demonstrate the general idea. Then it focuses on three IT-specific examples to show how the method applies and the benefits that follow.

The small batches principle is part of the DevOps methodology. It comes from the Lean Manufacturing movement, which is often called just-in-time (JIT) manufacturing. It can be applied to just about any kind of process that you do frequently. It also enables the minimum viable product (MVP) methodology, which involves launching a small version of a service to get early feedback that informs the decisions made later in the project.

2.1 The Carpenter Analogy

Imagine a carpenter who needs 50 two-by-fours, all the same length.

One could imagine cutting all 50 boards and then measuring them to verify that they are all the correct size. It would be very disappointing to discover that the blade shifted while making board 10, and boards 11 through 50 are now unusable. The carpenter would have to cut 40 new boards. How embarrassing!

A better method would be to verify the length after each board is cut. If the blade has shifted, the carpenter will detect the problem soon after it happened, and there would be less waste.

These two approaches demonstrate big batches versus small batches. In the big-batch world, the work is done in two large batches: The carpenter cuts all the boards, then inspects all the boards. In the small-batch world, there are many iterations of the entire process: cut and inspect, cut and inspect, cut and inspect ….

One benefit of the small-batch approach is less waste. Because an error or defect is caught immediately, the problem can be fixed before it affects other parts.

A less obvious benefit is latency. At the construction site there is a second team of carpenters who use the boards to build a house. The boards cannot be used until they are inspected. With the first method, the second team cannot begin its work until all the boards are cut and at least one is inspected. The chances are high that the boards will be delivered in a big batch after they have all been inspected. With the small-batch method, the new boards are delivered without this delay.

The sections that follow relate the small batches principle to system administration and show many benefits beyond reduced waste and improved latency.

2.2 Fixing Hell Month

A company had a team of software developers who produced a new release every six months. When a release shipped, the operations team stopped everything and deployed the release into production. The process took three or four weeks and was very stressful for all involved. Scheduling the maintenance window required complex negotiation. Testing each release was complex and required all hands on deck. The actual software installation never worked on the first try. Once it was deployed, a number of high-priority bugs would be discovered, and each would be fixed by various "hot patches" that would follow.

Even though the deployment process was labor intensive, there was no attempt to automate it. The team had many rationalizations that justified this omission. The production infrastructure changed significantly between releases, making each release a moving target. It was believed that any automation would be useless by the next release because each release's installation instructions were shockingly different. With each next release being so far away, there was always a more important "burning issue" that had to be worked on first. Thus, those who did want to automate the process were told to wait until tomorrow, and tomorrow never came. Lastly, everyone secretly hoped that maybe, just maybe, the next release cycle wouldn't be so bad. Such optimism is a triumph of hope over experience.

Each release was a stressful, painful month for all involved. Soon it was known as Hell Month. To make matters worse, each release was usually late. This made it impossible for the operations team to plan ahead. In particular, it was difficult to schedule any vacation time, which just made everyone more stressed and unhappy.

Feeling compassion for the team's woes, someone proposed that releases should be done less often, perhaps every 9 or 12 months. If something is painful, it is natural to want to do it less frequently.

To everyone's surprise the operations team suggested going in the other direction: monthly releases. This was a big-batch situation. To improve, the company didn't need bigger batches, it needed smaller ones.

People were shocked! Were they proposing that every month be Hell Month? No, by doing it more frequently, there would be pressure to automate the process. If something happens infrequently, there's less urgency to automate it, and we procrastinate. Also, there would be fewer changes to the infrastructure between each release. If an infrastructure change did break the release automation, it would be easier to fix the problem.

The change did not happen overnight. First the developers changed their methodology from mega-releases, with many new features, to small iterations, each with a few specific new features. This was a big change, and selling the idea to the team and management was a long process.

Meanwhile, the operations team automated the testing and deployment processes. The automation could take the latest code, test it, and deploy it into the beta-test area in less than an hour. Pushing code to production was still manual, but by reusing code for the beta rollouts it became increasingly less manual over time.

The result was that the beta area was updated multiple times a day. Since it was automated, there was little reason not to. This made the process continuous, instead of periodic. Each code change triggered the full testing suite, and problems were found in minutes rather than in months.

Pushes to the production area happened monthly because they required coordination among engineering, marketing, sales, customer support, and other groups. That said, all of these teams loved the transition from an unreliable hopefully every-six-months schedule to a reliable monthly schedule. Soon these teams started initiatives to attempt weekly releases, with hopes of moving to daily releases. In the new small-batch world, the following benefits were observed:

- **Features arrived faster.** Where in the past a new feature took up to six months to reach production, now it could go from idea to production in days.
- **Hell Month was eliminated.** After hundreds of trouble-free pushes to beta, pushing to production was easier than ever.
- **The operations team could focus on higher-priority projects.** The operations team was no longer directly involved in software releases other than fixing the automation, which was rare. This freed up the team for more important projects.
- **There were fewer impediments to fixing bugs.** The first step in fixing a bug is to identify which code change is responsible. Big-batch releases had hundreds or thousands of changes to sort through to identify the guilty party. With small batches, it was usually quite obvious where to find the bug.
- **Bugs were fixed in less time.** Fixing a bug in code that was written six months ago is much more difficult than if the code is still fresh in your mind. Small

batches meant bugs were reported soon after the code was written, which meant developers could fix it more expertly in a shorter amount of time.

- **Developers experienced instant gratification.** Waiting six months to see the results of your efforts is demoralizing. Seeing your code help people shortly after it was written is addictive.
- **Everyone was less stressed.** Most importantly, the operations team could finally take long vacations, the kind that require advance planning and scheduling, thus giving them a way to reset and live healthier lives.

While these technical benefits were worthwhile, the business benefits were even more exciting:

- **Improved ability to compete:** Confidence in the ability to add features and fix bugs led to the company becoming more aggressive about new features and fine-tuning existing ones. Customers noticed and sales improved.
- **Fewer missed opportunities:** The sales team had been turning away business due to the company's inability to strike fast and take advantage of opportunities as they arrived. Now the company could enter markets it hadn't previously imagined.
- **A culture of automation and optimization:** Rapid releases removed common excuses not to automate. New automation brought consistency, repeatability, and better error checking, and required less manual labor. Plus, automation could run anytime, not just when the operations team was available.

The ability to do rapid releases is often called a DevOps strategy. In Chapter 20, "Service Launch: DevOps," you'll see similar strategies applied to third-party software.

The Inner and Outer Release Loops

You can think of this process as two nested loops. The inner loop is the code changes done one at a time. The outer loop is the releases that move these changes to production.

2.3 Improving Emergency Failovers

Stack Overflow's main web site infrastructure is in a datacenter in New York City. If the datacenter fails or needs to be taken down for maintenance, duplicate equipment and software are running in Colorado. The duplicate in Colorado is a running and functional copy, except that it is in stand-by mode waiting to be activated. Database updates in NYC are replicated to Colorado. A planned switch to Colorado will result in no lost data. In the event of an unplanned failover—for example, as the result of a power outage—the system will lose an acceptably small quantity of updates.

The failover process is complex. Database masters need to be transitioned. Services need to be reconfigured. It takes a long time and requires skills from four different teams. Every time the process happens, it fails in new and exciting ways, requiring ad hoc solutions invented by whoever is doing the procedure.

In other words, the failover process is risky. When Tom was hired at Stack Overflow, his first thought was, "I hope I'm not on call when we have that kind of emergency."

Drunk driving is risky, so we avoid doing it. Failovers are risky, so we should avoid them, too. Right?

Wrong. There is a difference between behavior and process. Risky *behaviors* are inherently risky; they cannot be made less risky. Drunk driving is a risky *behavior.* It cannot be done safely, only avoided. A failover is a risky *process.* A risky *process* can be made less risky by doing it more often.

The next time a failover was attempted at Stack Overflow, it took ten hours. The infrastructure in New York had diverged from Colorado significantly. Code that was supposed to seamlessly fail over had been tested only in isolation and failed when used in a real environment. Unexpected dependencies were discovered, in some cases creating Catch-22 situations that had to be resolved in the heat of the moment.

This ten-hour ordeal was the result of big batches. Because failovers happened rarely, there was an accumulation of infrastructure skew, dependencies, and stale code. There was also an accumulation of ignorance: New hires had never experienced the process; others had fallen out of practice.

To fix this problem the team decided to do more failovers. The batch size was based on the number of accumulated changes and other things that led to problems during a failover. Rather than let the batch size grow and grow, the team decided to keep it small. Rather than waiting for the next real disaster to exercise the failover process, they would introduce simulated disasters.

The concept of activating the failover procedure on a system that was working perfectly might seem odd, but it is better to discover bugs and other problems in a controlled situation rather than during an emergency. Discovering a bug during an emergency at 4 AM is troublesome because those who can fix it may be unavailable—and if they are available, they're certainly unhappy to be awakened. In other words, it is better to discover a problem on Saturday at 10 AM when everyone is awake, available, and presumably sober.

If schoolchildren can do fire drills once a month, certainly system administrators can practice failovers a few times a year. The team began doing failover drills every two months until the process was perfected.

Each drill surfaced problems with code, documentation, and procedures. Each issue was filed as a bug and was fixed before the next drill. The next failover took five hours, then two hours, then eventually the drills could be done in an hour with no user-visible downtime.

The drills found infrastructure changes that had not been replicated in Colorado and code that didn't fail over properly. They identified new services that hadn't been engineered for smooth failover. They discovered a process that could be done by one particular engineer. If he was on vacation or unavailable, the company would be in trouble. He was a single point of failure.

Over the course of a year all these issues were fixed. Code was changed, better pretests were developed, and drills gave each member of the SRE (site reliability engineering) team a chance to learn the process. Eventually the overall process was simplified and easier to automate. The benefits Stack Overflow observed included

- **Fewer surprises:** More frequent the drills made the process smoother.
- **Reduced risk:** The procedure was more reliable because there were fewer hidden bugs waiting to bite.
- **Higher confidence:** The company had more confidence in the process, which meant the team could now focus on more important issues.
- **Speedier bug fixes:** The smaller accumulation of infrastructure and code changes meant each drill tested fewer changes. Bugs were easier to identify and faster to fix.
- **Less stressful debugging:** Bugs were more frequently fixed during business hours. Instead of having to find workarounds or implement fixes at odd hours when engineers were sleepy, they were worked on during the day when engineers were there to discuss and implement higher-quality fixes.
- **Better cross-training:** Practice makes perfect. Operations team members all had a turn at doing the process in an environment where they had help readily available. No person was a single point of failure.
- **Improved process documentation and automation:** The team improved documentation in real time as the drill was in progress. Automation was easier to write because the repetition helped the team see what could be automated and which pieces were most worth automating.
- **Revealed opportunities:** The drills were a big source of inspiration for big-picture projects that would radically improve operations.
- **Happier developers:** There was less chance of being woken up at odd hours.

- **Happier operations team:** The fear of failovers was reduced, leading to less stress. More people trained in the failover procedure meant less stress on the people who had previously been single points of failure.
- **Better morale:** Employees could schedule long vacations again.

Google's Forced Downtime

Google's internal lock service is called Chubby; an open source clone is called Zookeeper. Chubby's uptime was so perfect that engineers designing systems that relied on Chubby starting writing code that assumed Chubby could not go down. This led to cascading failures when Chubby did have an outage.

To solve this problem, Google management decreed that if Chubby had zero downtime in a given month, it would be taken down intentionally for five minutes. This would assure that error handling code was exercised regularly.

Developers were given three months' warning, yet the first "purposeful outage" was postponed when a team came forward to beg for an extension. One was granted, but since then Chubby has been down for five minutes every month to exercise the failure-related code.

2.4 Launching Early and Often

An IT department needed a monitoring system. The number of servers had grown to the point where situational awareness was no longer possible by manual means. The lack of visibility into the company's own network meant that outages were often first reported by customers, and often after the outage had been going on for hours and sometimes days.

The system administration team had a big vision for what the new monitoring system would be like. All services and networks would be monitored, the monitoring system would run on a pair of big, beefy machines, and when problems were detected a sophisticated oncall schedule would be used to determine whom to notify.

Six months into the project they had no monitoring system. The team was caught in endless debates over every design decision: monitoring strategy, how to monitor certain services, how the pager rotation would be handled, and so on. The hardware cost alone was high enough to require multiple levels of approval.

Logically the monitoring system couldn't be built until the planning was done, but sadly it looked like the planning would never end. The more the plans were discussed, the more issues were raised that needed to be discussed. The longer the planning lasted, the less likely the project would come to fruition.

Fundamentally they were having a big-batch problem. They wanted to build the perfect monitoring system in one big batch. This is unrealistic.

The team adopted a new strategy: small batches. Rather than building the perfect system, they would build a small system and evolve it.

At each step they would be able to show it to their co-workers and customers to get feedback. They could validate assumptions for real, finally putting a stop to the endless debates the requirements documents were producing. By monitoring something—anything—they would learn the reality of what worked best.

Small systems are more flexible and malleable; therefore, experiments are easier. Some experiments would work well; others wouldn't. Because they would keep things small and flexible, it would be easy to throw away the mistakes.

This would enable the team to pivot. **Pivot** means to change direction based on recent results. It is better to pivot early in the development process than to realize well into it that you've built something that nobody likes.

Google calls this idea "launch early and often." Launch as early as possible, even if that means leaving out most of the features and launching to only a few users. What you learn from the early launches informs the decisions later on and produces a better service in the end.

Launching early and often also gives you the opportunity to build operational infrastructure early. Some companies build a service for a year and then launch it, informing the operations team only a week prior. IT then has little time to develop operational practices such as backups, oncall playbooks, and so on. Therefore, those things are done badly. With the launch-early-and-often strategy, you gain operational experience early and you have enough time to do it right.

Fail Early and Often

The strategy of "launch early and often" is sometimes called "fail early and often." Early launches are so experimental that it is highly likely they will fail. That is considered a good thing. Small failures are okay, as long as you learn from them and they lead to future success. If you never fail, you aren't taking enough risks (the good kind) and you are missing opportunities to learn.

It is wasteful to discover that your base assumptions were wrong after months of development. By proving or disproving our assumptions as soon as possible, we have more success in the end.

Launching early and often is also known as the **minimum viable product (MVP)** strategy. As defined by Eric Ries, "The minimum viable product is that version of a new product which allows a team to collect the maximum amount of validated learning about customers with the least effort" (Ries 2009). In other words, rather than focusing on new functionality in each release, focus on testing an assumption in each release.

The team building the monitoring system adopted the launch-early-and-often strategy. They decided that each iteration, or small batch, would be one week long. At the end of the week they would release what was running in their beta environment to their production environment and ask for feedback from stakeholders.

For this strategy to work they had to pick very small chunks of work. Taking a cue from Jason Punyon and Kevin Montrose's "Providence: Failure Is Always an Option" (Punyon 2015), they called this "What can get done by Friday?"–driven development.

Iteration 1 had the goal of monitoring a few servers to get feedback from various stakeholders. The team installed an open source monitoring system on a virtual machine. This was in sharp contrast to their original plan of a system that would be highly scalable. Virtual machines have less I/O and network horsepower than physical machines. Hardware could not be ordered and delivered in a one-week time frame, however, so the first iteration used virtual machines. It was what could be done by Friday.

At the end of this iteration, the team didn't have their dream monitoring system, but they had more monitoring capability than ever before.

In this iteration they learned that Simple Network Management Protocol (SNMP) was disabled on most of the organization's networking equipment. They would have to coordinate with the network team if they were to collect network utilization and other statistics. It was better to learn this now than to have their major deployment scuttled by making this discovery during the final big deployment. To work around this, the team decided to focus on monitoring things they did control, such as servers and services. This gave the network team time to create and implement a project to enable SNMP in a secure and tested way.

Iterations 2 and 3 proceeded well, adding more machines and testing other configuration options and features.

During iteration 4, however, the team noticed that the other system administrators and managers hadn't been using the system much. This was worrisome. They paused to talk one on one with people to get some honest feedback.

What the team learned was that without the ability to have dashboards that displayed historical data, the system wasn't very useful to its users. In all the past debates this issue had never been raised. Most confessed they hadn't thought it would be important until they saw the system running; others hadn't raised the issue because they simply assumed all monitoring systems had dashboards.

It was time to pivot.

The software package that had been the team's second choice had very sophisticated dashboard capabilities. More importantly, dashboards could be configured and customized by individual users. Dashboards were self-service.

After much discussion, the team decided to pivot to the other software package.

In the next iteration, they set up the new software and created an equivalent set of configurations. This went very quickly because a lot of work from the previous iterations could be reused: the decisions on what and how to monitor, the work completed with the network team, and so on.

By iteration 6, the entire team was actively using the new software. Managers were setting up dashboards to display key metrics that were important to them. People were enthusiastic about the new system.

Something interesting happened around this time: A major server crashed on Saturday morning. The monitoring system alerted the SA team, who were able to fix the problem before people arrived at the office on Monday. In the past there had been similar outages but repairs had not begun until the SAs arrived on Monday morning, well after most employees had arrived. This showed management, in a very tangible way, the value of the system.

Iteration 7 had the goal of writing a proposal to move the monitoring system to physical machines so that it would scale better. By this time the managers who would approve such a purchase were enthusiastically using the system; many had become quite expert at creating custom dashboards. The case was made to move the system to physical hardware for better scaling and performance, plus a duplicate set of hardware would be used for a hot spare site in another datacenter. The plan was approved.

In future iterations the system became more valuable to the organization as the team implemented features such as a more sophisticated oncall schedule, more monitored services, and so on. The benefits of small batches observed by the SA team included:

- **Testing assumptions early prevents wasted effort.** The ability to fail early and often means we can pivot. Problems can be fixed sooner rather than later.
- **Providing value earlier builds momentum.** People would rather have some features today than all the features tomorrow. Some monitoring is better than no monitoring. The naysayers see results and become advocates. Management has an easier time approving something that isn't hypothetical.

- **Experimentation is easier.** Often, people develop an emotional attachment to code. With small batches we can be more agile because we haven't yet grown attached to our past decisions.
- **MVP enables instant gratification.** The team saw the results of their work faster, which improved morale.
- **The team was less stressed.** There is no big, scary due date, just a constant flow of new features.
- **Big-batch debating is procrastination.** Much of the early debate had been about details and features that didn't matter or didn't get implemented.

The first few weeks were the hardest. The initial configuration required special skills. Once it was running, however, people with less technical skill or desire could add rules and make dashboards. In other words, by taking a lead and setting up the scaffolding, others can follow. This is an important point of technical leadership. Technical leadership means going first and making it easy for others to follow.

A benefit of using the MVP model is that the system is always working or in a shippable state. The system is always providing benefit, even if not all the features are delivered. Therefore, if more urgent projects take the team away, the system is still usable and running. If the original big-batch plan had continued, the appearance of a more urgent project might have left the system half-developed and unlaunched. The work done so far would have been for naught.

Another thing the team realized during this process was that not all launches were of equal value. Some launches were significant because of the time and work put into them, but included only internal changes and scaffolding. Other launches included features that were tangibly useful to the primary users of the system. Launches in this latter category were the only thing that mattered to management when they measured progress. It was most important to work toward goals that produced features that would be visibly helpful and meaningful to people outside of the team.

Thanksgiving

The U.S. Thanksgiving holiday involves a large feast. If you are not used to cooking a large meal for many people, this once-a-year event can be a stressful, scary time. Any mistakes are magnified by their visibility: All your relatives are there to see you fail. It is a big batch.

You can turn this into a small batch by trying new recipes in the weeks ahead of time, by attempting test runs of large items, or by having the event be potluck. These techniques reduce risk and stress for all involved.

2.5 Summary

Why are small batches better?

Small batches result in happier customers. Features get delivered sooner. Bugs are fixed faster.

Small batches reduce risk. By testing assumptions, the prospect of future failure is reduced. More people get experience with procedures, which means their skills are improved.

Small batches reduce waste. They avoid endless debates and perfectionism that delay the team in getting started. Less time is spent implementing features that don't get used. In the event that higher-priority projects come up, the team has already delivered a usable system.

Small batches encourage experimentation. We can try new things—even crazy ideas, some of which turn into competition-killing features. We fear failure less because we can undo a small batch easily if the experiment fails. More importantly, we learned something that will help us make future improvements.

Small batches improve the ability to innovate. Because experimentation is encouraged, we test new ideas and keep only the good ones. We can take risks. We are less attached to old pieces that must be thrown away.

Small batches improve productivity. Bugs are fixed more quickly and the process of fixing them is accelerated because the code is fresher in our minds.

Small batches encourage automation. When something must happen often, excuses not to automate go away.

Small batches make system administrators happier. We get instant gratification and Hell Month disappears. It is just simply a better way to work.

The small batches principle is an important part of the DevOps methodology and applies whether you are directly involved in a development process, making a risky process go more smoothly, deploying an externally developed package, or cooking a large meal.

Exercises

1. What is the small batches principle?
2. Why are big batches more risky than small batches?
3. Why is it better to push a new software release into production monthly rather than every six months?
4. Pick a number of software projects you are or have been involved in. How frequently were new releases issued? Compare and contrast the projects' ability to address bugs and ship new features.
5. Is it better to fail over or take down a perfectly running system than to wait until it fails on its own?

6. What is the difference between behavior and process?
7. Why is it better to have a small improvement now than a large improvement a year from now?
8. Describe the minimum viable product (MVP) strategy. What are the benefits of it versus a larger, multi-year project plan?
9. List a number of risky behaviors that cannot be improved through practice. Why are they inherently risky?
10. Which big-batch releases happen in your environment? Describe them in detail.
11. Pick a project that you are involved in. How could it be restructured so that people benefit immediately instead of waiting for the entire project to be complete?