The Small Batches Principle

**REDUCING WASTE,
ENCOURAGING
EXPERIMENTATION,
AND MAKING
EVERYONE HAPPY**

THOMAS A. LIMONCELLI

Q: What do DevOps people mean when they talk about small batches?

A: To answer that, let's take a look at an unpublished chapter from the upcoming book *The Practice of System and Network Administration*, third edition, due out in October 2016.

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 big leaps. Small batches permit us to deliver results faster, with higher quality and less stress.

We begin with an example that has nothing to do with system administration in order to demonstrate the general idea. Then we focus 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 manufacturing. It can be applied to just about any kind of process. It also enables the MVP (minimum viable product) methodology, which involves launching a small version of a service to get early feedback that informs the decisions made later in the project.

THE CARPENTER ANALOGY

Imagine a carpenter who needs 50 pieces of two-by-four lumber, all the same length. One could imagine sawing all 50

pieces then measuring them to verify they are all the correct size. It would be very disappointing to discover that the blade shifted while making piece 10, and pieces 11 through 50 are unusable. The carpenter would have to remake 40 pieces.

A better method would be to verify the length after each piece is made. If the blade had shifted, the carpenter would 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 cut all the boards, then inspected 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,...

The first 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 pieces to build a house. The pieces cannot be used until they are inspected. Using the first method, the second team cannot begin its work until all the pieces are cut and at least one piece is inspected. The chances are high that the pieces will be delivered in a big batch after they have all been inspected. In the small-batch example the new pieces are delivered without this delay.

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

IN-HOUSE SOFTWARE DEPLOYMENT

A company had a team of developers that produced a new release every six months. When that 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 the release was complex and required all hands on deck. The actual software installation never worked on the first try. Once 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. The production infrastructure changed significantly between releases, thus making it 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 the 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, the new software was usually late.

By doing new releases more frequently, there would be pressure to automate the process.

This made it impossible for the operations team to plan ahead. In particular, it was difficult to schedule any vacation time, which just created more stress.

Feeling compassion for the team's woes, someone proposed that the release 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 always an excuse to put off automating it. Also, there would be fewer changes to the infrastructure between releases. 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 battle.

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. The push 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 mostly 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. While 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 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 was 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 fixing a bug

in code while it is still fresh in your mind. Small batches meant bugs were reported soon after the code was written, which meant developers could fix them 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.

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 are worthwhile, the business benefits are even more exciting:

- ➔ **Their ability to compete improved.** 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 because of 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.

- ➔ **Enabled a culture of automation and optimization.** Rapid releases removed common excuses not to automate. New automation brought consistency, repeatability, better error checking, and less manual labor. Plus, automation could run any time, not just when the operations team was available.

FAILOVER

Stack Overflow's main website infrastructure is in a data center in New York City. If the data center fails or needs to be taken down for maintenance, duplicate equipment and software are running in Oregon.

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, 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 10 hours. The infrastructure in New York had diverged from Oregon significantly. Code that was supposed to seamlessly failover 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 10-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 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 may seem odd, but it is better to discover bugs and other problems in a controlled situation than during an emergency. Discovering a bug during an emergency at 4 a.m. 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 a.m. 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 by the next drill. The next failover took five hours, then two hours, then eventually the drills could be done in an hour with zero user-visible downtime.

The process found infrastructure changes that had not been replicated in Oregon and code that didn't failover properly. It identified new services that hadn't been engineered for smooth failover. It discovered a process that could be done only 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. The more frequent the drills, the smoother the process became.

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.

Bugs fixed faster. The smaller accumulation of infrastructure and code changes meant each drill tested fewer changes. Bugs were easier to identify and faster to fix.

Bugs 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.

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. Documentation improved while the drill was running. Automation was easier to write because the repetition helped the team see what could be automated or what pieces were most worth automating.

New opportunities revealed. 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.

Again, it became easier to schedule long vacations.

THE MONITORING PROJECT

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 on-call schedule would be used to determine whom to alert.

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 finish. 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, however, it would be easy to throw away the mistakes. This would enable the team to pivot, meaning they could 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 nobody likes.

Google calls this “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 select 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, on-call 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.

This is also known as the MVP strategy. As defined by Eric Ries in 2009, “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” (“Minimum Viable Product: a guide”; <http://www.startuplessonslearned.com/2009/08/minimum-viable-product-guide.html>). 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 to work they had to pick very small chunks of work. Taking a cue from Jason Punyon and Kevin Montrose [“Providence: Failure Is Always an Option”; <http://jasonpunyon.com/blog/2015/02/12/providence-failure-is-always-an-option/>], 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 do not have the I/O and network performance that physical hardware has. Hardware could not be ordered in a one-week time frame, however. So the first iteration used virtual machines.

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 SNMP (Simple Network Management Protocol) 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 other things, 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. They 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 previous SNMP work 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.

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.

Some people turn this into a small-batch situation by attempting new recipes in the weeks ahead of time, or by making certain key elements in a large batch as a test run. These techniques reduce risk and stress from an otherwise-busy holiday.

Something interesting happened around this time: a major server crashed on Saturday morning. The monitoring system alerted the sysadmin team, who were able to fix the problem before staff arrived at the office on Monday. In the past there had been similar outages but repairs had not begun until the sysadmins 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 hardware for better scaling and performance, and to use a duplicate set of hardware for a hot spare site in another data center.

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 on-call schedule, monitored more services, and so on. The benefits of small batches observed by the sysadmin team included:

Testing assumptions early prevents wasted effort. The ability to fail early and often means the team 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 emotional attachment to code. With small batches they can be more agile because they have grown less attached to past decisions.

Instant gratification. The team saw the results of their work faster, which improved morale.

Less stress. 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. This is called “always being in a shippable state.” The system is always working and 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 but unlaunched. The work done so far would have been for naught.

SUMMARY

Why are small batches better?

Small batches result in happier customers. Features get delivered with less latency. 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 our skills improve.

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 improve the ability to innovate. Because experimentation is encouraged, the team can test new ideas and keep 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 quicker and the process of fixing them is accelerated because the code is fresher in the mind.

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

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

Small batches make system administrators happier. We get instant gratification, and hell month disappears. It is simply a better way to work.

Reprinted with permission. Volume 1: *The Practice of System and Network Administration*, 3rd Edition

By Thomas A. Limoncelli, Christina J. Hogan, Strata R. Chalup
Due to be Published Oct. 7, 2016. Addison-Wesley

Professional. <http://the-sysadmin-book.com>

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

LOVE IT, HATE IT? LET US KNOW feedback@queue.acm.org

Thomas A. Limoncelli is an author, speaker, and system administrator. He is a site reliability engineer at Stack Overflow Inc. in New York City. His books include *The Practice of Cloud Administration* (<http://the-cloud-book.com>), *The Practice of System and Network Administration* (<http://the-sysadmin-book.com>) and *Time Management for System Administrators*. He blogs at EverythingSysadmin.com and tweets at [@YesThatTom](https://twitter.com/YesThatTom).
Copyright © 2016 held by owner/author. Publication rights licensed to ACM.