and then construct an idealized, future value stream map, which serves as a target condition to achieve by some date (e.g., usually three to twelve months).

Leadership helps define this future state and then guides and enables the team to brainstorm hypotheses and countermeasures to achieve the desired improvement to that state, perform experiments to test those hypotheses, and interpret the results to determine whether the hypotheses were correct. The teams keep repeating and iterating, using any new learnings to inform the next experiments.

CREATING A DEDICATED TRANSFORMATION TEAM

One of the inherent challenges with initiatives such as DevOps transformations is that they are inevitably in conflict with ongoing business operations. Part of this is a natural outcome of how successful businesses evolve. An organization that has been successful for any extended period of time (years, decades, or even centuries) has created mechanisms to perpetuate the practices that made them successful, such as product development, order administration, and supply chain operations.

Many techniques are used to perpetuate and protect how current processes operate, such as specialization, focus on efficiency and repeatability, bureaucracies that enforce approval processes, and controls to protect against variance. In particular, bureaucracies are incredibly resilient and are designed to survive adverse conditions—one can remove half the bureaucrats and the process will still survive.

While this is good for preserving status quo, we often need to change how we work to adapt to changing conditions in the marketplace. Doing this requires disruption and innovation, which puts us at odds with groups who are currently responsible for daily operations and the internal bureaucracies, and who will almost always win.

In their book *The Other Side of Innovation: Solving the Execution Challenge*, Dr. Vijay Govindarajan and Dr. Chris Trimble, both faculty members of Dartmouth College's Tuck School of Business, described their studies of how disruptive innovation is achieved despite these powerful forces of daily operations. They documented how customer-driven auto insurance products were successfully developed and marketed at Allstate, how the profitable digital publishing business was created at the *Wall Street Journal*, the development of the break-

through trail-running shoe at Timberland, and the development of the first electric car at BMW.

Based on their research, Dr. Govindarajan and Dr. Trimble assert that organizations need to create a dedicated transformation team that is able to operate outside of the rest of the organization that is responsible for daily operations (which they call the "dedicated team" and "performance engine" respectively).

First and foremost, we will hold this dedicated team accountable for achieving a clearly defined, measurable, system-level result (e.g., reduce the deployment lead time from "code committed into version control to successfully running in production" by 50%). In order to execute such an initiative, we do the following:

- Assign members of the dedicated team to be solely allocated to the DevOps transformation efforts (as opposed to "maintain all your current responsibilities but spend 20% of your time on this new DevOps thing").

- Select team members who are generalists, who have skills across a wide variety of domains.

- Select team members who have longstanding and mutually respectful relationships with the rest of the organization.

- Create a separate physical space for the dedicated team, if possible, to maximize communication flow within the team and create some isolation from the rest of the organization.

If possible, we will free the transformation team from many of the rules and policies that restrict the rest of the organization, as National Instruments did, described in the previous chapter. After all, established processes are a form of institutional memory—we need the dedicated team to create the new processes and learnings required to generate our desired outcomes, creating new institutional memory.

Creating a dedicated team is not only good for the team but also good for the performance engine. By creating a separate team, we create the space for them to experiment with new practices, protecting the rest of the organization from the potential disruptions and distractions associated with it.

AGREE ON A SHARED GOAL

One of the most important parts of any improvement initiative is to define a measurable goal with a clearly defined deadline, between six months and two years in the future. It should require considerable effort but still be achievable. And achievement of the goal should create obvious value for the organization as a whole and to our customers.

These goals and the time frame should be agreed upon by the executives and known to everyone in the organization. We also want to limit the number of these types of initiatives going on simultaneously to prevent us from overly taxing the organizational change management capacity of leaders and the organization. Examples of improvement goals might include:

- Reduce the percentage of the budget spent on product support and unplanned work by 50%.
- Ensure lead time from code check-in to production release is one week or less for 95% of changes.
- Ensure releases can always be performed during normal business hours with zero downtime.
- Integrate all the required information security controls into the deployment pipeline to pass all required compliance requirements.

Once the high-level goal is made clear, teams should decide on a regular cadence to drive the improvement work. Like product development work, we want transformation work to be done in an iterative, incremental manner. A typical iteration will be in the range of two to four weeks. For each iteration, the teams should agree on a small set of goals that generate value and makes some progress toward the long-term goal. At the end of each iteration, teams should review their progress and set new goals for the next iteration.

KEEP OUR IMPROVEMENT PLANNING HORIZONS SHORT

In any DevOps transformation project, we need to keep our planning horizons short, just as if we were in a startup doing product or customer development. Our initiative should strive to generate measurable improvements or actionable data within weeks (or, in the worst case, months).

By keeping our planning horizons and iteration intervals short, we achieve the following:

- Flexibility and the ability to reprioritize and replan quickly
- Decrease the delay between work expended and improvement realized, which strengthens our feedback loop, making it more likely to reinforce desired behaviors—when improvement initiatives are successful, it encourages more investment
- Faster learning generated from the first iteration, meaning faster integration of our learnings into the next iteration
- Reduction in activation energy to get improvements
- Quicker realization of improvements that make meaningful differences in our daily work
- Less risk that our project is killed before we can generate any demonstrable outcomes

RESERVE 20% OF CYCLES FOR NON-FUNCTIONAL REQUIREMENTS AND REDUCING TECHNICAL DEBT

A problem common to any process improvement effort is how to properly prioritize it—after all, organizations that need it most are those that have the least amount of time to spend on improvement. This is especially true in technology organizations because of technical debt.

Organizations that struggle with financial debt only make interest payments and never reduce the loan principal, and may eventually find themselves in situations where they can no longer service the interest payments. Similarly, organizations that don't pay down technical debt can find themselves so burdened with daily workarounds for problems left unfixed that they can no longer complete any new work. In other words, they are now only making the interest payment on their technical debt.

We will actively manage this technical debt by ensuring that we invest at least 20% of all Development and Operations cycles on refactoring, investing in automation work and architecture and non-functional requirements (NFRs,

sometimes referred to as the "ilities"), such as maintainability, manageability, scalability, reliability, testability, deployability, and security.

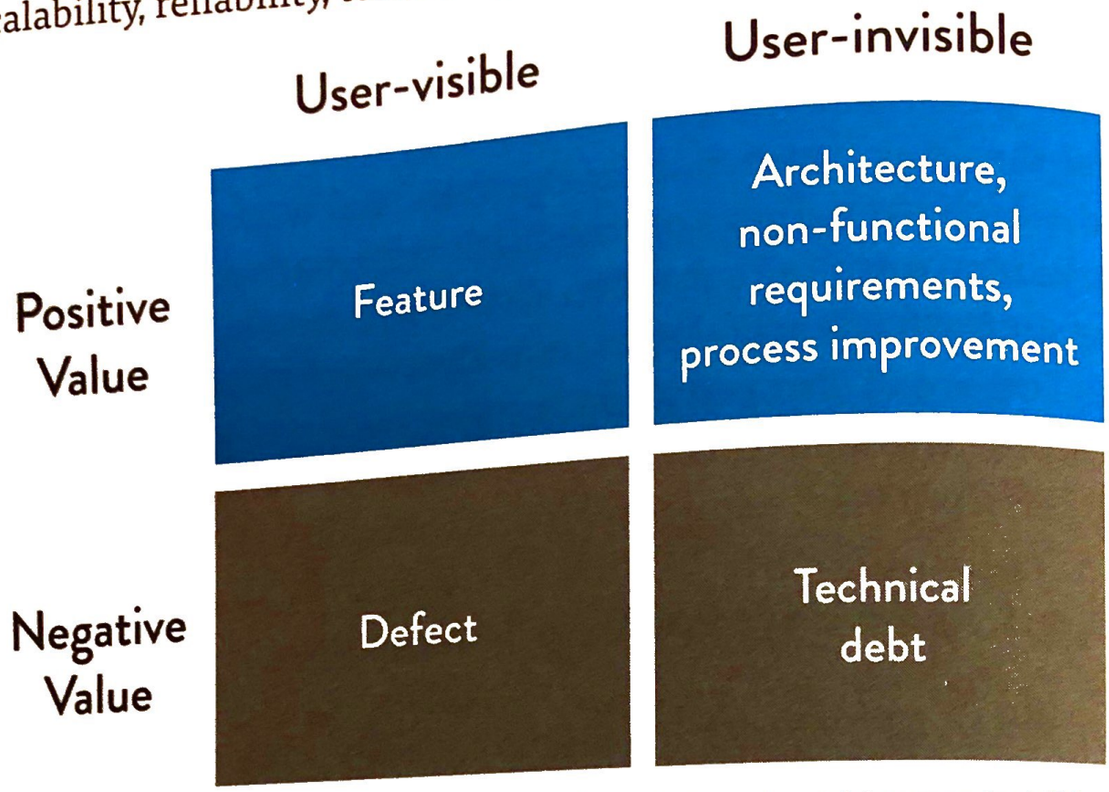

Figure 11: *Invest 20% of cycles on those that create positive, user-invisible value (Source: "Machine Learning and Technical Debt with D. Sculley," Software Engineering Daily podcast, November 17, 2015, http://softwareengineeringdaily.com/2015/11/17/machine-learning-and-technical-debt-with-d-sculley/.)*

After the near-death experience of eBay in the late 1990s, Marty Cagan, author of *Inspired: How To Create Products Customers Love*, the seminal book on product design and management, codified the following lesson:

> The deal [between product owners and] engineering goes like this: Product management takes 20% of the team's capacity right off the top and gives this to engineering to spend as they see fit. They might use it to rewrite, re-architect, or re-factor problematic parts of the code base...whatever they believe is necessary to avoid ever having to come to the team and say, 'we need to stop and rewrite [all our code].' If you're in really bad shape today, you might need to make this 30% or even more of the resources. However, I get nervous when I find teams that think they can get away with much less than 20%.

Cagan notes that when organizations do not pay their "20% tax," technical debt will increase to the point where an organization inevitably spends all of its cycles paying down technical debt. At some point, the services become so fragile that feature delivery grinds to a halt because all the engineers are working on reliability issues or working around problems.

By dedicating 20% of our cycles so that Dev and Ops can create lasting countermeasures to the problems we encounter in our daily work, we ensure that

technical debt doesn't impede our ability to quickly and safely develop and operate our services in production. Alleviating added pressure of technical debt from workers can also reduce levels of burnout.

Case Study
Operation InVersion at LinkedIn (2011)

LinkedIn's Operation InVersion presents an interesting case study that illustrates the need to pay down technical debt as a part of daily work. Six months after their successful IPO in 2011, LinkedIn continued to struggle with problematic deployments that became so painful that they launched Operation InVersion, where they stopped all feature development for two months in order to overhaul their computing environments, deployments, and architecture.

LinkedIn was created in 2003 to help users "connect to your network for better job opportunities." By the end of their first week of operation, they had 2,700 members. One year later, they had over one million members, and have grown exponentially since then. By November 2015, LinkedIn had over 350 million members, who generate tens of thousands of requests per second, resulting in millions of queries per second on the LinkedIn backend systems.

From the beginning, LinkedIn primarily ran on their homegrown Leo application, a monolithic Java application that served every page through servlets and managed JDBC connections to various backend Oracle databases. However, to keep up with growing traffic in their early years, two critical services were decoupled from Leo: the first handled queries around the member connection graph entirely in-memory, and the second was member search, which layered over the first.

By 2010, most new development was occurring in new services, with nearly one hundred services running outside of Leo. The problem was that Leo was only being deployed once every two weeks.

Josh Clemm, a senior engineering manager at LinkedIn, explained that by 2010, the company was having significant problems with Leo. Despite vertically scaling Leo by adding memory and CPUs, "Leo was often going down in production, it was difficult to troubleshoot and recover, and difficult to release new code....It was clear we needed to 'Kill Leo' and break it up into many small functional and stateless services."

In 2013, journalist Ashlee Vance of Bloomberg described how "when LinkedIn would try to add a bunch of new things at once, the site would crumble into a broken mess, requiring engineers to work long into the night and fix the problems." By Fall 2011, late nights were no longer a rite of passage or a bonding activity, because the problems had become intolerable. Some of LinkedIn's top engineers, including Kevin Scott who had joined as the LinkedIn VP of Engineering three months before their initial public offering, decided to completely stop engineering work on new features and dedicate the whole department to fixing the site's core infrastructure. They called the effort Operation InVersion.

Scott launched Operation InVersion as a way to "inject the beginnings of a cultural manifesto into his team's engineering culture. There would be no new feature development until LinkedIn's computing architecture was revamped—it's what the business *and* his team needed."

Scott described one downside, "You go public, have all the world looking at you, and then we tell management that we're not going to deliver anything new while all of engineering works on this [InVersion] project for the next two months. It was a scary thing."

However, Vance described the massively positive results of Operation InVersion. "LinkedIn created a whole suite of software and tools to help it develop code for the site. Instead of waiting weeks for their new features to make their way onto LinkedIn's main site, engineers could develop a new service, have a series of automated systems examine the code for any bugs and issues the service might have interacting with existing features, and launch it right to the live LinkedIn site...LinkedIn's engineering corps [now] performs major upgrades to the site three times a day." By creating a safer system of work, the value they created included fewer late night cram sessions, with more time to develop new, innovative features.

As Josh Clemm described in his article on scaling at LinkedIn, "Scaling can be measured across many dimensions, including organizational....[Operation InVersion] allowed the entire engineering organization to focus on improving tooling and deployment, infrastructure, and developer productivity. It was successful in enabling the engineering agility we need to build the scalable new products we have today....[In] 2010, we already had over 150 separate services. Today, we have over 750 services."

Kevin Scott stated, "Your job as an engineer and your purpose as a technology team is to help your company win. If you lead a team of engineers, it's better

to take a CEO's perspective. Your job is to figure out what it is that your company, your business, your marketplace, your competitive environment needs. Apply that to your engineering team in order for your company to win."

By allowing LinkedIn to pay down nearly a decade of technical debt, Project InVersion enabled stability and safety, while setting the next stage of growth for the company. However, it required two months of total focus on non-functional requirements, at the expense of all the promised features made to the public markets during an IPO. By finding and fixing problems as part of our daily work, we manage our technical debt so that we avoid these "near death" experiences.

INCREASE THE VISIBILITY OF WORK

In order to be able to know if we are making progress toward our goal, it's essential that everyone in the organization knows the current state of work. There are many ways to make the current state visible, but what's most important is that the information we display is up to date, and that we constantly revise what we measure to make sure it's helping us understand progress toward our current target conditions.

The following section discusses patterns that can help create visibility and alignment across teams and functions.

USE TOOLS TO REINFORCE DESIRED BEHAVIOR

As Christopher Little, a software executive and one of the earliest chroniclers of DevOps, observed, "Anthropologists describe tools as a cultural artifact. Any discussion of culture after the invention of fire must also be about tools." Similarly, in the DevOps value stream, we use tools to reinforce our culture and accelerate desired behavior changes.

One goal is that our tooling reinforces that Development and Operations not only have shared goals but have a common backlog of work, ideally stored in a common work system and using a shared vocabulary, so that work can be prioritized globally.

By doing this, Development and Operations may end up creating a shared work queue, instead of each silo using a different one (e.g., Development uses JIRA while Operations uses ServiceNow). A significant benefit of this is that when production incidents are shown in the same work systems as develop-

ment work, it will be obvious when ongoing incidents should halt other work, especially when we have a kanban board.

Another benefit of having Development and Operations using a shared tool is a unified backlog, where everyone prioritizes improvement projects from a global perspective, selecting work that has the highest value to the organization or most reduces technical debt. As we identify technical debt, we add it to our prioritized backlog if we can't address it immediately. For issues that remain unaddressed, we can use our "20% time for non-functional requirements" to fix the top items from our backlog.

Other technologies that reinforce shared goals are chat rooms, such as IRC channels, HipChat, Campfire, Slack, Flowdock, and OpenFire. Chat rooms allow the fast sharing of information (as opposed to filling out forms that are processed through predefined workflows), the ability to invite other people as needed, and history logs that are automatically recorded for posterity and can be analyzed during post-mortem sessions.

An amazing dynamic is created when we have a mechanism that allows any team member to quickly help other team members, or even people outside their team—the time required to get information or needed work can go from days to minutes. In addition, because everything is being recorded, we may not need to ask someone else for help in the future—we simply search for it.

However, the rapid communication environment facilitated by chat rooms can also be a drawback. As Ryan Martens, the founder and CTO of Rally Software, observes, "In a chat room, if someone doesn't get an answer in a couple of minutes, it's totally accepted and expected that you can bug them again until they get what they need."

The expectations of immediate response can, of course, lead to undesired outcomes. A constant barrage of interruptions and questions can prevent people from getting necessary work done. As a result, teams may decide that certain types of requests should go through more structured and asynchronous tools.

CONCLUSION

In this chapter, we identified all the teams supporting our value stream and captured in a value stream map what work is required in order to deliver value to the customer. The value stream map provides the basis for understanding

our current state, including our lead time and %C/A metrics for problematic areas, and informs how we set a future state.

This enables dedicated transformation teams to rapidly iterate and experiment to improve performance. We also make sure that we allocate a sufficient amount of time for improvement, fixing known problems and architectural issues, including our non-functional requirements. The case studies from Nordstrom and LinkedIn demonstrate how dramatic improvements can be made in lead times and quality when we find problems in our value stream and pay down technical debt.