CHAPTER 2

MEASURING PERFORMANCE

There are many frameworks and methodologies that aim to improve the way we build software products and services. We wanted to discover what works and what doesn't in a scientific way, starting with a definition of what "good" means in this context. This chapter presents the framework and methods we used to work towards this goal, and in particular the key outcome measures applied throughout the rest of this book.

By the end of this chapter, we hope you'll know enough about our approach to feel confident in the results we present in the rest of the book.

Measuring performance in the domain of software is hard—in part because, unlike manufacturing, the inventory is invisible. Furthermore, the way we break down work is relatively arbitrary, and the design and delivery activities—particularly in the Agile software development paradigm—happen simultaneously. Indeed, it's expected that we will change and evolve our design based on what we learn by trying to implement it. So our first step must be to define a valid, reliable measure of software delivery performance.

THE FLAWS IN PREVIOUS ATTEMPTS TO MEASURE PERFORMANCE

There have been many attempts to measure the performance of software teams. Most of these measurements focus on productivity. In general, they suffer from two drawbacks. First, they focus on *outputs* rather than *outcomes*. Second, they focus on individual or local measures rather than team or global ones. Let's take three examples: lines of code, velocity, and utilization.

Measuring productivity in terms of lines of code has a long history in software. Some companies even required developers to record the lines of code committed per week.[1] However, in reality we would prefer a 10-line solution to a 1,000-line solution to a problem. Rewarding developers for writing lines of code leads to bloated software that incurs higher maintenance costs and higher cost of change. Ideally, we should reward developers for solving business problems with the minimum amount of code—and it's even better if we can solve a problem without writing code at all or by deleting code (perhaps by a business process change). However, minimizing lines of code isn't an ideal measure either. At the extreme, this too has its drawbacks: accomplishing a task in a single line of code that no one else can understand is less desirable than writing a few lines of code that are easily understood and maintained.

With the advent of Agile software development came a new way to measure productivity: velocity. In many schools of Agile, problems are broken down into stories. Stories are then estimated by developers and assigned a number of "points" representing the relative effort expected to complete them. At the end of an iteration, the total number of points signed off by the customer is recorded—this is the team's velocity. Velocity is designed to be used as a *capacity planning tool;* for example, it can be used to extrapolate how long it will take the team to complete all the work that has been planned and estimated. However, some managers have also used it as a way to measure team productivity, or even to compare teams.

Using velocity as a productivity metric has several flaws. First, velocity is a relative and team-dependent measure, not an absolute one. Teams usually have significantly different contexts which render their velocities incommensurable. Second, when velocity is used as a productivity measure, teams inevitably work to game their velocity. They inflate their estimates and focus on completing as many stories as possible at the expense of collaboration with other teams (which might decrease their velocity and increase the other team's velocity, making them look bad). Not only does this destroy the utility of velocity for its intended purpose, it also inhibits collaboration between teams.

Finally, many organizations measure utilization as a proxy for productivity. The problem with this method is that high utilization is only good up to a point. Once utilization gets above a certain level, there is no spare capacity (or "slack") to absorb unplanned work, changes to the plan, or improvement work. This results in longer lead times to complete work. Queue theory in math tells us that as utilization approaches 100%, lead times approach infinity—in other words, once you get to very high levels of utilization, it takes teams exponentially longer to get anything done. Since lead time—a measure of how fast work can be completed—is a productivity metric that doesn't suffer from the drawbacks of the other metrics we've seen, it's essential that we manage utilization to balance it against lead time in an economically optimal way.

MEASURING SOFTWARE DELIVERY PERFORMANCE

A successful measure of performance should have two key characteristics. First, it should focus on a global outcome to ensure teams aren't pitted against each other. The classic example is rewarding developers for throughput and operations for stability: this is a key contributor to the "wall of confusion" in which development throws poor quality code over the wall to operations, and operations puts in place painful change management processes as a way to inhibit change. Second, our measure should focus on outcomes not output: it shouldn't reward people for putting in large amounts of busywork that doesn't actually help achieve organizational goals.

In our search for measures of delivery performance that meet these criteria, we settled on four: delivery lead time, deployment frequency, time to restore service, and change fail rate. In this section, we'll discuss why we picked these particular measures.

The elevation of lead time as a metric is a key element of Lean theory. Lead time is the time it takes to go from a customer making a request to the request being satisfied. However, in the context of product development, where we aim to satisfy multiple customers in ways they may not anticipate, there are two parts to lead time: the time it takes to design and validate a product or feature, and the time to deliver the feature to customers. In the design part of the lead time, it's often unclear when to start the clock, and often there is high variability. For this reason, Reinertsen calls this part of the lead time the "fuzzy front end" (Reinertsen 2009). However, the *delivery* part of the lead time—the time it takes for work to be implemented, tested, and delivered—is easier to measure and has a lower variability. Table 2.1 (Kim et al. 2016) shows the distinction between these two domains.

Table 2.1 Design vs. Delivery

Product Design and Development	Product Delivery (Build, Testing, Deployment)
Create new products and services that solve customer problems using hypothesis-driven delivery, modern UX, design thinking.	Enable fast flow from development to production and reliable releases by standardizing work, and reducing variability and batch sizes.
Feature design and implementation may require work that has never been performed before.	Integration, test, and deployment must be performed continuously as quickly as possible.
Estimates are highly uncertain.	Cycle times should be well-known and predictable.
Outcomes are highly variable.	Outcomes should have low variability.

Shorter product delivery lead times are better since they enable faster feedback on what we are building and allow us to course correct more rapidly. Short lead times are also important when there is a defect or outage and we need to deliver a fix rapidly and with high confidence. We measured product delivery lead time as the time it takes to go from code committed to code successfully running in production, and asked survey respondents to choose from one of the following options:

- less than one hour
- less than one day
- between one day and one week
- between one week and one month
- between one month and six months
- more than six months

The second metric to consider is batch size. Reducing batch size is another central element of the Lean paradigm—indeed, it was one of the keys to the success of the Toyota production system. Reducing batch sizes reduces cycle times and variability in flow, accelerates feedback, reduces risk and overhead, improves efficiency, increases motivation and urgency, and reduces costs and schedule growth (Reinertsen 2009, Chapter 5). However, in software, batch size is hard to measure and communicate across contexts as there is no visible inventory. Therefore, we settled on deployment frequency as a proxy for batch size since it is easy to measure and typically has low variability.[2] By "deployment" we mean a software deployment to production or to an app store. A release (the changes that get deployed) will typically consist of multiple version control commits, unless the organization has achieved a single-piece flow where each commit can be released to production (a practice known as continuous deployment). We asked survey respondents how often their organization deploys code for the primary service or application they work on, offering the following options:

- on demand (multiple deploys per day)
- between once per hour and once per day
- between once per day and once per week
- between once per week and once per month
- between once per month and once every six months
- fewer than once every six months

Delivery lead times and deployment frequency are both measures of software delivery performance *tempo*. However, we wanted to investigate whether teams who improved their performance were doing so at the expense of the stability of the systems they were working on. Traditionally, reliability is measured as time between failures. However, in modem software products and services, which are rapidly changing complex systems, failure is inevitable, so the key question becomes: How quickly can service be restored? We asked respondents how long it generally takes to restore service for the primary application or service they work on when a service incident (e.g., unplanned outage, service impairment) occurs, offering the same options as for lead time (above).

Finally, a key metric when making changes to systems is what percentage of changes to production (including, for example, software releases and infrastructure configuration changes) fail. In the context of Lean, this is the same as percent complete and accurate for the product delivery process, and is a key quality metric. We asked respondents what percentage of changes for the primary application or service they work on either result in degraded service or subsequently require remediation (e.g., lead to service impairment or outage, require a hotfix, a rollback, a fix-forward, or a patch). The four measures selected are shown in Figure 2.1.

Software Delivery Performance
Lead Time
Deployment Frequency
Mean Time to Restore (MTTR)
Change Fail Percentage

Figure 2.1: Software Delivery Performance

In order to analyze delivery performance across the cohort we surveyed, we used a technique called *cluster analysis*. Cluster analysis is a foundational technique in statistical data analysis that attempts to group responses so that responses in the same group are more similar to each other than to responses in other groups. Each measurement is put on a separate dimension, and the clustering algorithm attempts to minimize the distance between all cluster members and maximize differences between clusters. This technique has no understanding of the semantics of responses—in other words, it doesn't know what counts as a "good" or "bad" response for any of the measures.[3]

This data-driven approach that categorizes the data without any bias toward "good" or "bad" gives us an opportunity to view trends in the industry without biasing the results a priori. Using cluster analysis also allowed us to identify categories of software delivery performance seen in the industry: Are there high performers and low performers, and what characteristics do they have?

We applied cluster analysis in all four years of the research project and found that every year, there were significantly different categories of software delivery performance in the industry. We also found that all four measures of software delivery performance are good classifiers and that the groups we identified in the analysis—high, medium, and low performers—were all significantly different across all four measures.

Tables 2.2 and 2.3 show you the details for software delivery performance for the last two years of our research (2016 and 2017).

Table 2.2 Software. Delivery Performance for 2016

2016	High Performers	Medium Performers	Low Performers
Deployment Frequency	On demand (multiple deploys per day)	Between once per week and once per month	Between once per month and once every six months
Lead Time for Changes	Less than one hour	Between one week and one month	Between one month and six months
MTTR	Less than one hour	Less than one day	Less than one day*
Change Failure Rate	0-15%	31-45%	16-30%

Table 2.3 Software Delivery Performance for 2017

2017	High Performers	Medium Performers	Low Performers
Deployment Frequency	On demand (multiple deploys per day)	Between once per week and once per month	Between once per week and once per month*
Lead Time for Changes	Less than one hour	Between one week and one month	Between one week and one month*
MTTR	Less than one hour	Less than one day	Between one day and one week
Change Failure Rate	0-15%	0-15%	31-45%

* Low performers were lower on average (at a statistically significant level) but had the same median as the medium performers.

Astonishingly, these results demonstrate that there is no tradeoff between improving performance and achieving higher levels of stability and quality. Rather, high performers do better at *all* of these measures. This is precisely what the Agile and Lean movements predict, but much dogma in our industry still rests on the false assumption that moving faster means *trading off* against other performance goals, rather than enabling and reinforcing them.[4]

Furthermore, over the last few years we've found that the high-performing cluster is pulling away from the pack. The DevOps mantra of continuous improvement is both exciting and real, pushing companies to be their best, and leaving behind those who do not improve. Clearly, what was state of the art three years ago is just not good enough for today's business environment.

Compared to 2016, high performers in 2017 maintained or improved their performance, consistently maximizing both tempo and stability. Low performers, on the other hand, maintained the same level of throughput from 2014-2016 and only started to increase in 2017—likely realizing that the rest of the industry was pulling away from them. In 2017, we saw low performers lose some ground in stability. We suspect this is due to attempts to increase tempo ("work harder!") which fail to address the underlying obstacles to improved overall performance (for example, rearchitecture, process improvement, and automation). We show the trends in Figures 2.2 and 2.3.

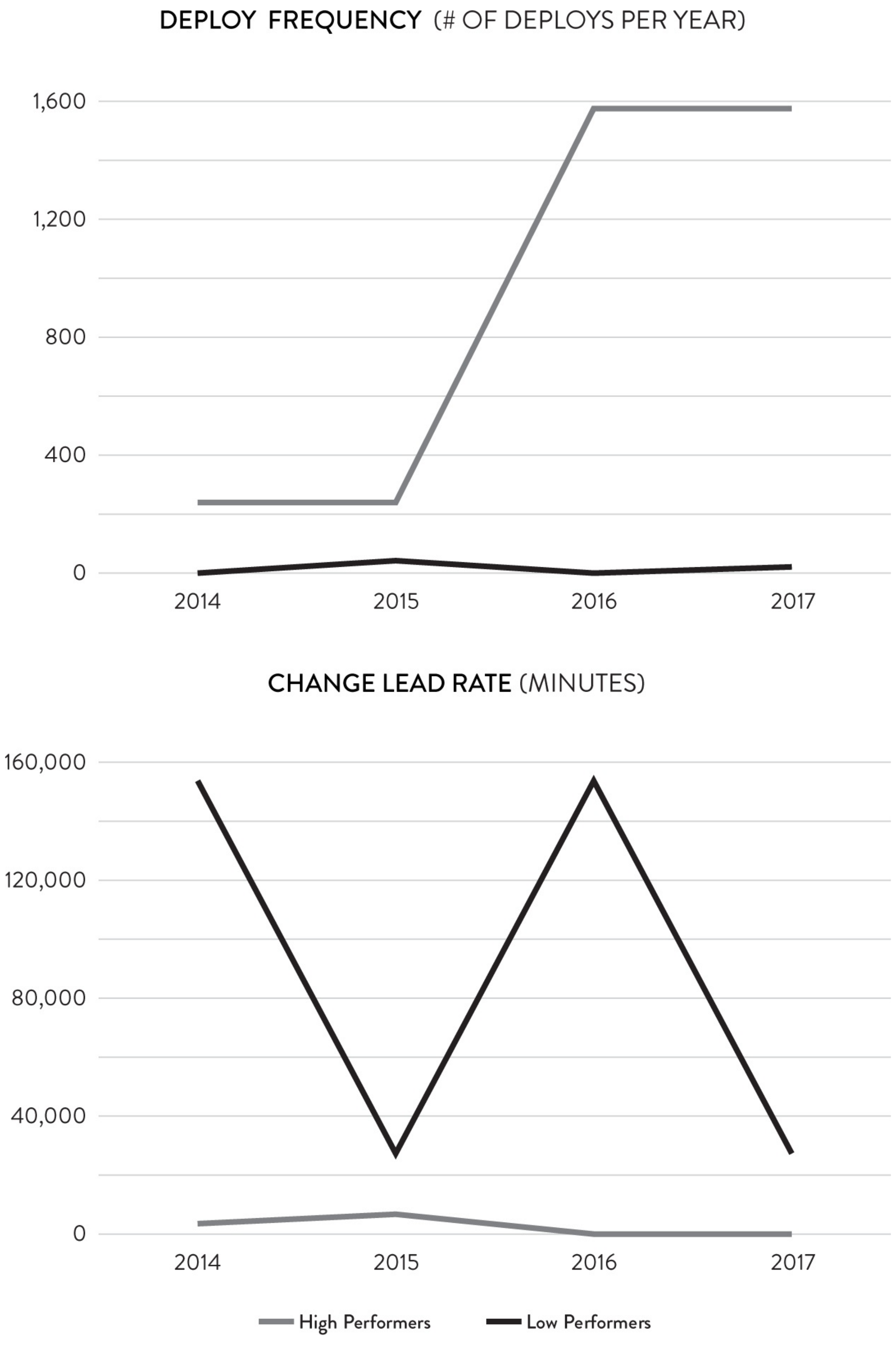

Figure 2.2: Year over Year Trends: Tempo

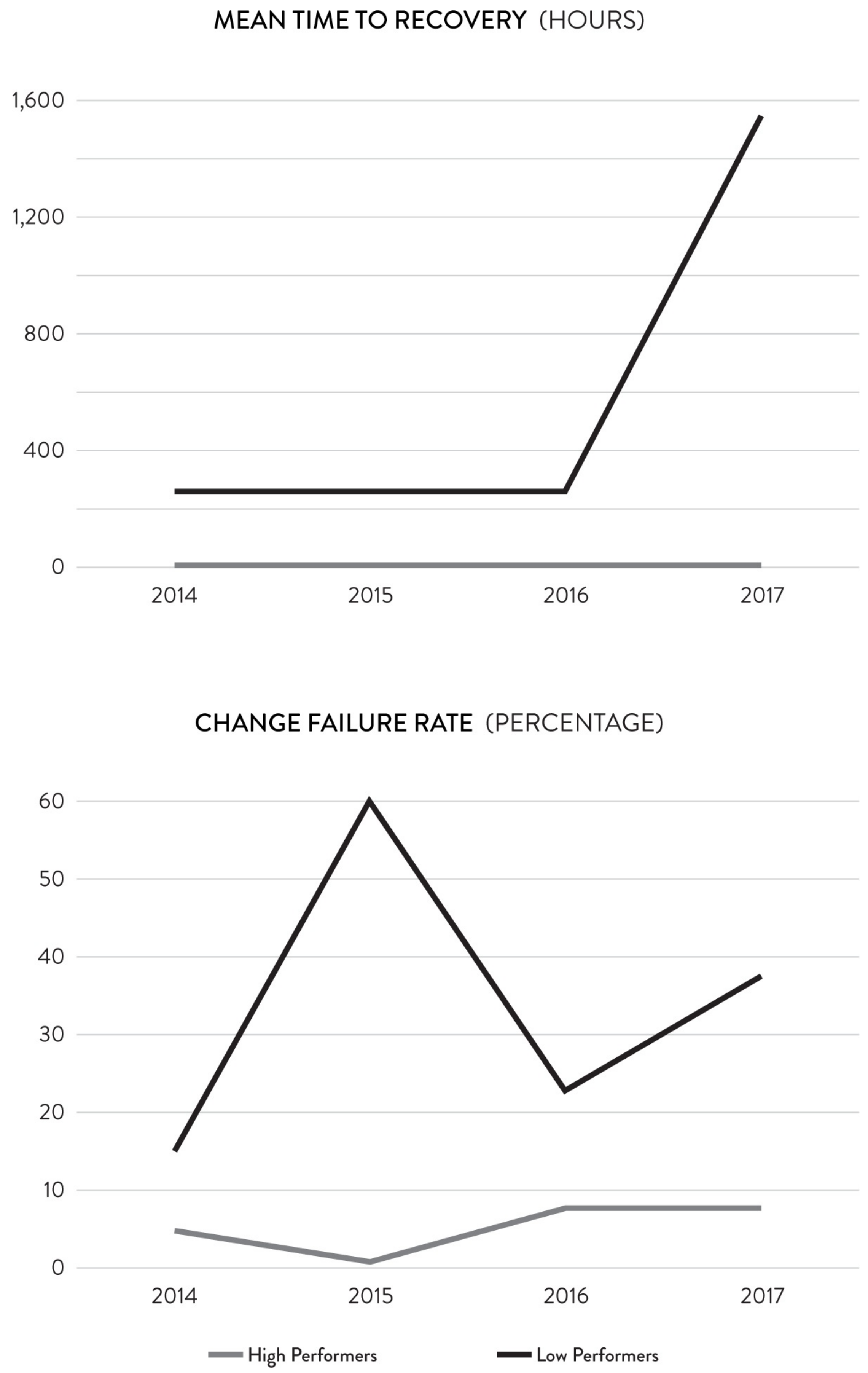

Figure 2.3: Year over Year Trends: Stability

Surprise!

Observant readers will notice that medium performers do worse than low performers on change fail rate in 2016. 2016 is the first year of our research where we see slightly inconsistent performance across our measures in any of our performance groups, and we see it in medium and low performers. Our research doesn't conclusively explain this, but we have a few ideas about why this might be the case.

One possible explanation is that medium performers are working along their technology transformation journey and dealing with the challenges that come from large-scale rearchitecture work, such as transitioning legacy code bases. This would also match another piece of the data from the 2016 study, where we found that medium performers spend more time on unplanned rework than low performers— because they report spending a greater proportion of time on new work.

We believe this new work could be occurring at the expense of ignoring critical rework, thus racking up technical debt which in turn leads to more fragile systems and, therefore, a higher change fail rate.

We have found a valid, reliable way to measure software delivery performance that satisfies the requirements we laid out. It focuses on global, system-level goals, and measures outcomes that different functions must collaborate in order to improve. The next question we wanted to answer is: Does software delivery performance matter?

THE IMPACT OF DELIVERY PERFORMANCE ON ORGANIZATIONAL PERFORMANCE

In order to measure organizational performance, survey respondents were asked to rate their organization's relative performance across several dimensions: profitability, market share, and productivity. This is a scale that has been validated multiple times in prior research (Widener 2007). This measure of organizational performance has also been found to be highly correlated to measures of return on investment (ROI), and it is robust to economic cycles—a great measure for our purposes. Analysis over several years shows that high-performing organizations were consistently *twice as likely* to exceed these goals as low performers. This demonstrates that your organization's software delivery capability can in fact provide a competitive advantage to your business.

In 2017, our research also explored how IT performance affects an organization's ability to achieve broader organizational goals—that is, goals that go beyond simple profit and revenue measures. Whether you're trying to generate profits or not, any organization today depends on technology to achieve its mission and provide value to its customers or stakeholders quickly, reliably, and securely. Whatever the mission, how a technology organization performs can predict overall organizational performance. To measure noncommercial goals, we used a scale that has been validated multiple times and is particularly well-suited for this purpose (Cavalluzzo and Ittner 2004). We found that high performers were also twice as likely to exceed objectives in quantity of goods and services, operating efficiency, customer satisfaction, quality of products or services, and achieving organization or mission goals. We show this relationship in Figure 2.4.

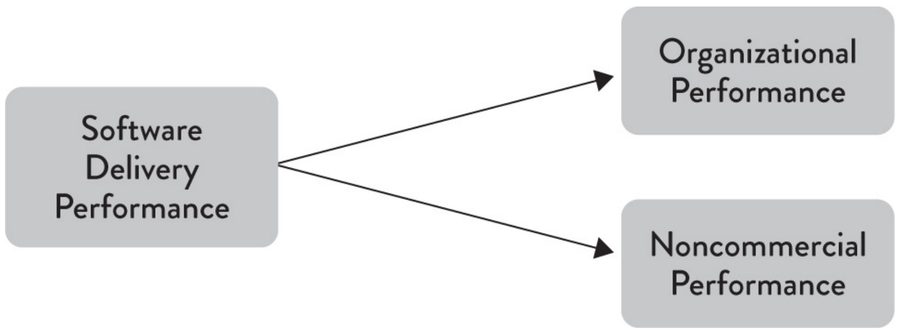
Software Delivery Performance
Organizational Performance
Noncommercial Performance

Figure 2.4: Impacts of Software Delivery Performance

Reading the Figures in This Book

We will include figures to help guide you through the research.

- When you see a box, this is a construct we have measured. (For details on constructs, see Chapter 13.)
- When you see an arrow linking boxes, this signifies a predictive relationship. You read that right: the research in this book includes analyses that go beyond correlation into prediction. (For details, see Chapter 12 on inferential prediction.) You can read these arrows using the words “drives,” “predicts,” “affects,” or “impacts.” These are all positive relationships unless otherwise noted.

For example, Figure 2.4 could be read as “software delivery performance impacts organizational performance and noncommercial performance.”

In software organizations, the ability to work and deliver in small batches is especially important, because it allows you to gather user feedback quickly using techniques such as A/B testing. It's worth noting that the ability to take an experimental approach to product development is highly correlated with the technical practices that contribute to continuous delivery.

The fact that software delivery performance matters provides a strong argument against outsourcing the development of software that is strategic to your business, and instead bringing this capability into the core of your organization. Even the US Federal Government, through initiatives such as the US Digital Service and its agency affiliates and the General Services Administration's Technology Transformation Service team, has invested in bringing software development capability in-house for strategic initiatives.

In contrast, most software used by businesses (such as office productivity software and payroll systems) are not strategic and should in many cases be acquired using the software-as-a-service model. Distinguishing which software is strategic and which isn't, and managing them appropriately, is of enormous importance. This topic is dealt with at length by Simon Wardley, creator of the Wardley mapping method (Wardley 2015).

DRIVING CHANGE

Now that we have defined software delivery performance in a way that is rigorous and measurable, we can make evidence-based decisions on how to improve the performance of teams building software-based products and services. We can compare and benchmark teams against the larger organizations they work in and against the wider industry. We can measure their improvement—or backsliding—over time. And perhaps most exciting of all, we can go beyond correlation and start testing prediction. We can test hypotheses about which practices—from managing work in process to test automation—actually impact delivery performance and the strength of these effects. We can measure other outcomes we care about, such as team burnout and deployment pain. We can answer questions like, "Do change management boards actually improve delivery performance?" (Spoiler alert: they do not; they are negatively correlated with tempo and stability.)

As we show in the next chapter, it is also possible to model and measure culture quantitatively. This enables us to measure the effect of DevOps and continuous delivery practices on culture and, in turn, the effect of culture on software delivery performance and organizational performance. Our ability to measure and reason about practices, culture, and outcomes is an incredibly powerful tool that can be used to great positive effect in the pursuit of ever higher performance.

You can, of course, use these tools to model your own performance. Use Table 2.3 to discover where in our taxonomy you fall. Use our measures for lead time, deployment frequency, time to restore service, and change fail rate, and ask your teams to set targets for these measures.

However, it is essential to use these tools carefully. In organizations with a learning culture, they are incredibly powerful. But “in pathological and bureaucratic organizational cultures, measurement is used as a form of control, and people hide information that challenges existing rules, strategies, and power structures. As Deming said, ’whenever there is fear, you get the wrong numbers’” (Humble et al. 2014, p. 56). Before you are ready to deploy a scientific approach to improving performance, you must first understand and develop your culture. It is to this topic we now turn.

---

[1] There’s a good story about how the Apple Lisa team’s management discovered that lines of code were meaningless as a productivity metric: http://www.folklore.org/StoryView.py?story=Negative_2000_Lines_Of_Code.txt.

[2] Strictly, deployment frequency is the reciprocal of batch size-the more frequently we deploy, the smaller the size of the batch. For more on measuring batch size in the context of IT service management, see Forsgren and Humble (2016).

[3] For more on cluster analysis, see Appendix B.

[4] See https://continuousdelivery.com/2016/04/the-flaw-at-the-heart-of-bimodal-it/ for an analysis of problems with the bimodal approach to ITSM, which rests on this false assumption.