TECHNICAL PRACTICES

At the time the Agile Manifesto was published in 2001, Extreme Programming (XP) was one of the most popular Agile frameworks.[1] In contrast to Scrum, XP prescribes a number of technical practices such as test-driven development and continuous integration. *Continuous Delivery* (Humble and Farley 2010) also emphasizes the importance of these technical practices (combined with comprehensive configuration management) as an enabler of more frequent, higher-quality, and lower-risk software releases.

Many Agile adoptions have treated technical practices as secondary compared to the management and team practices that some Agile frameworks emphasize. Our research shows that technical practices play a vital role in achieving these outcomes.

In this chapter, we discuss the research we performed to measure continuous delivery as a capability and to assess its impact on software delivery performance, organizational culture, and other outcome measures, such as team burnout and deployment pain. We find that continuous delivery practices do in fact have a measurable impact on these outcomes.

WHAT IS CONTINUOUS DELIVERY?

Continuous delivery is a set of capabilities that enable us to get changes of all kinds—features, configuration changes, bug fixes, experiments—into production or into the hands of users *safely, quickly,* and *sustainably*. There are five key principles at the heart of continuous delivery:

- **Build quality in.** The third of W. Edwards Deming's fourteen points for management states, "Cease dependence on inspection to achieve quality. Eliminate the need for inspection on a mass basis by building quality into the product in the first place" (Deming 2000). In continuous delivery, we invest in building a culture supported by tools and people where we can detect any issues quickly, so that they can be fixed straight away when they are cheap to detect and resolve.
- **Work in small batches.** Organizations tend to plan work in big chunks—whether building new products or services or investing in organizational change. By splitting work up into much smaller chunks that deliver measurable business outcomes quickly for a small part of our target market, we get essential feedback on the work we are doing so that we can course correct. Even though working in small chunks adds some overhead, it reaps enormous rewards by allowing us to avoid work that delivers zero or negative value for our organizations.

A key goal of continuous delivery is changing the

economics of the software delivery process so the cost of pushing out individual changes is very low.

- **Computers perform repetitive tasks; people solve problems.** One important strategy to reduce the cost of pushing out changes is to take repetitive work that takes a long time, such as regression testing and software deployments, and invest in simplifying and automating this work. Thus, we free up people for higher-value problem-solving work, such as improving the design of our systems and processes in response to feedback.
- **Relentlessly pursue continuous improvement.** The most important characteristic of high-performing teams is that they are never satisfied: they always strive to get better. High performers make improvement part of everybody's daily work.
- **Everyone is responsible.** As we learned from Ron Westrum, in bureaucratic organizations teams tend to focus on departmental goals rather than organizational goals. Thus, development focuses on throughput, testing on quality, and operations on stability. However, in reality these are all system-level outcomes, and they can only be achieved by close collaboration between everyone involved in the software delivery process.

A key objective for management is making the state of these system-level outcomes transparent, working with the rest of the organization to set measurable, achievable, time-bound goals for these outcomes, and then helping their teams work toward them.

In order to implement continuous delivery, we must create the following foundations:

- **Comprehensive configuration management.** It should be possible to provision our environments and build, test, and deploy our software in a fully automated fashion purely from information stored in version control. Any change to environments or the software that runs on them should be applied using an automated process from version control. This still leaves room for manual approvals—but once approved, all changes should be applied automatically.
- **Continuous integration (CI).** Many software development teams are used to developing features on branches for days or even weeks. Integrating all these branches requires significant time and rework. Following our principle of working in small batches and building quality in, high- performing teams keep branches short-lived (less than one day's work) and integrate them *into* trunk/master frequently. Each change triggers a build process that includes running unit tests. If any part of this process fails, developers fix it immediately.
- **Continuous testing.** Testing is not something that we should only start once a feature or a release is "dev complete." Because testing is so essential, we should be doing it all the time as an integral part of the development process. Automated unit and acceptance tests should be run against every commit to version control to give developers fast feedback on their changes. Developers

should be able to run all automated tests on their workstations in order to triage and fix defects. Testers should be performing exploratory testing continuously against the latest builds to come out of CI. No one should be saying they are "done" with any work until all relevant automated tests have been written and are passing.

Implementing continuous delivery means creating multiple feedback loops to ensure that high-quality software gets delivered to users more frequently and more reliably.[2] When implemented correctly, the process of releasing new versions to users should be a routine activity that can be performed on demand at any time. Continuous delivery requires that developers and testers, as well as UX, product, and operations people, collaborate effectively throughout the delivery process.

THE IMPACT OF CONTINUOUS DELIVERY

In the first few iterations of our research from 2014-2016, we modeled and measured a number of capabilities:

- The use of version control for application code, system configuration, application configuration, and build and configuration scripts
- Comprehensive test automation that is reliable, easy to fix, and runs regularly
- Deployment automation
- Continuous integration

- Shifting left on security: bringing security—and security teams—in process with software delivery rather than as a downstream phase
- Using trunk-based development as opposed to long-lived feature branches
- Effective test data management

Most of these capabilities are measured in the form of constructs, using Likert-type questions.[3] For example, to measure the version control capability, we ask respondents to report, on a Likert scale, the extent to which they agree or disagree with the following statements:

- Our application code is in a version control system.
- Our system configurations are in a version control system.
- Our application configurations are in a version control system.
- Our scripts for automating build and configuration are in a version control system.

We then use statistical analysis to determine the extent to which these capabilities influence the outcomes we care about. As expected, when taken together, these capabilities have a strong positive impact on software delivery performance. (We discuss some of the nuances of how to implement these practices later in this chapter.) However, they also have other significant benefits: they help to decrease deployment pain and team burnout. While we have heard in the organizations we work with anecdotal evidence of these quality-of-work benefits for years, seeing

evidence in the data was fantastic. And it makes sense: we expect this because when teams practice CD, deployment to production is not an enormous, big-bang event—it's something that can be done during normal business hours as a part of regular daily work. (We cover team health in more depth in Chapter 9.) Interestingly, teams that did well with continuous delivery also identified more strongly with the organization they worked for—a key predictor of organizational performance that we discuss in Chapter 10.

As discussed in Chapter 3, we hypothesized that implementing CD would influence organizational culture. Our analysis shows that this is indeed the case. If you want to improve your culture, implementing CD practices will help. By giving developers the tools to detect problems when they occur, the time and resources to invest in their development, and the authority to fix problems straight away, we create an environment where developers accept responsibility for global outcomes such as quality and stability. This has a positive influence on the group interactions and activities of team members' organizational environment and culture.

In 2017, we extended our analysis and were more explicit in how we measured the relationship between the technical capabilities that were important to CD. To do this, we created a first-order continuous delivery construct. That is, we measured CD directly, which gave us insights into a team's ability to achieve the following outcomes:

- Teams can deploy to production (or to end users) on demand, throughout the software delivery lifecycle.

- Fast feedback on the quality and deployability of the system is available to everyone on the team, and people make acting on this feedback their highest priority.

Our analysis showed that the original capabilities measured in 2014-2016 had a strong and statistically significant impact on these outcomes.[4] We also measured two new capabilities, which also turned out to have a strong and statistically significant impact on continuous delivery:

- A loosely coupled, well-encapsulated architecture (this is discussed in more detail in Chapter 5)
- Teams that can choose their own tools based on what is best for the users of those tools

We show these relationships in Figure 4.1.

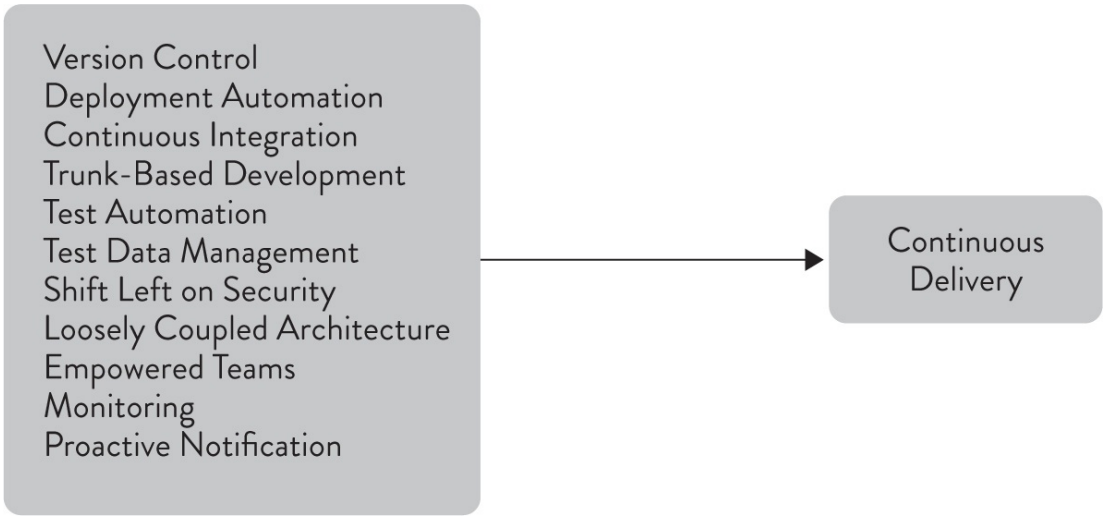

Figure 4.1: Drivers of Continuous Delivery

Since achieving continuous delivery for the sake of continuous delivery is not enough, we wanted to investigate its impacts on

organizations. We hypothesized that it should drive performance improvements in software delivery, and prior research suggested it could even improve culture. As before, we found that teams that did well at continuous delivery achieved the following outcomes:

- Strong identification with the organization you work for (see Chapter 10)
- Higher levels of software delivery performance (lead time, deploy frequency, time to restore service)
- Lower change fail rates
- A generative, performance-oriented culture (see Chapter 3)

These relationships are shown in Figure 4.2.

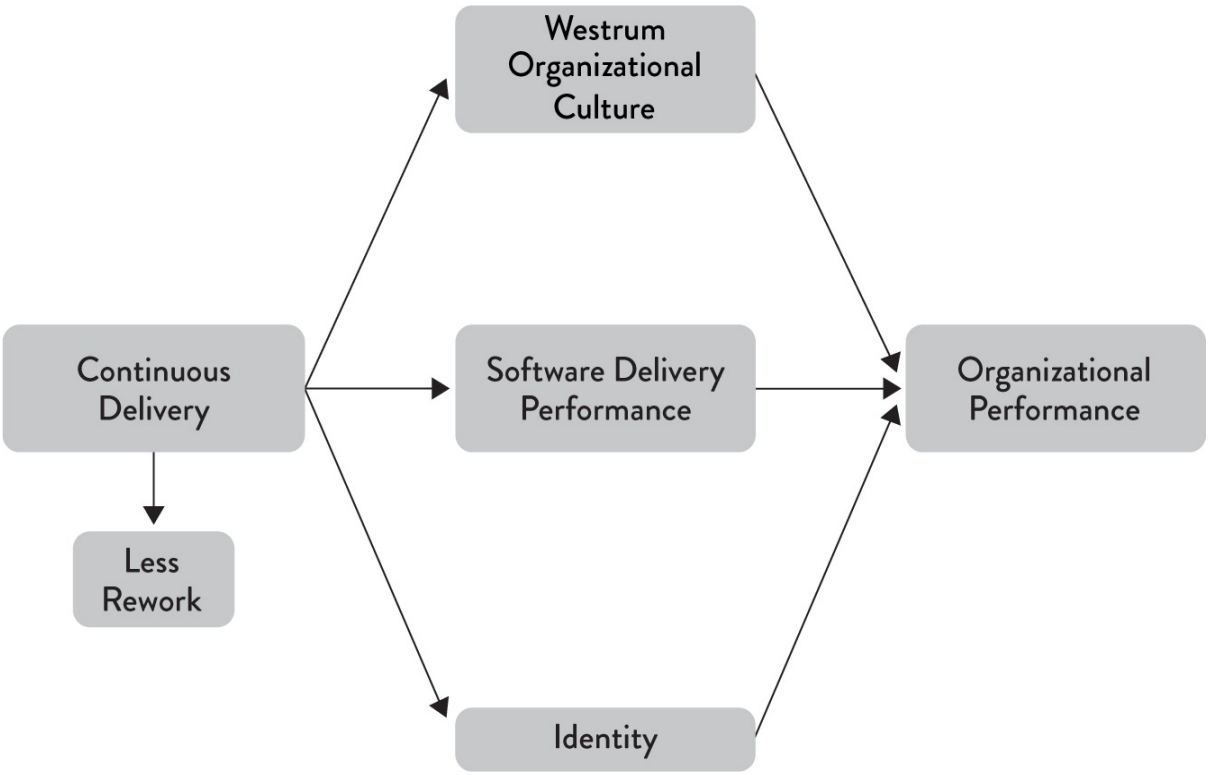

Figure 4.2: Impacts of Continuous Delivery

Even better, our research found that improvements in CD

brought payoffs in the way that work *felt*. This means that investments in technology are also investments in people, and these investments will make our technology process more sustainable (Figure 4.3). Thus, CD helps us achieve one of the twelve principles of the Agile Manifesto: "Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely" (Beck et al. 2001).

- Lower levels of deployment pain
- Reduced team burnout (see Chapter 9)

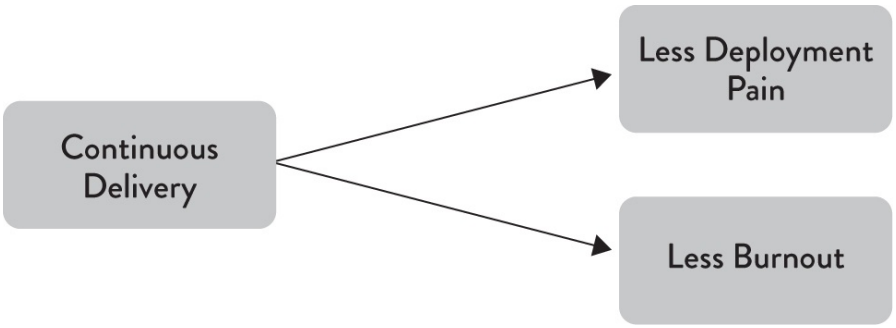

Figure 4.3: Continuous Delivery Makes Work More Sustainable

THE IMPACT OF CONTINUOUS DELIVERY ON QUALITY

A crucial question we wanted to address is: Does continuous delivery increase quality? In order to answer this, we first have to find some way to measure quality. This is challenging because quality is very contextual and subjective. As software quality expert Jerry Weinberg says, "Quality is value to some person"

(Weinberg 1992, p. 7).

We already know that continuous delivery predicts lower change fail rates, which is an important quality metric. However, we also tested several additional proxy variables for quality:

- The quality and performance of applications, as perceived by those working on them
- The percentage of time spent on rework or unplanned work
- The percentage of time spent working on defects identified by end users

Our analysis found that all measures were correlated with software delivery performance. However, the strongest correlation was seen in the percentage of time spent on rework or unplanned work, including break/fix work, emergency software deployments and patches, responding to urgent audit documentation requests, and so forth. Furthermore, continuous delivery predicts lower levels of unplanned work and rework in a statistically significant way. We found that the amount of time spent on new work, unplanned work or rework, and other kinds of work, was significantly different between high performers and low performers. We show these differences in Figure 4.4.

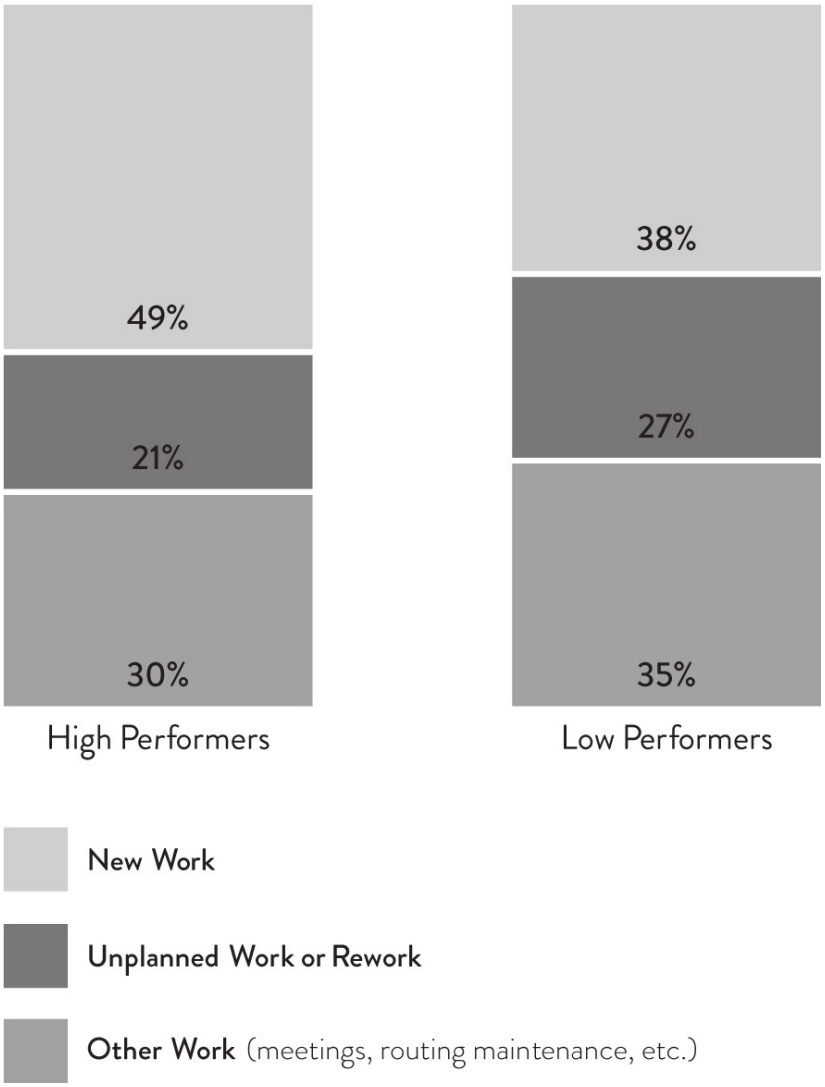

Figure 4.4: New Work vs. Unplanned Work

High performers reported spending 49% of their time on new work and 21% on unplanned work or rework. In contrast, low performers spend 38% of their time on new work and 27% on unplanned work or rework.

Unplanned work and rework are useful proxies for quality because they represent a failure to build quality into our products. In *The Visible Ops Handbook,* unplanned work is described as the difference between "paying attention to the low fuel warning light on an automobile versus running out of gas on the highway" (Behr et al. 2004). In the first case, the organization can fix the problem

in a planned manner, without much urgency or disruption to other scheduled work. In the second case, they must fix the problem in a highly urgent manner, often requiring all hands on deck—for example, have six engineers drop everything and run down the highway with full gas cans to refuel a stranded truck.

Similarly, John Seddon, creator of the Vanguard Method, emphasizes the importance of reducing what he calls failure demand— demand for work caused by the failure to do the right thing the first time by improving the quality of service we provide. This is one of the key goals of continuous delivery, with its focus on working in small batches with continuous in-process testing.

CONTINUOUS DELIVERY PRACTICES: WHAT WORKS AND WHAT DOESN'T

In our research, we discovered nine key capabilities that drive continuous delivery, listed earlier in this chapter. Some of these capabilities have interesting nuances which we'll discuss in this section—with the exception of architecture and tool choice, which get a whole chapter to themselves (Chapter 5). Continuous integration and deployment automation are not discussed further in this chapter.

VERSION CONTROL

The comprehensive use of version control is relatively

uncontroversial. We asked if respondents were keeping application code, system configuration, application configuration, and scripts for automating build and configuration in version control. These factors together predict IT performance and form a key component of continuous delivery. What was most interesting was that keeping system and application configuration in version control was more highly correlated with software delivery performance than keeping application code in version control. Configuration is normally considered a secondary concern to application code in configuration management, but our research shows that this is a misconception.

TEST AUTOMATION

As discussed above, test automation is a key part of continuous delivery. Based on our analysis, the following practices predict IT performance:

- Having automated tests that are *reliable*: when the automated tests pass, teams are confident that their software is releasable. Furthermore, they are confident that test failures indicate a real defect. Too many test suites are flaky and unreliable, producing false positives and negatives—it's worth investing ongoing effort into a suite that is reliable. One way to achieve this is to put automated tests that are not reliable in a separate *quarantine suite* that is run independently.[5] Or, of course, you could just delete them. If they're version-controlled (as they should be), you

can always get them back.

- Developers primarily create and maintain acceptance tests, and they can easily reproduce and fix them on their development workstations. It's interesting to note that having automated tests primarily created and maintained either by QA or an outsourced party is not correlated with IT performance. The theory behind this is that when developers are involved in creating and maintaining acceptance tests, there are two important effects. First, the code becomes more testable when developers write tests. This is one of the main reasons why test-driven development (TDD) is an important practice—it forces developers to create more testable designs. Second, when developers are responsible for the automated tests, they care more about them and will invest more effort into maintaining and fixing them.

None of this means that we should be getting rid of testers. Testers serve an essential role in the software delivery lifecycle, performing manual testing such as exploratory, usability, and acceptance testing, and helping to create and evolve suites of automated tests by working alongside developers.

Once you have these automated tests, our analysis shows it's important to run them regularly. Every commit should trigger a build of the software and running a set of fast, automated tests. Developers should get feedback from a more comprehensive suite of acceptance and performance tests every day. Furthermore, current builds should be available to testers for exploratory

testing.

TEST DATA MANAGEMENT

When creating automated tests, managing test data can be hard. In our data, successful teams had adequate test data to run their fully automated test suites and could acquire test data for running automated tests on demand. In addition, test data was not a limit on the automated tests they could run.

TRUNK-BASED DEVELOPMENT

Our research also found that developing off trunk/master rather than on long-lived feature branches was correlated with higher delivery performance. Teams that did well had fewer than three active branches at any time, their branches had very short lifetimes (less than a day) before being merged into trunk and never had "code freeze" or stabilization periods. It's worth re-emphasizing that these results are independent of team size, organization size, or industry.

Even after finding that trunk-based development practices contribute to better software delivery performance, some developers who are used to the "GitHub Flow" workflow[6] remain skeptical. This workflow relies heavily on developing with branches and only periodically merging to trunk. We have heard, for example, that branching strategies are effective if development teams don't maintain branches for too long—and we agree that

working on short-lived branches that are merged into trunk at least daily is consistent with commonly accepted continuous integration practices.

We conducted additional research and found that teams using branches that live a short amount of time (integration times less than a day) combined with short merging and integration periods (less than a day) do better in terms of software delivery performance than teams using longer-lived branches. Anecdotally, and based on our own experience, we hypothesize that this is because having multiple long-lived branches discourages both refactoring and intrateam communication. We should note, however, that GitHub Flow *is* suitable for open source projects whose contributors are not working on a project full time. In that situation, it makes sense for branches that part-time contributors are working on to live for longer periods of time without being merged.

INFORMATION SECURITY

High-performing teams were more likely to incorporate information security into the delivery process. Their infosec personnel provided feedback at every step of the software delivery lifecycle, from design through demos to helping with test automation. However, they did so in a way that did not slow down the development process, integrating security concerns into the daily work of teams. In fact, integrating these security practices contributed to software delivery performance.

ADOPTING CONTINUOUS DELIVERY

Our research shows that the technical practices of continuous delivery have a huge impact on many aspects of an organization. Continuous delivery improves both delivery performance and quality, and also helps improve culture and reduce burnout and deployment pain. However, implementing these practices often requires rethinking everything—from how teams work, to how they interact with each other, to what tools and processes they use. It also requires substantial investment in test and deployment automation, combined with relentless work to simplify systems architecture on an ongoing basis to ensure that this automation isn't prohibitively expensive to create and maintain.

Thus, a critical obstacle to implementing continuous delivery is enterprise and application architecture. We'll discuss the results of our research into this important topic in Chapter 5.

---

[1] According to Google Trends, Scrum overtook Extreme Programming around January 2006, and has continued to grow in popularity while Extreme Programming has flatlined.

[2] The key pattern which connects these feedback loops is known as a *deployment pipeline,* see https://continuousdelivery.com/implementing/patterns/.

[3] A notable exception is deployment automation.

[4] Only a subset of technical capabilities was tested due to length limitations. See the diagram at the end of Appendix A for these capabilities.

[5] For more information, see https://martinfowler.com/articles/nonDeterminism.html.

[6] For a description of GitHub Flow, see https://guides.github.com/introduction/flow/.