ARCHITECTURE

We've seen that adopting continuous delivery practices improves delivery performance, impacts culture, and reduces burnout and deployment pain. However, the architecture of your software and the services it depends on can be a significant barrier to increasing both the tempo and stability of the release process and the systems delivered.

Furthermore, DevOps and continuous delivery originated in web-based systems, so it's legitimate to ask if they can be applied to mainframe systems, firmware, or to an average big-ball-of-mud enterprise environment (Foote and Yoder 1997) consisting of thousands of tightly coupled systems.

We set out to discover the impact of architectural decisions and constraints on delivery performance, and what makes an effective architecture. We found that high performance is possible with all kinds of systems, provided that systems—and the teams that build and maintain them—are loosely coupled.

This key architectural property enables teams to easily test and deploy individual components or services even as the organization and the number of systems it operates grow—that is, it allows organizations to increase their productivity as they scale.

TYPES OF SYSTEMS AND DELIVERY PERFORMANCE

We examined a large number of types of systems to discover if there was a correlation between the type of system and team performance. We looked at the following types of systems, both as the primary system under development and as a service being integrated against:

- Greenfield: new systems that have not yet been released
- Systems of engagement (used directly by end users)
- Systems of record (used to store business-critical information where data consistency and integrity is critical)
- Custom software developed by another company
- Custom software developed in-house
- Packaged, commercial off-the-shelf software
- Embedded software that runs on a manufactured hardware device
- Software with a user-installed component (including mobile apps)
- Non-mainframe software that runs on servers operated by another company
- Non-mainframe software that runs on our own servers
- Mainframe software

We discovered that low performers were more likely to say that the software they were building—or the set of services they

had to interact with—was custom software developed by another company (e.g., an outsourcing partner). Low performers were also more likely to be working on mainframe systems. Interestingly, having to integrate *against* mainframe systems was not significantly correlated with performance.

In the rest of the cases, there was no significant correlation between system type and delivery performance. We found this surprising: we had expected teams working on packaged software, systems of record, or embedded systems to perform worse, and teams working on systems of engagement and greenfield systems to perform better. The data shows that this is not the case.

This reinforces the importance of focusing on the architectural *characteristics,* discussed below, rather than the implementation details of your architecture. It's possible to achieve these characteristics even with packaged software and "legacy" mainframe systems—and, conversely, employing the latest whizzy microservices architecture deployed on containers is no guarantee of higher performance if you ignore these characteristics.

As we said in Chapter 2, given that software delivery performance impacts organizational performance, it's important to invest in your capabilities to create and evolve the core, strategic software products and services that provide a key differentiator for your business. The fact that low performers were more likely to be using—or integrating against—custom software developed by another company underlines the importance of bringing this capability in-house.

FOCUS ON DEPLOYABILITY AND TESTABILITY

Although in most cases the type of system you are building is not important in terms of achieving high performance, two architectural *characteristics* are. Those who agreed with the following statements were more likely to be in the high-performing group:

- We can do most of our testing without requiring an integrated environment.[1]
- We can and do deploy or release our application independently of other applications/services it depends on.

It appears that these *characteristics* of architectural decisions, which we refer to as testability and deployability, are important in creating high performance. To achieve these characteristics, design systems are loosely coupled—that is, can be changed and validated independently of each other. In the 2017 survey, we expanded our analysis to test the extent to which a loosely coupled, well-encapsulated architecture drives IT performance. We discovered that it does; indeed, the biggest contributor to continuous delivery in the 2017 analysis—larger even than test and deployment automation—is whether teams can:

- Make large-scale changes to the design of their system without the permission of somebody outside the team
- Make large-scale changes to the design of their system without depending on other teams to make changes in their systems or creating significant work for other teams

- Complete their work without communicating and coordinating with people outside their team
- Deploy and release their product or service on demand, regardless of other services it depends upon
- Do most of their testing on demand, without requiring an integrated test environment
- Perform deployments during normal business hours with negligible downtime

In teams which scored highly on architectural capabilities, little communication is required between delivery teams to get their work done, and the architecture of the system is designed to enable teams to test, deploy, and change their systems without dependencies on other teams. In other words, architecture and teams are loosely coupled. To enable this, we must also ensure delivery teams are cross-functional, with all the skills necessary to design, develop, test, deploy, and operate the system on the same team.

This connection between communication bandwidth and systems architecture was first discussed by Melvin Conway, who said, "organizations which design systems . . . are constrained to produce designs which are copies of the communication structures of these organizations" (Conway 1968). Our research lends support to what is sometimes called the "inverse Conway Maneuver,"[2] which states that organizations should evolve their team and organizational structure to achieve the desired architecture. The goal is for your architecture to support the ability of teams to get their work done—from design through to

deployment—without requiring high-bandwidth communication between teams.

Architectural approaches that enable this strategy include the use of bounded contexts and APIs as a way to decouple large domains into smaller, more loosely coupled units, and the use of test doubles and virtualization as a way to test services or components in isolation. Service-oriented architectures are supposed to enable these outcomes, as should any true microservices architecture. However, it's essential to be very strict about these outcomes when implementing such architectures. Unfortunately, in real life, many so-called service-oriented architectures don't permit testing and deploying services independently of each other, and thus will not enable teams to achieve higher performance.[3]

Of course DevOps is all about better collaboration between teams, and we don't mean to suggest teams shouldn't work together. The goal of a loosely coupled architecture is to ensure that the available communication bandwidth isn't overwhelmed by fine-grained decision-making at the implementation level, so we can instead use that bandwidth for discussing higher-level shared goals and how to achieve them.

A LOOSELY COUPLED ARCHITECTURE ENABLES SCALING

If we achieve a loosely coupled, well-encapsulated architecture with an organizational structure to match, two important things

happen. First, we can achieve better delivery performance, increasing both tempo and stability while reducing the burnout and the pain of deployment. Second, we can substantially grow the size of our engineering organization and increase productivity linearly—or better than linearly—as we do so.

To measure productivity, we calculated the following metric from our data: number of deploys per day per developer. The orthodox view of scaling software development teams states that while adding developers to a team may increase overall productivity, individual developer productivity will in fact decrease due to communication and integration overheads. However, when looking at number of deploys per day per developer for respondents who deploy at least once per day, we see the results plotted in Figure 5.1.

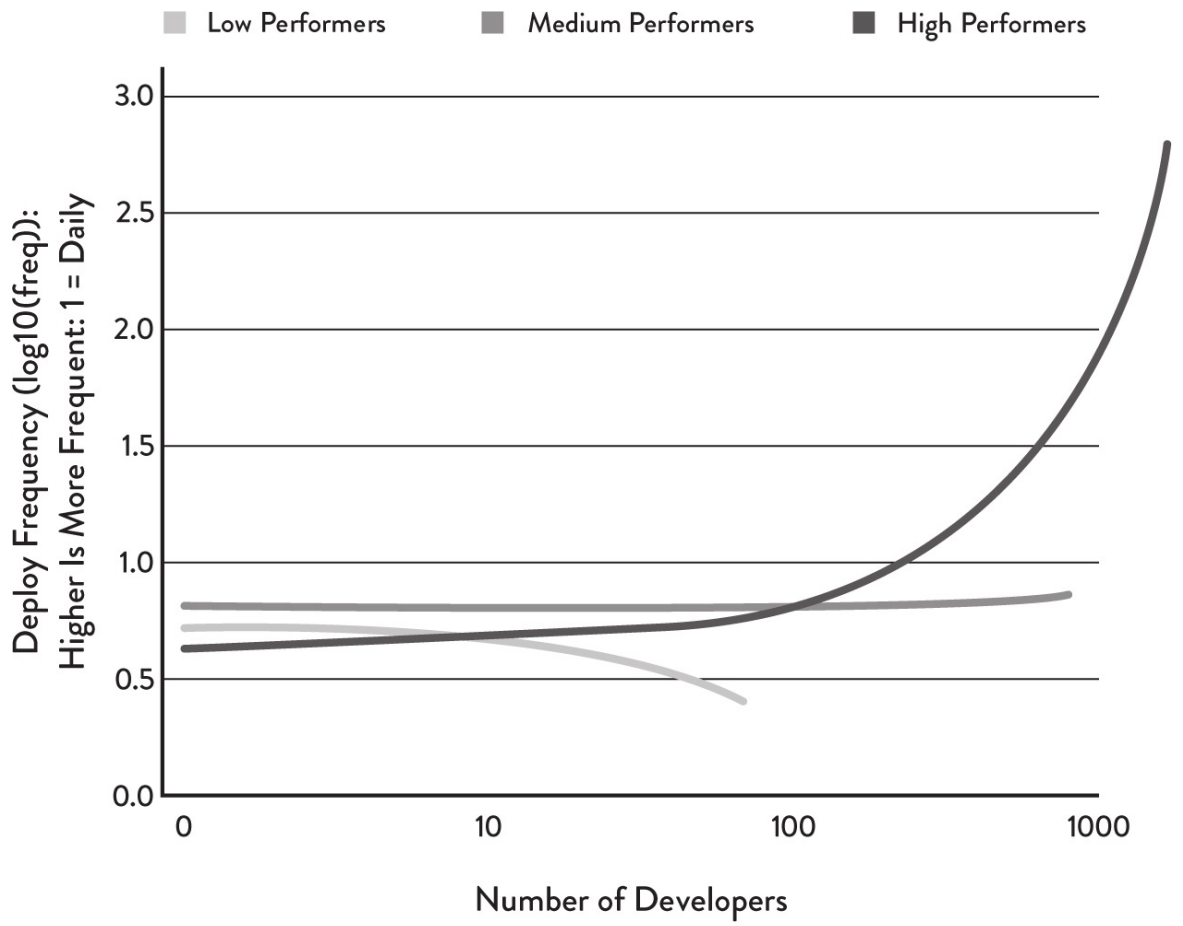

Figure 5.1: Deploys per Developer per Day

As the number of developers increases, we found:

- Low performers deploy with decreasing frequency.
- Medium performers deploy at a constant frequency.
- High performers deploy at a significantly increasing frequency.

By focusing on the factors that predict high delivery performance—a goal-oriented generative culture, a modular architecture, engineering practices that enable continuous delivery, and effective leadership—we can scale deployments per developer per day linearly or better with the number of developers. This allows our business to move *faster* as we add more

people, not slow down, as is more typically the case.

ALLOW TEAMS TO CHOOSE THEIR OWN TOOLS

In many organizations, engineers must use tools and frameworks from an approved list. This approach typically serves one or more of the following purposes:

- Reducing the complexity of the environment
- Ensuring the necessary skills are in place to manage the technology throughout its lifecycle
- Increasing purchasing power with vendors
- Ensuring all technologies are correctly licensed

However, there is a downside to this lack of flexibility: it prevents teams from choosing technologies that will be most suitable for their particular needs, and from experimenting with new approaches and paradigms to solve their problems.

Our analysis shows that tool choice is an important piece of technical work. When teams can decide which tools they use, it contributes to software delivery performance and, in turn, to organizational performance. This isn't surprising. The technical professionals who develop and deliver software and run complex infrastructures make these tool choices based on what is best for completing their work and supporting their users. Similar results have been found in other studies of technical professionals (e.g., Forsgren et al. 2016), suggesting that the upsides of delegating tool choice to teams may outweigh the disadvantages.

That said, there is a place for standardization, particularly around the architecture and configuration of infrastructure. The benefits of a standardized operational platform are discussed at length by Humble (2017). Another example is Steve Yegge's description of Amazon's move to an SOA, in which he notes, "Debugging problems with someone else's code gets a LOT harder, and is basically impossible unless there is a universal standard way to run every service in a debuggable sandbox" (Yegge 2011).

Another finding in our research is that teams that build security into their work also do better at continuous delivery. A key element of this is ensuring that information security teams make preapproved, easy-to-consume libraries, packages, toolchains, and processes available for developers and IT operations to use in their work.

There is no contradiction here. When the tools provided actually make life easier for the engineers who use them, they will adopt them of their own free will. This is a much better approach than forcing them to use tools that have been chosen for the convenience of other stakeholders. A focus on usability and customer satisfaction is as important when choosing or building tools for internal customers as it is when building products for external customers, and allowing your engineers to choose whether or not to use them ensures that we keep ourselves honest in this respect.

ARCHITECTS SHOULD FOCUS ON ENGINEERS AND OUTCOMES, NOT TOOLS OR

TECHNOLOGIES

Discussions around architecture often focus on tools and technologies. Should the organization adopt microservices or serverless architectures? Should they use Kubernetes or Mesos? Which CI server, language, or framework should they standardize on? Our research shows that these are wrong questions to focus on.

What tools or technologies you use is irrelevant if the people who must use them hate using them, or if they don't achieve the outcomes and enable the behaviors we care about. What is important is enabling teams to make changes to their products or services without depending on other teams or systems. Architects should collaborate closely with their users—the engineers who build and operate the systems through which the organization achieves its mission—to help them achieve better outcomes and provide them the tools and technologies that will enable these outcomes.

---

[1] We define an integrated environment as one in which multiple independent services are deployed together, such as a staging environment. In many enterprises, integrated environments are expensive and require significant set-up time.

[2] See https://www.thoughtworks.com/radar/techniques/inverse-conway-maneuver for more information.

[3] Steve Yegge's "platform rant" contains some excellent advice on achieving these goals: http://bit.ly/yegge-platform-rant.