An Integrated Approach to Enterprise Improvement

Bryan Finster

IT REVOLUTION

Author

Bryan Finster

Bryan Finster is an experienced value stream architect, technical lead, product owner, and passionate advocate for organizational improvement required to optimize the flow of delivery. With over two decades of experience developing enterprise supply chain solutions, he applies lean supply chain thinking to the problem of software delivery to minimize costs and maximize delivered quality.

He is a practitioner and mentor of continuous delivery practices and DevOps principles who uses data-driven approaches to optimize the flow of value, improve business outcomes, and improve the lives of software engineers.

"Transformations" are happening everywhere. Enterprises facing disruptors are starting, failing, and re-starting "Digital," "Agile," and "DevOps" transformations. Why is there so much churn?

When these transformations occur, they typically follow the pattern of "We will standardize on this scaling framework we were sold" or "Unicorn, Inc. is successful so we should copy what they say at conferences." They focus their transformation on duplicating someone's process and then cookie-cutter scaling that process across the enterprise. New tools are purchased, consultants are hired, but outcomes don't improve. Maybe there's a talent problem?

In this paper, I will explore the strategic and tactical changes that can be applied over time to systematically improve the flow of delivery, improve quality, and improve employee and customer satisfaction.

Why Do Transformations Continue to Fail?

Transformations fail because the goals of the transformation are wrong. We do not want a sudden, massive change where we reorganize the company, re-align titles, and change everyone to new ways of working. What we want to do is improve our ability to deliver value in measurable ways.

We actually want to avoid transformations. "Transformation" means "change," not "improvement." And big changes have big failures. How can we re-align to an improved flow of small, valuable changes by starting with a big-batch approach? We need real goals.

If we give teams the goal of "changing ways of working" and measure them against a "Maturity Model™" we will not get the outcomes we expect. Teams may change ceremonies and initiate cargo-cult capabilities, but the organization will not see systemic improvements in the flow of value. There are no cookie-cutter solutions, especially if we don't understand the problems we are trying to solve. The reality is that the organization is delivering exactly the way it is designed to deliver. To improve the outcomes, we need to improve the organization.

DevOps maturity and Agile maturity are not goals. Implementing "best practices" is not a goal. What are our goals? How do we reorganize our work to achieve those goals? What outcomes do we expect to achieve? How do we measure them? Achieving the outcomes we want requires a comprehensive and disciplined approach to improvement.

Change is not our goal. Our goal is to deliver value, as Jonathan Smart says, "sooner, safer, and happier."[1] We want higher delivered quality, delivered more frequently with a lower total cost of ownership, happier teams, happier customers, and we want to continuously improve how we do that. First, let's start by looking at ways to improve flow.

Methods to Improve Flow

Continuous Delivery (CD)

Continuous delivery (CD) is the ability to deliver the latest changes on demand while focusing on minimizing change size to minimize risk and improve time to market. It is a disciplined approach to the development of any system, not only greenfield development or microservice architectures. CD is a very effective tool for improving quality, efficiency, value delivery to the end user, and the overall flow of product change through the organization.

This is because the demands of the process are very effective at uncovering waste. With computers eating the world, you can see the impact of this with Tesla products. Tesla implements CD as a philosophy in everything they do, and Tesla owners continuously receive upgrades to their cars, which increases the value of the product they've purchased. It's hard to argue the disruption this is causing to heritage auto manufacturing.

The discipline of CD means we can release small, low-risk changes and get rapid feedback on the quality of those changes. This allows us to rapidly pivot on those changes or to exploit the outcomes to add more value. This is true agility instead of Agile. There are no reasons why CD cannot be done, only excuses for why it is not being done yet.

Focusing on implementing a true CD workflow is also an effective tool for changing the mindset that is so critical for better outcomes. Solving the culture, process, and engineering problems of how to release small changes very rapidly requires us to find and remove overhead and waste in our current processes. Instead of fighting the current process and failing at CD, we use the goals of CD to fix the process, as follows:

1. We should deploy daily.
2. We need a consistent flow of prioritized product features
3. We should integrate everyone's code on the trunk at least daily.
4. We should not have human touch points after code reaches version control.

1 Jonathan Smart, with Zsolt Berend, Myles Ogilvie, and SImon Rohrer, Sooner Safer Happier: Antipatterns and Patterns for Business Agility (Portland, OR: IT Revolution, 2020).

Shrinking Timeboxes

> "If you can't get everything done at your current sprint length, consider shortening your sprint." —**Ron Jeffries**[2]

One of our goals is more rapid feedback of delivered quality. To achieve this we need to expose the reasons that feedback gets delayed. The most direct route is by shrinking timeboxes. We set a goal for a maximum time from when we start work until it is delivered to the user, and then find and fix everything that prevents that.

Why can't we fit all of the activities to deliver a story in five days instead of fourteen? Why not two? In 1955, C. Northcote Parkinson wrote, "Work complicates to fill the available time."[3] In error-tolerant design, a forcing function is a built-in constraint to prevent common user errors. In our case, shorter iterations act as a forcing function for improving quality. With a shorter iteration, we are required to create smaller units of work. To make the work smaller, we must get into the details of what we are delivering, and uncover the assumptions and dependencies we may have missed. This improves quality and allows us to deliver value sooner with less toil, rework, and stress.

There are several common reasons for long development cycle times and large change-set sizes. Anything that adds process overhead or delay while we wait on something else will increase the size of changes and the risk of issues.

Limiting External Approvals

Studies show that external change approvals are less effective for delivering quality than having no approval process at all.[4] Internal change approvals with automated quality gates are far more effective because they help keep the size of a change small, and the people making the change are the most qualified to judge readiness. Smaller changes deliver value faster and are easier to validate and fix. It's far better to have the pipeline and the team approve changes.

Team/Application Alignment

Having an "entangled" team architecture (see Figure 1) where a system is maintained by many teams, with no specific capabilities owned by each team, extends development time because of poor communication, the difficulty in testing, and the uncertainty of how a change in one area will impact another area. This is an expected outcome of ignoring the importance of team architecture.

---

2 Ron Jeffries, "Small Topics: Done, and Spring Length," RonJeffries.com, June 12, 2017, https://ronjeffries.com/articles/017-02ff/small-topics/.

3 "Parkinson's Law," https://www.economist.com/news/1955/11/19/parkinsons-law

4 Nicole Forsgren, Ph.D., Jez Humble, and Gene Kim, *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations* (Portland, OR: IT Revolution, 2018), 75–82.

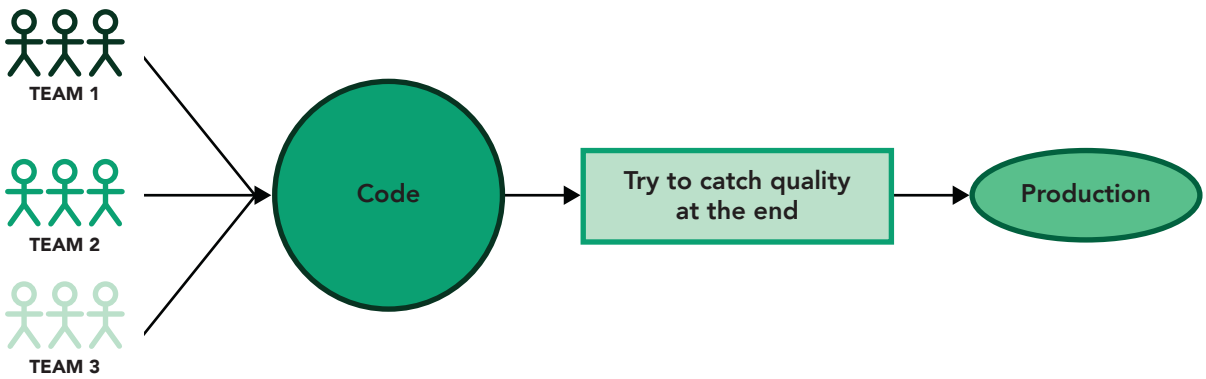

Figure 1: Entangled Monolith

First, it's important to understand the impact of Conway's Law, which predicts that your system architecture will reflect your communication structure. Imagine a system that is designed by the HR department instead of by engineers.

Aligning teams to capabilities in the system and architecting continuous integration automation that enables faster feedback from tests is the first step to improving this. In Figure 2, a large monolith is organized into capabilities owned by individual teams, and the CI automation provides feedback to each team if a change breaks the system.

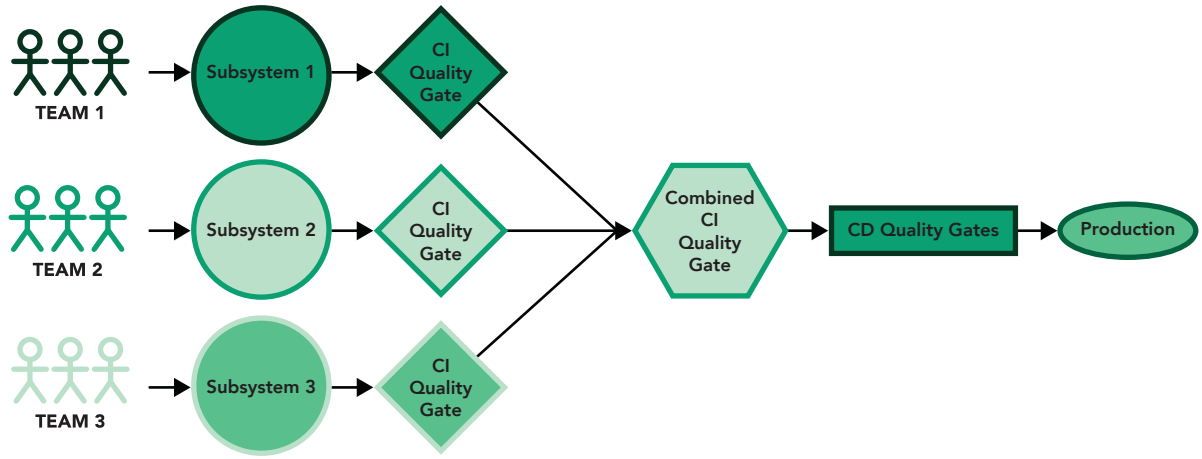

Figure 2: Monolith Organized Into Capabilities Owned by Teams with CI Automation

Decomposing the monolith into individual delivery flows yields faster and more reliable quality feedback, and allows smaller deliveries to occur more frequently (see Figure 3).

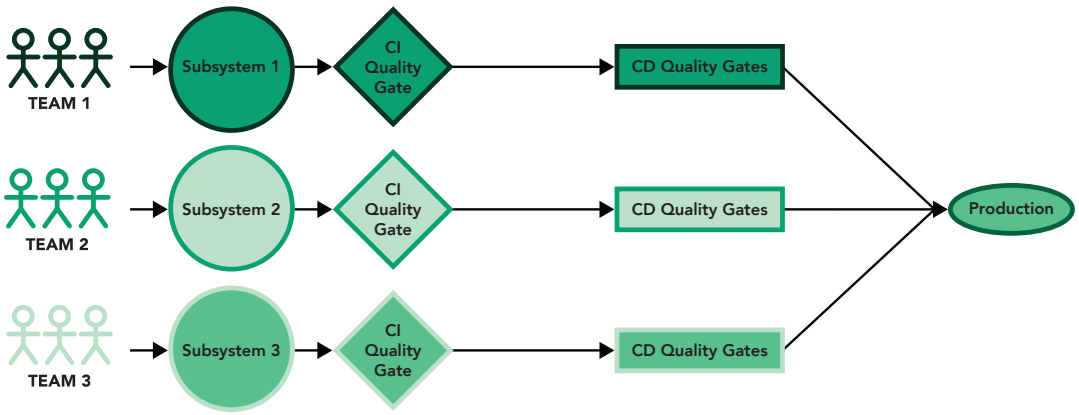

Figure 3: Individually Deliverable Components

Test Architecture

Testing isn't a phase, but many organizations suffer from the mindset that developers code and testers verify the code. This leads to another handoff.

In Figure 4 presents a heritage workflow, where developers and testers are provided requirements from an external product management team. The result is multiple points of communication failure and long feedback loops that result in poor outcomes.

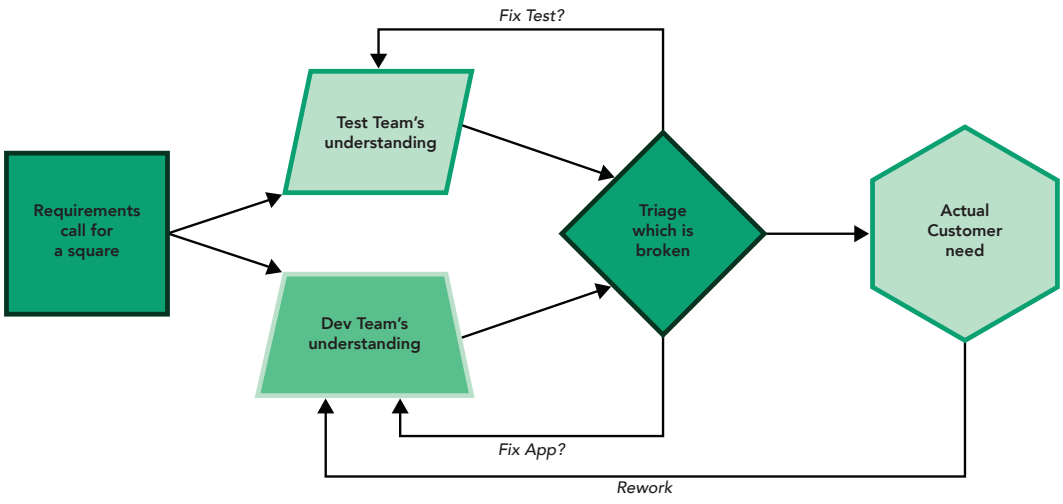

Figure 4: Handoff Driven Development

Interfaces between systems or people are where most defects are born due to miscommunication. With software, we can test for data corruption at interfaces. With teams, we must organize around minimizing communication handoffs to reduce where corruption can occur. This means that refining work,

testing and development happens concurrently and by the same people. The most effective process is always "test first," with requirements defined with acceptance tests in plain English, which can then be implemented by the developers.

Test architecture should be optimized for early detection, preferably detecting defects as they are created. We should never write a large end-to-end test that requires multiple components to run if we can design faster and less expensive tests to validate the behavior.

Proper testing requires good engineering and a "test first" mindset and workflow. This can only occur when a culture of professionalism exists where developers see quality as their responsibility. Testing is not "extra work" or "optional." A professional developer delivers working solutions that they know work because they tested them. Any amateur can work in a feature factory delivering code.

Where Do We Start Improving?

To start improving, we need to find skilled people who know how to improve things to achieve our goals. The good news is that those people are already in your organization. Growing those abilities is a matter of giving motivated people the right problems to solve. This can be accelerated by hiring externally to help kick-start the first effort, but make sure they are people who "do" and not just "talk."

You need to grow the skills within the organization, so the core team needs to focus on pushing knowledge out as they learn it. They need to find ways to enable others to improve with patterns, examples, and shared experiences. Avoid building them into process gatekeepers or a Center of Excellence (CoE). CoEs are where good ideas go to die. The core team should not be telling other teams how to work. Their focus should be on helping teams solve the problems so that the teams own the solutions.

Aim Small, Miss Small

Start with one or two product teams. These should be aligned to specific business capabilities. If they're a project team or a feature team they will not be able to establish quality ownership. Project teams will disband after the project, and the improvements will disappear with them. Feature teams have no ownership of outcomes or long-term quality feedback because they build things for others to own.

Our goal is to grow a team that focuses on continuously improving what they deliver and how they deliver it. That requires long-term ownership. If teams are already organized around product capabilities, there's no need to assemble special "unicorn" development teams for this effort. It will slow things down to form new teams since existing teams already understand how to work together and understand their business problem space. "Unicorn" teams will also undermine the message that we need to change *how* we work, not *who* is doing the work. Any normal product team can improve rapidly when given the correct problems to solve and the ownership to solve them.

Our first effort is to improve value by reducing waste. Delivered customer value is a trailing indicator, but waste can be measured very early in the flow, and is a leading indicator for how well we can

deliver potential customer value. If we are delivering the same experience to the end-user but can do it safer, with less manual processes, fewer meetings, and less rework, then we improve the value proposition.[5] We need to make waste visible and systematically remove it.

To find the obvious waste, a value stream map is a valuable tool. We can identify where we have handoffs, long wait times, and quality problems. We can also quickly see how much money we are spending on each step.

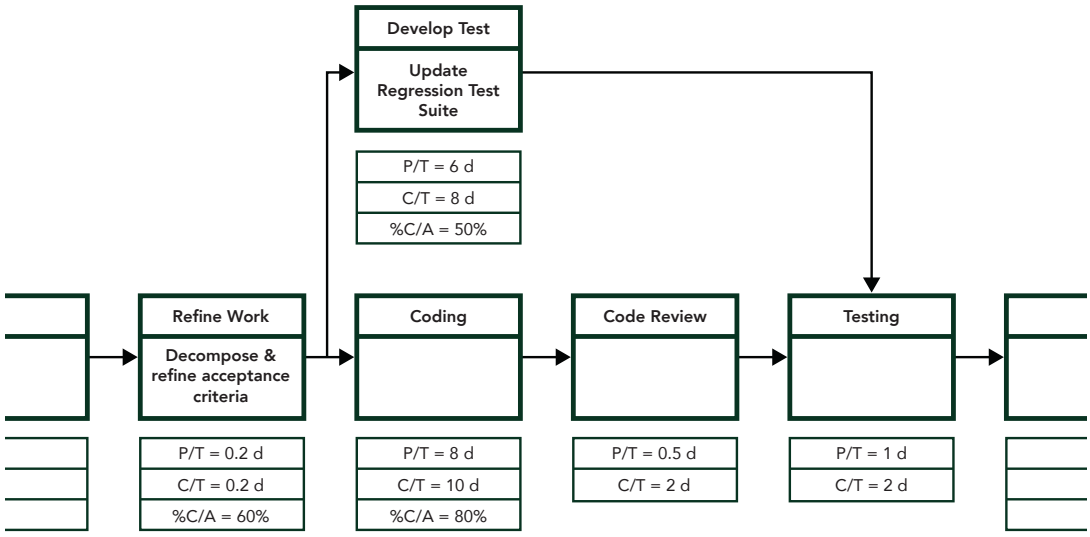

Figure 5: Partial Value Stream with Areas for Improvement

In Figure 5 we have a partial value stream with several opportunities for improvement. The process time is the time spent working on an activity. The cycle time for each step is the process time plus time spent waiting to start the action. For example, if a code review takes four hours to complete but a code change typically waits a day and a half before someone starts the activity, then the process time is four hours but the cycle time is two days.

In Figure 5 the total time between when coding begins and when testing is completed is fourteen days, assuming no rework. It shows that a typical user story will take a couple of hours to refine, but only 60% of the time does the outcome of refining contain all of the information needed, so 40% of the time someone will ask for clarification after they begin work. These same stories are used by the development team and the testing team, so any lack of clarity impacts both teams, probably in different ways.

Two days later coding begins. After eight days of coding and more time for code review, the development team and the testing team come together to find out which of their efforts is wrong or if both are. This is in no way a worst-case scenario.

5 Paul Hammant, "Call to Arms: Average Story Sizes of One Day," PaulHammant.com, April 24, 2012.

Now that we have an idea of where the pain is, we can begin addressing it. However, to fix the multiple problems in the flow it's not enough to test better or refine the stories better. The way the flow is structured in Figure 5 is fundamentally flawed and will deliver low quality due to the handoffs and the long delays in feedback. We need to find ways to systematically improve this process and the culture that created it.

Continuous Delivery Requires Continuous Integration

As we've already seen, continuous delivery (CD) is much more than automating delivery. CD is how we use the tools to deliver better value sooner. This starts with continuous integration. Continuous integration (CI) is also a way of working that uses automation. The focus of CI is to integrate changes from every developer very frequently so everyone is communicating with each other through code. This reduces change conflict issues and improves quality by shrinking change set size, making peer review more effective, and reducing the number of defects each change could contain.

There are rules for CI:

- All changes originate from the trunk and integrate back to the trunk at least daily.
  - This can be done with branches as long as the branches last no longer than one day.
- Changes include all required test automation changes.
  - ALL tests in the flow, not just unit tests.
- Changes need not be complete, but they must be deployable without breaking.
- We can get feedback from the CI server on defects in less than ten minutes.
- We do not check in changes to a broken build.

These rules are important, and the discipline of CI is critical to achieving our goals. There are common reasons teams struggle with this that highlight many of the causes for organizational struggles.

Teamwork

Teams should use a pull system where the whole team focuses on delivering the highest priority before moving to the next. This ensures that the whole team has context to the current work and can contribute to the quality and speed of delivery.

It is common for organizations migrating to agile workflows to apply ceremony and jargon but not the underlying principles of Lean manufacturing. Old habits create a system where each developer is assigned a set of tasks to work on. Worse is creating individual ownership of subsystems within the team. This is destructive to the quality and to value delivery. Nothing finishes quickly. There is little or no collaboration within the team. No one has context for someone else's work, so code review is ineffective. CI falls apart. Worse, the organization's goals depend on a single person.

Work Decomposition

Work should not be handed to the team to complete. Product owners are team members who have the responsibility for providing the team the information needed to build the right things in the right order. They should collaborate with the rest of the team to refine the work into acceptance tests. Behavior Driven Development is a powerful tool for this. No work is ready to start until the team agrees with the acceptance tests and that each unit of work can be completed in a few (1–2) days. Smaller is better since smaller batches have fewer defects and fewer unused behaviors.

Team Architecture

Teams should be organized around business capabilities and the flow of value to the customer, not around roles. Teams should be resilient to absence, contain all the skills required to deliver without handoffs, and have total quality ownership of their domain. This ownership means they have the authority for how to solve the problem, the authority to improve, and responsibility for the outcomes.

Incentives

Individuals can have the career goals and learning they want to achieve, but business goals and customer value delivery are the responsibility of the team. Any performance metrics must be focused on team improvement, not individual evaluation. If we incentivize people to look out for themselves or compete with others on the team for individual productivity goals instead of focusing on team outcomes, they will do that. This may make HR decisions easy, but it serves neither our customers nor our organizational goals.

Since the team is responsible, we need to make sure that the team has ownership to prevent knowledge silos that put them at risk. If we assign ownership of something to one person to optimize for HR metrics, we are betting the success of our business on the hope that that person doesn't have a life event that will impact delivery.

We need to overcome these struggles for CI to function, and pursuing CI is the best tool for fixing them. Keep asking "Why can't we deliver changes to the trunk today?" and solve the problems. Understand that CI is about people communicating, not about technology.

The primary product of any team is its CD pipeline. It requires teamwork and a sense of ownership to continuously improving the feedback on the efficiency and effectiveness of the pipeline. By focusing on shrinking the size of everything we do, CD helps us uncover the problems in our flow that are impacting quality and increasing the cost of change.

It is also effective at growing high-performing, high-morale teams who can deliver sustainably at speed and see their work improve the lives of their end users. We learn at the speed of delivery, and we want to learn faster than our competition.

Scaling Things Out

We've seen how we can use CD to improve team outcomes. However, we cannot cut and paste the ways of working between teams. There are too many variables. So, how can we take the lessons learned and horizontally scale?

Leadership Must Lead

Improvement requires an excited and engaged grassroots movement. But the grassroots need excited and engaged leadership to succeed. The most successful change comes from executives guiding and cheering instead of directing. Leadership needs to set clear and measurable improvement goals. They need to speak about them frequently and celebrate improvement toward those goals. They need to be partners who provide space and air cover to those working to improve outcomes.

"We didn't want to change everything! We just wanted everything to change!"

"Every system is perfectly designed to get the results it gets."
—W. Edwards Deming

We are changing everything, including how we think of change. It is a permanent program of continuous improvement. Many consultants will sell shake-and-bake, silver-bullet solutions. "Give us six months and everything will change!" But the only change will be the transfer of capital and frustrated teams and customers.

Our true goal isn't to change but to improve. This is a long-term goal and will touch every aspect of the organization. Structure, culture, policies, processes, and everything else are subordinate to improving value delivery. This is not only a change in how the current teams work. The teams are perfectly evolved to deliver their current outcomes in the environment they are in. To improve the outcomes, we must improve the environment. This has little to do with the level of skill of the teams or the tools. It is due to the culture and structure of the organization, and the problems the teams are given to solve.

Executives need to be active participants in fostering the culture required to successfully improve the environment in their organization. Inspire people to improve, create a culture of open conversation, where anyone feels safe to both suggest improvements and to push back on something they feel is headed in the wrong direction. Lead from the front.

Define Goals

We need clearly defined, objective, and measurable improvement goals communicated from executive leadership and updates from them on how we, as an organization, are doing. These goals need to be aspirational and encourage innovation, not targets that inspire fear of failure. We need recognition for

teams trying new things and publishing their results, not just successes. Experiments mostly fail, but improvement doesn't come from not trying new things. We need to become more efficient, more effective, and retain great people with the domain expertise to compete in the marketplace. We need to be laser focused on increasing value by doing things that make, save, or protect money while improving the lives of our customers.

Efficiency

When looking at efficiency, we must look at the entire value stream, from idea to delivery and operations. Improvements in efficiency must be measurably more efficient across the entire flow. We can measure efficiency by tracking the cost of change, the speed of change, and the delivered quality.

The cost of change is not just the cost of the physical effort to transcribe requirements into code. How much does it cost to have meetings to discuss the change? How much operational support for the new features? If we outsource development to save development costs but must have more meetings to communicate ideas, and we generate more rework and defects, have we saved money across the entire value stream? We need to consider the total cost

Speed isn't about how fast a team can code a feature. Speed is a measure of the whole lead time from request to delivery. Focusing on exposing and removing waste in the value stream is the driving force for shrinking the critical feedback loops. As the feedback loops shrink, more waste is exposed. Remember to focus improvement efforts on the biggest constraint in the flow. No other improvement matters.

As we become more efficient, toil is reduced, and it becomes more economical to run product experiments. This enables us to be more effective at delivering customer value.

Effectiveness

Improving efficiency is important, but we need to be effective at delivering value. Delivering slowly is bad enough, but it's worse if we spend months or years building something only to find out that our customers don't want it.

At the 2018 BlizzCon, Blizzard Games announced the next installment of Diablo, Diablo Immortal. The crowd was excited until the demo showed a new game built for mobile devices instead of PC. The excitement died. One attendee literally asked, "Is this an off-season April Fools' joke?"[6]

We can identify the value goals for each product, and instrument our systems to measure that. However, we need to get feedback rapidly from the customer to validate if our value goals have been met with what we delivered. We can measure what matters to the customer. Would they recommend it to others? Is it stable? Is it buggy? We need to make these better while not degrading efficiency.

6 "The Moment Diablo Died At Blizzcon 2018," YouTube video, posted by Nexius, 2:42, November 3, 2018, https://www.youtube.com/watch?v=MmkHAlhCvWg.

Health

If we are efficient and effective but have heroes keeping our systems stable, we cannot sustain improvement. We need to set goals around stability, both in our systems and in our teams. We cannot build resilient systems without resilient teams. Are teams happy? Do we have retention issues or requests to leave teams? Are there knowledge silos on the team?

If the team lacks ownership, is getting burned out, or is not incentivized to operate as a team, we are putting our business goals at risk. We need to measure this and make sure we have sustainable delivery with continuous improvement.

Define Terms

A common challenge is people talking past each other using words that each understands differently. Resolving this early on will prevent many headaches and align everyone on a common mental model.

Testing language is a good example. There are no industry standard terms for testing patterns. If you ask three people what an "integration test" is, you'll get at least four different answers. To grow the conversations and communities around testing, an effective approach is to assemble a cross-section of engineers to define a glossary and then use those terms exclusively in the organization. Consensus is the best way to achieve this.

Metrics are another domain language we must address. What measures do we use? Why do we measure them and what decisions will we make based on them? How do we prevent metrics from becoming goals instead of indicators? How do we group metrics to prevent gaming or destructive outcomes? We'll talk more on this later, but explicitly documenting the core metrics is key.

Two terms that frequently create issues are "Agile" and "DevOps." Many people will relate "Agile" to some specific process instead of the outcome of being able to respond to the needs of the end-user. "DevOps" is particularly problematic as companies continue to build "DevOps Teams" and hire "DevOps Engineers" to do release management, platform, or support roles. It's tempting to avoid using those words at all and instead focus on the goals. However, if we don't own those words someone else will, and usually to the detriment of our goals.

I've found that these two definitions remove most misunderstandings:

- **DevOps:** "The union of people, process, and products to enable the continuous delivery of value to the end-users," as Donovan Brown puts it.[7]
- **Agile:** A set of principles and practices focused on small batch sizes, rapid delivery, and continuous improvement. Agile processes are a required subset of DevOps.

7 Donovan Brown, "What is DevOps," DonovanBrown.com (blog), September 1, 2015, https://www.donovanbrown.com/post/what-is-devops.

The glossary does not need to be complete on the first attempt but remember it will be the internal contract the organization uses to communicate with itself. Start small and expand, but avoid churn in the definitions. Involve the people who do the work daily to avoid "ivory tower" mandates. People will reject solutions that are imposed upon them. Find passionate people and get their help. They will help you align with others to help improve communication in the enterprise.

Grow a Delivery Platform

To effectively scale change across the organization, we need a common way for teams to deliver changes. Teams need to have options for the tools they use for daily work. However, some non-negotiables must be considered for security and compliance. We also want to encourage a continuous delivery workflow to reduce waste and improve quality feedback. We need common delivery platform tools that have security and compliance guardrails built-in, encourage good practices, and provide delivery metrics observability, but allow teams to configure the platform to meet their needs. This also removes the need for every team to configure and maintain operational support of their delivery tooling. Product teams should focus on their customers, not supporting the tools that allow them to deliver. Besides, if everyone is using common tooling, then the whole community can help solve problems as new teams onboard.

A common platform anti-pattern is creating a "DevOps Team" that builds and maintains delivery pipelines for the product teams. Not only does this not scale well, but it also disconnects the team from their primary product, a pipeline that amplifies quality feedback and increasingly prevents poor outcomes. The teams should own the pipelines. They understand their delivery context. The platform teams should focus on delivering tools that enable the teams to safely build the pipelines.

An effective approach to building the central platform is the same as every other product; start with a high-value customer, product teams who drive core value, and grow to meet the needs of other customers. If the product teams are supporting their tools, accept ownership of those tools and help them migrate to the new platform. There will always be those who resist change. However, when required to meet the increasing standards for security and compliance, it becomes an easier sell that the central platform handles that for them.

Another selling point is to make sure the platform focuses on developer experience (DX). A good DX reduces toil and defects. Any good platform team will see themselves as advocates for their peers on product teams to help them stay safe while also making daily delivery easier. They will see themselves as service providers, not the police.

The observability of delivery flow metrics is often overlooked as a core capability of the platform. Without the ability to collect and display delivery metrics, teams will be blind to the information they need to keep pipelines green and improve the flow of delivery through the team. We will discuss more of this later.

Having a centralized delivery platform as a aervice allows teams to configure flows that meet their needs while allowing the enterprise to standardize security scans, vulnerability scans, compliance checks, and the rest of the non-negotiables in a scalable and auditable way.

The Power of Community

Improving the outcomes of the organization requires the help of the organization. Bridging the gap between the change agents, early adopters, and the rest of the organization requires broadcasting the knowledge, lessons learned, and value to the rest of the organization. It does not scale to have everyone starting from scratch. Everyone should be encouraged to share. Doing this grows more change agents and builds more excitement. For this, we need to build communities.

Some examples of common communities are Communities of Practice (CoP) and Communities of Interest (CoI). CoPs are where practitioners come together to share and formalize common good patterns and practices. You'll commonly see CoPs form around languages, frameworks, testing, and product management. CoIs tend to be less formal. People gather to discuss whatever subject they are focused on.

The most important thing about communities is that they are organic and emergent. They form around a common need in the organization. Typically, a few interested parties want to learn or share so they start a meeting or a chat group around it and try to bring others into the conversation. If it is valuable, the community remains. When it is no longer valuable, it fades away.

The key is that there is explicit permission to form and join communities. While some will assume permission is given, most will feel pressured to focus on their daily work. Improving daily work is the work, and community participation is a valuable tool of improvement. We need to make sure people feel they have time to participate, share, learn, and grow.

Community building is hard and few are willing to take it on. Anyone willing should be vocally encouraged to do so. It cannot be assigned as a task, but it can be encouraged by leadership leading by example. Leaders need to participate and be community members, but not direct or lead communities.

Active participation by leadership is very important as it:

- Shows the organization that improvement work is important and isn't just lip service.
- Demonstrates an understanding of "we need to improve" instead of "you need to improve."
- Makes everyone else feel free to participate in improvement, not only heads-down feature delivery.

Communities are also a rich source of user feedback for the platform products. These are the early adopters and the excited users who will contribute to your success and help sell the value of the platform. The communities grow the culture and make change sustainable and resilient. Invest in them.

Grow Mentors

We will learn many lessons from our first teams; hard-won examples of things that work and don't work. We need to expand that knowledge with action, not talking. We need to grow the ability to directly mentor teams on techniques to move closer to our goals while understanding that each team will have

its own context and challenges. We need to grow people who are skilled at helping teams solve problems and guiding them to solutions, letting them fail small while avoiding falling off a cliff. We need a disciplined approach that is repeatable and teachable, but flexible enough to allow for team context.

In 2006, Google had a problem. They could not improve their ability to deliver because they lacked the discipline of developer testing and CI. They needed to spread testing knowledge and culture across the enterprise. They created the Test Mercenaries to help teams progress through the "Test Certified" program and develop a culture of testing. Two or more members of the Mercenary team worked daily with a team to identify testing issues and help them achieve Test Certified Level 1. This program went on for three years and resulted in a culture of testing that allowed them to accelerate delivery.[8]

Enterprise Dojos

In Japanese, "dojo" means "the place of the way." It is where you go to achieve mastery. It takes mentoring from more experienced people to master any skill, and continuous delivery (CD) is no different. CD is a team sport, and teams must train together to master it. Dojos are an immersive learning experience where teams engage with the Dojo to learn how to improve how work is done using their current backlog. This is distinctly different from normal technical training because the team is learning how to apply skills to their current context. There are many flavors of this, but they typically have a few things in common:

- They are metrics-driven, not process-driven. Desired outcomes and metrics are defined, and all changes are informed by their impact on the metrics.
- They focus on rapid feedback and learning. A typical Dojo challenge will have iterations of one or two per week, with the team reviewing and planning the next improvement step every iteration.

There are anti-patterns for Dojos that have emerged that must be avoided. If a team is "broken," a Dojo cannot fix it. Dojos are also not where teams are sent to accelerate a critical deliverable. Teams cannot be volen-told to learn in a Dojo. Dojos are not punishment for poor performance. Dojos are a reward for teams that want to improve.

A properly formed Dojo helps product teams to succeed by guiding them the advanced skills needed to continuously deliver value in their unique context. The Dojo is an enabling team for improving the flow of enterprise value, not a gatekeeper of practice.

When we think about the implication of that mandate, it is much more than directly mentoring teams. Do teams and the broader organization have the right structure to meet our goals? What organi

8 Mike Bland, "Test Mercenaries," Mike-Bland.com (blog), July 10, 2012, https://mike-bland.com/2012/07/10/test-mercenaries.html.

zational processes are impacting delivery? How do we help leaders understand some of the changes that are required that they may not have anticipated?

Building out a library of good practices, helping to standardize language across the enterprise, helping define appropriate measures that can be standardized to track improvement toward our goals, etc. These and many other broader responsibilities could lie in the domain of the Dojo as they seek to help clear the organizational constraints that prevent the organization from improving the flow of value.

Similar to Google's Test Mercenaries, Dojos have technical and process coaches assigned to every team working full time with them. However, as the size of the organization grows, so does the backlog of the Dojo, resulting in the need to scale. The obvious choice is to grow the size of the Dojo. However, eventually we've created an organization that is too large to be nimble, and we are standardizing things that should require frequent adaptation to change.

Rather than vertically scaling a central organization, embedding the capability across the organization with the coordination of the Dojo scales much better. Train associates outside the Dojo on the methods for helping teams improve and make it a permanent role spread across the company.

Value Stream Architects

> "The Value Stream Architect is an influencer, a consultant, an optimizer, and an architect."
>
> **—Dr. Mik Kersten**[9]

Value Stream Architects mentor teams on technical practices, use data to suggest area improvements, inform the organization on training needs, design new team structures to improve application architecture, collaborate with business specialists to improve value, make suggestions on improved tooling, and everything else needed to better serve the customer at lower cost by improving the flow of value.

This requires a broad set of skills, and growing this skill set across the enterprise can embed improvement into the daily work. It is important, however, that Value Stream Architects work in a common context for improvement.

Certifying Knowledge

To enable common context there needs to be a disciplined approach to growing this skill set; a certification process to demonstrate that candidates understand the concepts, principles, and data-driven approach to improving delivery. With any new skill, we follow the same steps: we learn, apply the lessons to achieve mastery, and then teach what we have learned. Achieving our goals uses the same approach.

---

9 Mik Kersten, "The Age of Software Needs Value Stream Architects," The New Stack, December 12, 2018, https://thenewstack.io/the-age-of-software-needs-value-stream-architects/.

- **Level 1** consists of training[10] and a certification assessment that is focused on how to identify waste, improve flow, and amplify quality feedback. The test should not be a rubber stamp but should fully exercise the knowledge required to lead an improvement effort on a team.
- **Level 2** requires that the candidate apply their knowledge to an improvement project with a product team. They work with an experienced Value Stream Architect and a cohort of other candidates for mentorship and support for solving difficult problems. They will plan what improvements they are attempting and share the measurable outcomes to become certified Value Stream Architects.
- **Level 3** is achieved by pairing with another experienced practitioner to mentor a Level 2 cohort and demonstrate the ability to teach the cohort.

We can incentivize the growth of the knowledge and the roles across the entire enterprise by recognizing associates for attempting this path and for completing improvement projects. By doing so the knowledge and context become a horizontally scaling pull instead of requiring direct action by a central team. This is also a useful approach for aligning the context of value stream improvement through the leadership chain.

A common issue organizations face is how to change the mindset of the layers of management between the executives asking for improvement and the individual contributors experiencing the new ways of working required to do so. By using this learning path, middle management can more quickly understand the tactical changes required and become active participants by helping to lead improvement kata instead of feeling like victims of change.

The Thin Red Line

Helping the organization improve is difficult. There are only a few people in every organization who have the skills and the passion to lead the charge, build communities, build networks, and work with leadership to solve the hard problems. In *The Unicorn Project*, Gene Kim refers to the Red Shirts; the people on the front lines who, out of a passion for better outcomes, knowingly put themselves at risk to push forward improvement.[11]

This thin red line cares deeply about making the change happen. It helps them, it helps their coworkers, and it helps the company where they work. As we stabilize our systems, remove toil, and remove waste and handoffs, trust improves. As we see the outcomes of our work, we feel pride in the outcomes. Morale improves, and good people want to stay and continue to make things better.

10 An example of training material for this effort is the White Belt Certification from *Engineering the Digital Transformation*. https://www.engineeringthedigitaltransformation.com/white-belt-certification/.

11 Gene Kim, *The Unicorn Project: A Novel about Developers, Digital Disruption, and Thriving in the Age of Data* (Portland, OR: IT Revolution, 2019).

It's important to understand how thin this line is. Your goals depend on not taking the Red Shirts for granted. Encourage them and find ways to make that line broader. Don't depend on improvement being driven by a passionate few people not burning out.

How Are We Doing?

Metrics are key to knowing where we are and how we are progressing toward our goals. We need to be careful and deliberate about measuring. Metrics are a tool that can be easily misused. Misuse impacts trust and reduced trust impacts not only data quality for the metrics but business outcomes as well.

To identify and use metrics, we need to first understand the goals. *The Phoenix Project* summarizes them nicely with "The Three Ways."[12]

1. **Systems thinking:** Optimize the whole system of value delivery. How do we improve our ability to meet the customers' needs?
2. **Amplify feedback:** Establish and shrink feedback loops to improve value delivery. How do we get feedback on what we've done at every step and from the end user more rapidly?
3. **Continuous improvement:** Continue the process of identifying and improving constraints in the flow of delivery. What should we make better next? How do we drive more innovation?

We also need to make sure we understand what action we will take with the information from each metric. So, how do we measure that we are meeting these goals? We need to control inventory and reduce cycle times to identify and remove waste in the flow.

Lead Time

What is the total time between receiving a request and delivering a solution? We want lead times to be short to enable us to deliver value while it is still relevant. How do we improve it? There are several large cycle times that we should focus on as well as specific places in the flow where inventory accumulates. Value stream mapping is the most effective way of uncovering the details so we can prioritize what to address first.

Development Cycle Time

We need to track how long it takes from when work starts until we deliver to the end-user. Long cycle times here typically indicate upstream issues with work definition. This also predicts downstream issues

12 Gene Kim, "The Three Ways: The Principles Underpinning DevOps," ITRevolution blog, August 22, 2012, https://itrevolution.com/the-three-ways-principles-underpinning-devops/.

with delivered quality. Attempting to reduce this cycle time uncovers issues with process overhead, ineffective testing, insufficient teamwork, excess WIP, poor work decomposition, handoffs, etc. Value stream mapping is the most effective way of uncovering the details, and reducing this improves most team issues.

Build Cycle Time

In the book *Accelerate* build cycle time is referred to as the "hard lead time."[13] It's the time between making a change and when the change deploys to production. We want this to be very short while also containing the quality gates required to verify functionality, performance, security, etc. "Very short" is less than an hour. When we investigate why we cannot do this, we will typically find issues with testing and application architecture: manual testing, handoffs for testing, non-deterministic tests, tests that rely too much on testing the full application together, etc. How can we make the quality gates more efficient and effective? Reducing this cycle time also helps us reduce code inventory, costs, and waste.

Code Integration Frequency

This is the frequency teams are integrating tested, deployable code to the trunk of version control. If we cannot do this multiple times per day per developer, why not? Continuous integration is a core quality process. Issues here can indicate issues with testing, teamwork, and work definition and decomposition.

Defect and Change Fail Rates

To measure the effectiveness of quality gates, we need to measure the results of our quality process. Is the rate of defect generation improving? What percentage of changes require remediation? Measuring these in production validates our processes. Measuring them in lower environments gives us information on how to improve the feedback loops to alert the developer to defects sooner.

Mean Time To Repair

We need to make sure we have the instrumentation to quickly detect and resolve issues, as well as an effective way to deliver fixes quickly. We measure the average time to repair production to inform improvements needed in detection and repair.

Undelivered Inventory

Inventory is risk and waste. The more inventory we have, the more likely we have problems with either quality or responsiveness to customers' needs. Inventory comes in many forms: work backlog, work in progress, code change, etc. We reduce these to expose and remove waste. Do we have work refined for the next quarter? How much of that will actually be relevant when we get to it, and how will we be able to

13 Nicole Forsgren, Ph.D., Jez Humble, and Gene Kim, *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations* (Portland, OR: IT Revolution, 2018), 75–82.

react to changing priorities? How much undelivered code change do we have? None of that is producing any value and the size of the defects grow with every additional change. Ship it!

Morale: How Does the team Feel?

Happy teams who have ownership of the solution and the outcomes deliver higher quality and more secure solutions. If we push for more efficiency and more effectiveness but don't remember that this work is about people solving problems, we may get short-term improvements, but long-term sustainability will elude us as people burn out and leave.

There are things we should not measure outside of a team or possibly at all. It's easy to fall into the trap of easy numbers that seem to be meaningful but are not:

- **Test coverage:** This isn't a metric. This is a tool for finding untested code. It should not be confused with quality in any way. Asking for this to improve results increases coverage of bad tests and hides untested code. This is worse than having no tests.
- **Velocity:** This is a planning tool, not a performance tool. It tells you when you may get there, not if you should get there at all. It's also the easiest to change without modifying anything about how we work.
- **Individual activity:** Tracking tasks completed or code change metrics at the individual level dis-incentivizes teamwork and incentivizes feature factories and reduced ownership of outcomes. It may be valuable if your goal is to measure typing, but it's destructive to business goals.

Metrics are tricky and if used as goals instead of indicators they cease being useful for either. If used as sticks to drive teams instead of information for helping teams then they cease being useful AND encourage turnover. If not used in offsetting sets, then focusing on one can degrade another. Be careful of what you measure, and make sure you know why and what decisions those measures will inform. Put guardrails in place to prevent measuring from making things worse. Use the information to help. Don't be evil.

Conclusion

It requires strategy and tactics to improve an organization. There are no silver bullets, and big bang changes will lack allowances for the various contexts within the enterprise. Each organization will face its own challenges, so we need to first learn what those challenges are with some small experiments and then use those lessons to grow the improvements. However, we must always be aware that we cannot clone those successes, only find common patterns that can be applied.

There is no need to create special "unicorn" teams to lead improvement. Improving an existing

team to demonstrate what is possible is not difficult. It takes a few weeks, in most cases, to show significant improvement. Scaling that across the larger organization takes a focused strategy of systemic improvement.

We need active participation from all levels of leadership. We need to grow networks and grow improvement as a capability instead of imposing a standardized process. Every level is part of this change and organizational change should be an expected outcome because the current outcomes are a result of the current organization. We cannot jump to the end by copying another organization's processes. First, our culture and context are very different from another organization's. Second, and most importantly, there is no end, only continuous improvement.