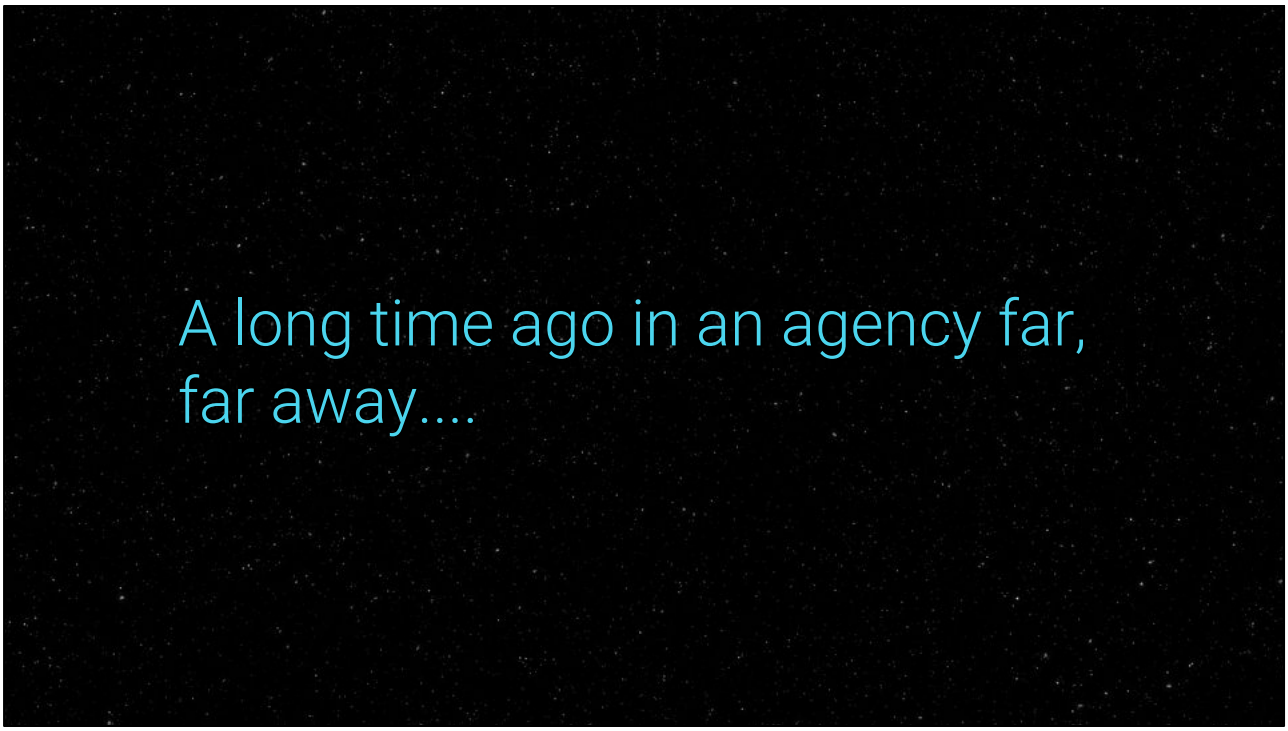
Tech Talk: DevOps

GSA Digital Service



I'd like to open with a real story.

A long time ago, in an agency far, far away…

One of the users of an app needed to search records by zip code. The app data already included zip codes, the search results already printed zip codes, and there was already a search form. The only thing missing was a zip code field on the search form. She contacted IT and asked them to add a zip code search feature. IT sent back an estimate: it would cost tens of thousands of dollars and it would be done next year. Just for a basic search field. That's a big problem for sure, so let's think about what was causing that problem.

The problem couldn't be in requirements analysis: The user story was clear and simple enough.

The problem couldn't be in development: This basic change would have taken the developer a couple of hours to implement and write a unit test.

The problem couldn't be in testing: Even with manual QA testing, this change would only add a few minutes to the testers' work.

The problem couldn't be in security: There was no substantial change to the app, the security controls, the security categorization, the type of data, or any functionality that might affect security or privacy.

The problem couldn't be in operations: Such a minor change that didn't even require a database update wouldn't require any special handling during deployment.

So where was the problem? Within each individual team, everything seemed fine. The problem was in between and, really, it was everywhere. It was the way things were being handed off: each team was using the Wall of Confusion method. This is the traditional method that often comes as a side effect of organizational structures or the waterfall lifecycle, in which each team throws a batch of changes over a wall to the next team. They don't have a clear sight of whether the next team is ready to catch it, and they can only hope the next team doesn't drop it.

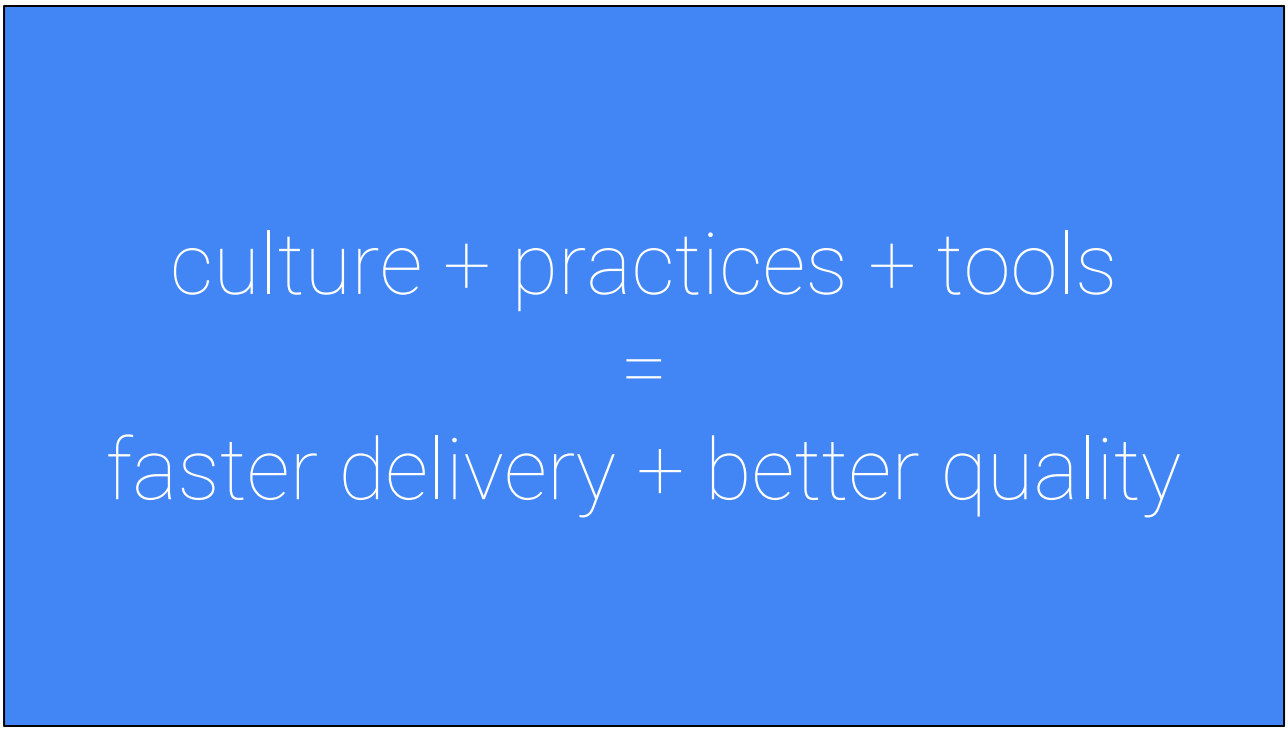
How did this problem translate to an excessive price tag and a lead time of many months? A couple of reasons.

First, the processes were largely manual, so everyone had a tendency to do things in large batches containing many changes. This way, they only had to open one ticket to deploy a few dozen changes, rather than doing a few dozen tickets. The thinking was that the operations staff could just manually deploy one package of changes rather than deploy each individual change. The security team would perform a single scan of the app in the test environment with many changes applied to it. This type of batch work was necessary due to the manual processes. Imagine the hassle of individually deploying many small changes. In their situation, it made perfect sense to go in large batches even if that meant one small change in the batch wouldn't be available until all the other changes were ready.

Second, due to the Wall of Confusion standing between the teams, there was little feedback between the teams. This led to a tendency to kick the can down the road: If the package didn't break horribly on your team, that means it was ready to pass on to the next team. For instance, developers wrote documentation on how to deploy the package, and assumed it would work in production. It didn't always work. In fact, even though operations followed the instructions and the deployment returned no errors, unseen parts sometimes broke. This lack of feedback meant that if something went wrong, it was often discovered at the deployment phase or reported by users. The issue then had to go all the way back to the analysts and developers for rework.

Because of this manual work and the chances of discovering issues late in the process, it was an risky and lengthy process just to incorporate a small change. The IT organization knew the risks involved with introducing even a small change, and estimated it appropriately. That is how they came up with the high price tag and long lead time. Given the situation, their estimate was absolutely correct.

In this Tech Talk, let's see how DevOps could have helped these teams get the change out faster, at a lower cost, and with less risk.

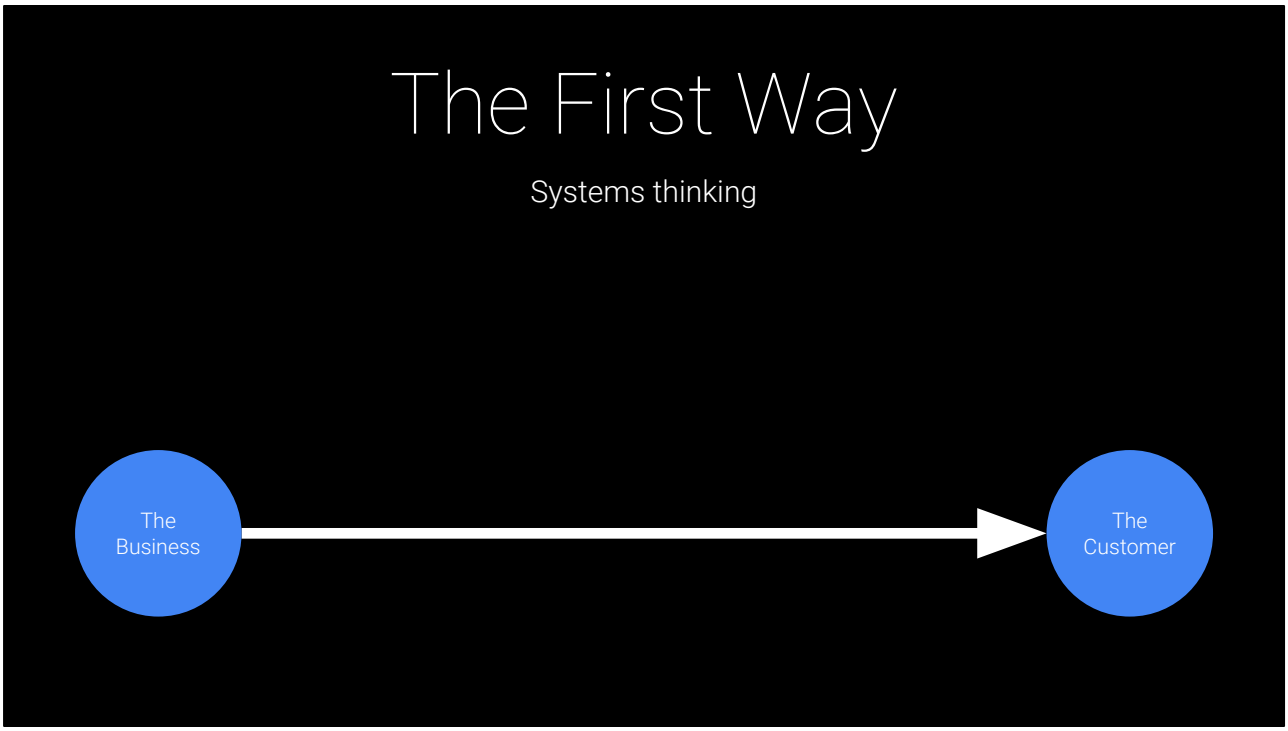


DevOps, ultimately, is about optimizing flow of work through an IT organization in order for the business to more efficiently deliver value to the customer.

It is a collection of practices that unify the various teams that are involved in creating applications. Although DevOps is named after two traditionally siloed organizations, development and operations, it actually brings together analysts, developers, QA, security, operations, and any other teams that are involved in creating applications. DevOps introduces new practices and prescribes the use of automated tools for all of these teams. The goal is to deliver changes faster with better quality. It sounds paradoxical that a methodology that increases the velocity of changes would also improve quality, but this is exactly the effect that one sees in organizations that embrace DevOps.

These days, it's not unusual for a modern technology business like Netflix or Etsy to deploy their product 50 or more times a day.

Today, we'll take a high-level look at DevOps and the practices that enable modern IT organizations to deliver changes so rapidly. While we won't go into specifics on how to build a DevOps shop, I hope this will spark an interest in growing the DevOps culture here at GSA. We'll start by looking at the Three Ways, which were suggested by Gene Kim as the overarching principles of DevOps. He has written several books on the topic, so I recommend looking him up in the library.



The First Way of DevOps is systems thinking. In systems thinking, we depart the territory of siloed teams and think about the overall goals and how the entire system works to deliver an application from the business to the customer. You want this delivery to occur at the highest velocity possible. When you're applying DevOps principles, you need to look at the performance of the entire system, and not the individual silos.

You'll notice that on this diagram, you have the business on the left, and it is delivering services to the customer on the right. Somewhere in between these two actors lies IT and all its teams: development, QA, security, and operations. But in systems thinking, we want to focus on the overall picture. We want this line between the business and the customer to be executed as efficiently as possible. Every decision that IT makes must consider this line on the diagram. No matter what IT organization you work for, your primary goal is making work flow from left to right in the most efficient way possible.

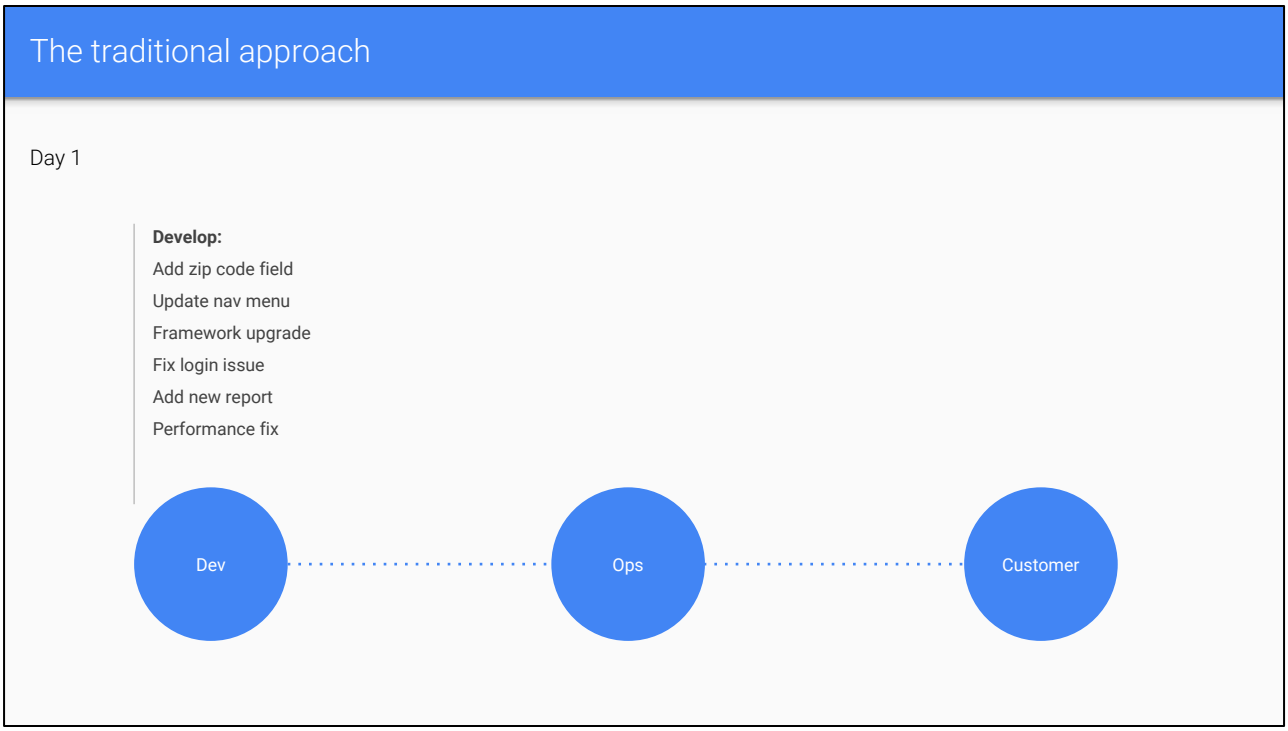
When thinking about how to make this flow more efficient, any optimization a team makes must be an optimization for the overall system. That is, if a team develops a local optimization that helps it meet or exceed its goals, but realizes it will have an adverse effect on the overall system's goals, they should not implement it. Also, any local optimization that has zero net effect on the overall system is possibly a wasted effort; that time could have gone toward more productive work. Likewise, at the management level, organizational and individual performance plans should be written in a way that encourages overall optimization and discourages local optimization if it

comes at the cost of the overall system.

Automation is one important optimization in DevOps. The more the development and deployment process can be automated, the more efficient the overall system becomes. Every organization will have different automation needs, and consequently different tools. There is no one software package out there that provides DevOps in a box. Every successful DevOps shop uses a collection of various tools, each of which excels at its particular job.

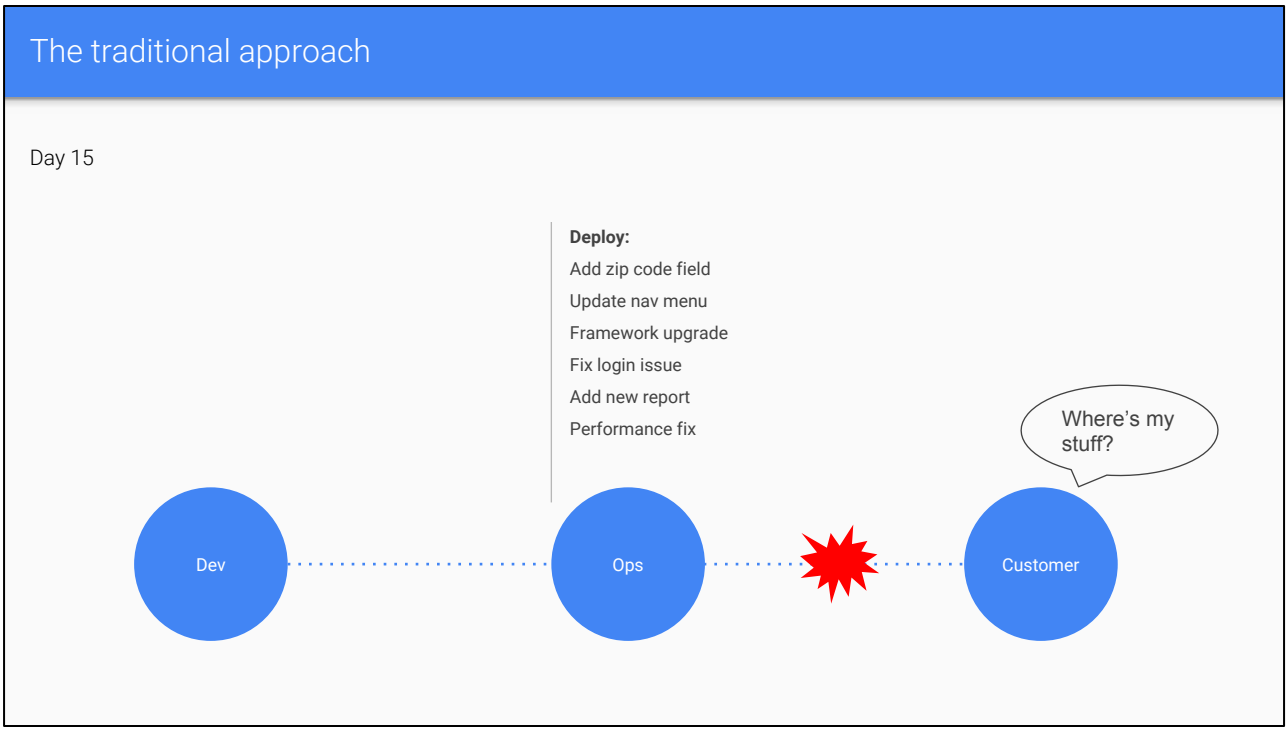
Another general optimization each individual or team can make is to limit how much work they take in at once. This is called work in progress, or WIP. For instance, a developer might be tasked to work only on one user story at a time. The reason we want to limit WIP is so that the work gets done faster. The more WIP a team has, the more they have a tendency to switch between the various units of work before completing one, causing a bottleneck in the overall system.

In parallel to the concept of less WIP, you also want to have smaller batch sizes. Batch size refers to the amount of work that is moved in a group from one step to the next. For instance, a single application update could contain 20 different changes in one release. This means the release has a batch size of 20. In this case, the development team would build a package containing 20 changes and hand it off to operations. Operations would deploy those 20 changes to production in one deployment. In general, the smaller this number, the better. The best case is a batch size of one. But passing work from one step to the next involves a certain amount of overhead, and this overhead dictates how small your batches realistically will get.



I would like to propose a scenario to clarify how smaller batch sizes are beneficial.

Here, we've illustrated a traditional approach: a small development team has taken on six tickets. One of these tickets is a highly anticipated zip code search field that our customer wanted. The plan is to release all of these changes at once in a single deployment. So that means we have a single batch with a size of 6. They only have to open one ticket for operations to deploy everything. All in all, let's say that these changes will take 15 days, or three weeks. The zip code field is only one day, but the other changes are pretty sizable. For instance, the framework upgrade ticket will take 4 days because it's a complex change. So, assuming all goes smoothly, the customer will wait three weeks before they can use their new zip code search feature, but they also receive all the other changes at the same time at the end of those three weeks.



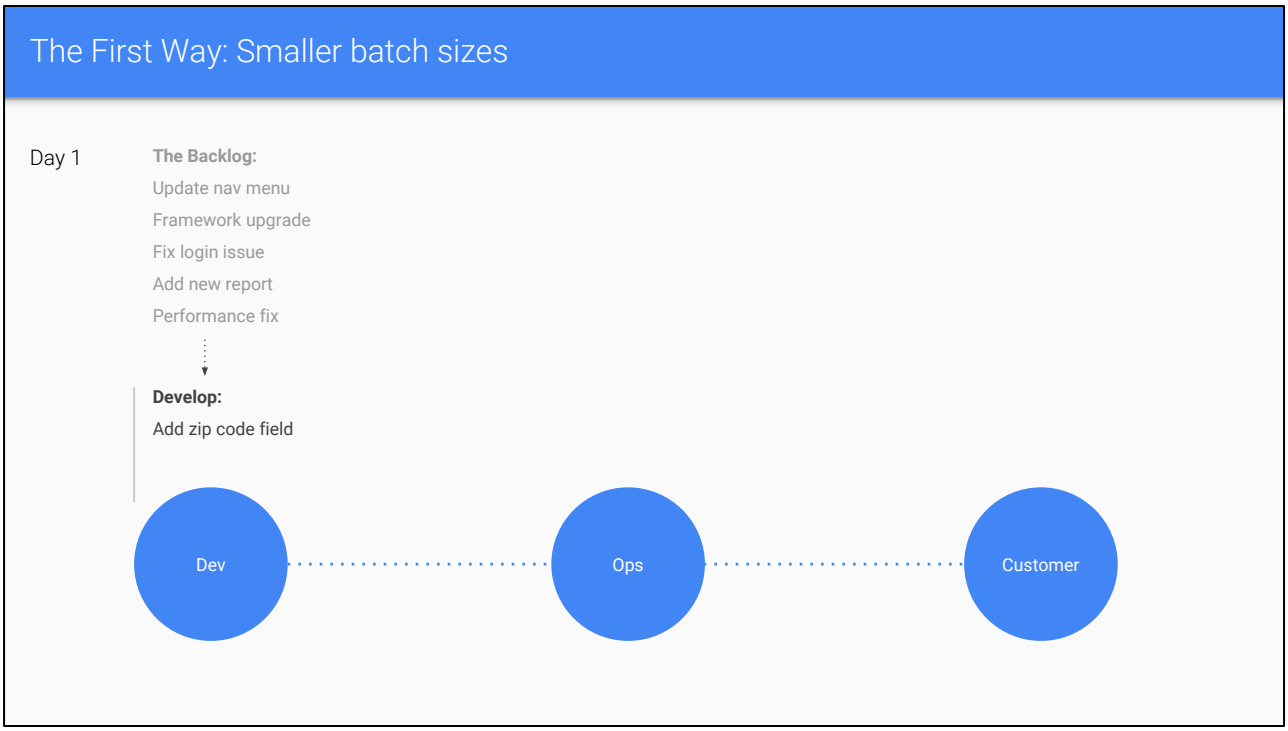
Three weeks have passed. The dev team has finished coding, and has handed off the completed package to operations. During a maintenance window, operations personnel log in to the servers and deploy the updated app.

Unfortunately, it's a no-go, because the framework upgrade isn't working in production. Since all six changes were batched together, operations sends it all back to development, rolling back production to the previous version. Now the operations and development teams have to figure out which change went wrong. It could be any of the six changes. They have to perform an analysis before they can confirm it was the framework upgrade.

Although this means the dev team only has to fix one issue, it delays all six changes. The dev team fixes the framework issue within a few days, and is ready to reattempt the deployment. But now the next maintenance window operations has isn't for another couple of weeks. Operations has to carefully plan out maintenance windows because they only have so many system administrators that can perform a deployment at any given time.

We eventually do get a happy ending: the customer got their new features. But because of one failure, it took a long time for the customer to get any of these features, even the small ones that only took a few hours to implement.

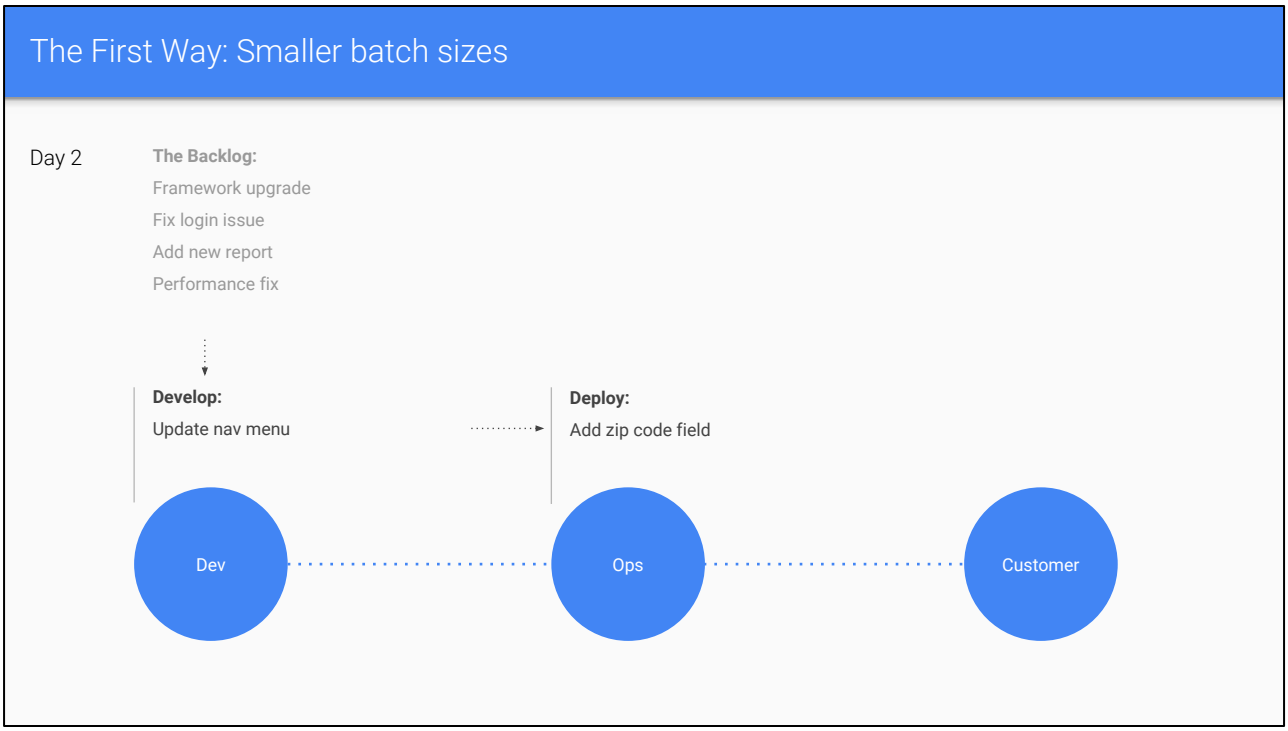
You can bet there's a better way to do this.



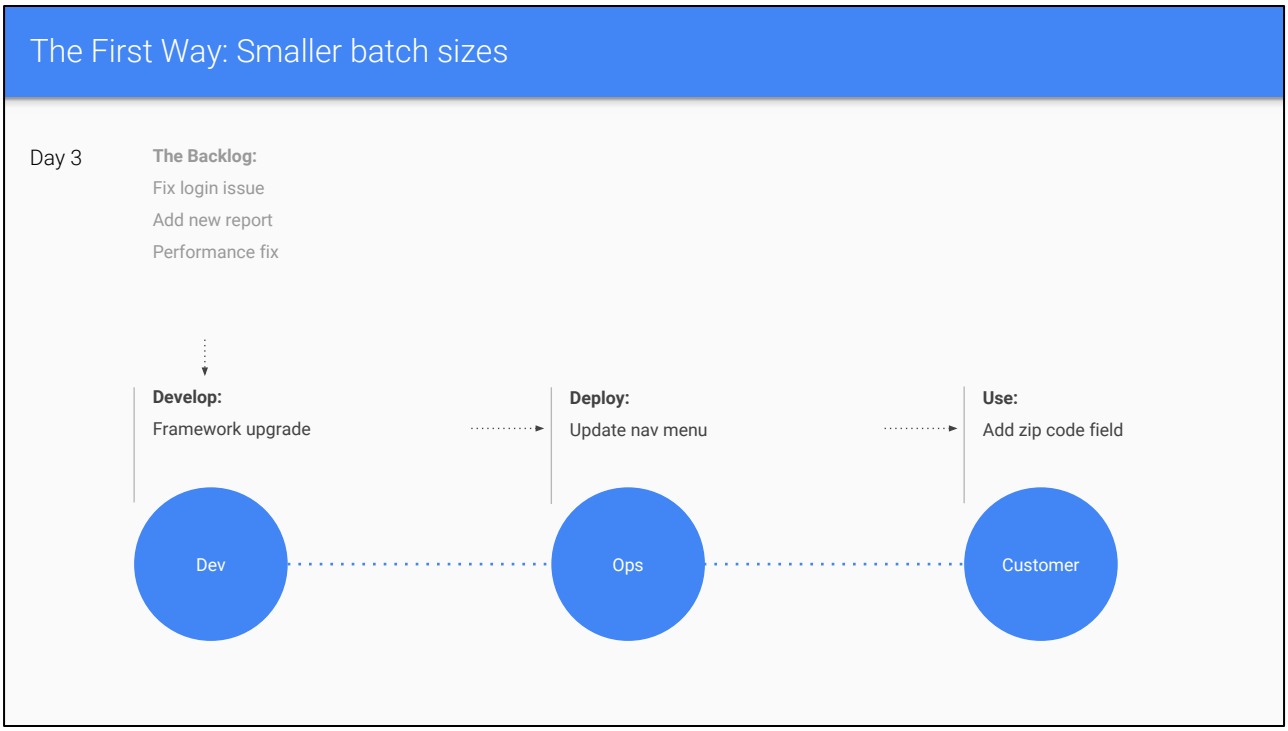
Now let's look at how these changes could be deployed in an organization that has successfully applied the First Way of DevOps, where they've optimized their system by reducing their deployment batch size to 1.

A batch size of 1 is certainly a viable batch size in DevOps when you have a highly automated deployment pipeline. This means a developer works on the zip code field, and when they're done, they check in the code and an automated continuous integration system generates a package that operations can deploy.

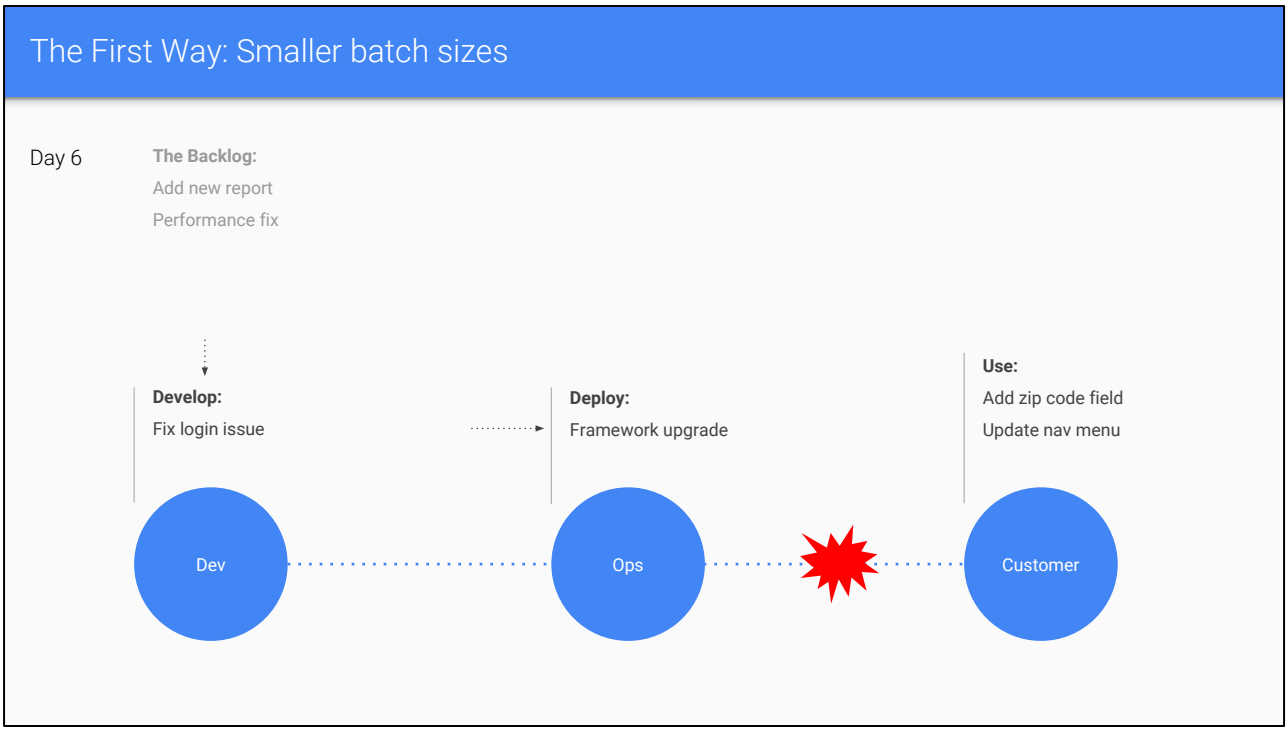
In this case, the continuous integration system has generated a deployable package containing the zip code field change.



Once the deployment package for the zip code change has been created, operations can deploy it. Like the dev team, operations has automated their part of the process, meaning the infrastructure is automatically deploying this package. Operations doesn't have to manually deploy it; they just have to keep an eye on the automated deployment to make sure everything runs smoothly. In the meantime, now that the dev team has completed their work on the zip code field, they can move on to the nav menu ticket. Also, freed up from having to perform manual deployments, operations can focus on improving the infrastructure.



Thanks to operations automating their deployment process, the zip code field quickly goes into the production environment where the customer can use it right away. At the same time, the development team has completed the nav menu change, and that package has been picked up by operations and is on its way to production too. This means the dev team is now focused on their next task, which is upgrading the app's framework.



At this point, it's been six days. That's one day to develop the zip code field change, one day to develop the nav menu change, and four days to develop the framework upgrade. Operations has successfully deployed the zip code field and the nav menu changes. The customer now has the ability to use these new features. They didn't have to wait for all six changes to go through first.

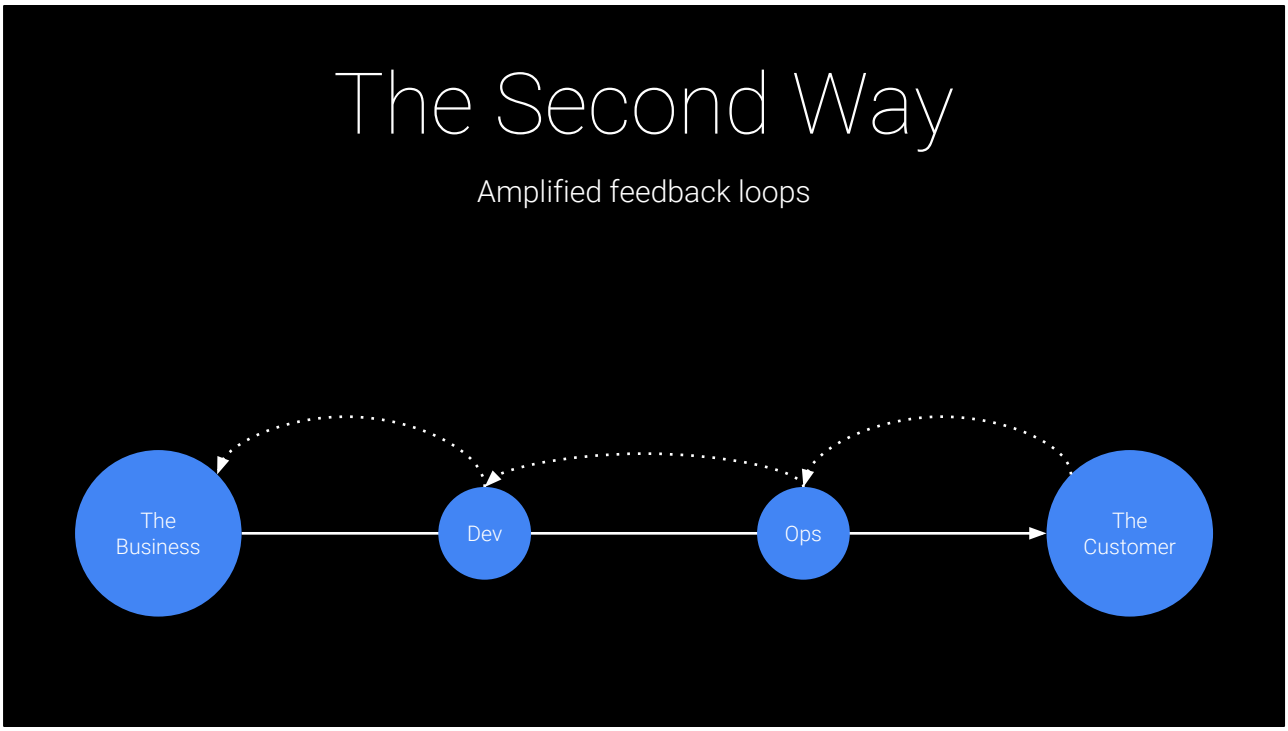
Now operations is automatically deploying the upgraded framework as usual. Again, let's say this framework upgrade has a problem that causes it to not work well in production for whatever reason. The tests in the automated deployment system rolls back the framework upgrade. But that's all right, the customer can keep on using the two features they already received. At the same time, the automated deployment system informs both operations and development that the framework upgrade has failed. Now the development team has to return the framework upgrade to their backlog and work on it. They can work on it now if it's urgent, or they can continue working on what they're doing right now, which is the login issue, and get that out to the customer.

This DevOps approach has three major differences in comparison with the traditional method:

First, all parties instantly knew which change caused the issue, and the issue was easy to deal with. Since the batch size of the framework upgrade was just one, rollback was fast, root cause analysis was easy, and the incident was short-lived. The framework upgrade can now go back into the developers' backlog.

Second, the impact on the customer was minimized because at least the customer got some features from previous deployments, and any production downtime was shorter due to the smaller deployment.

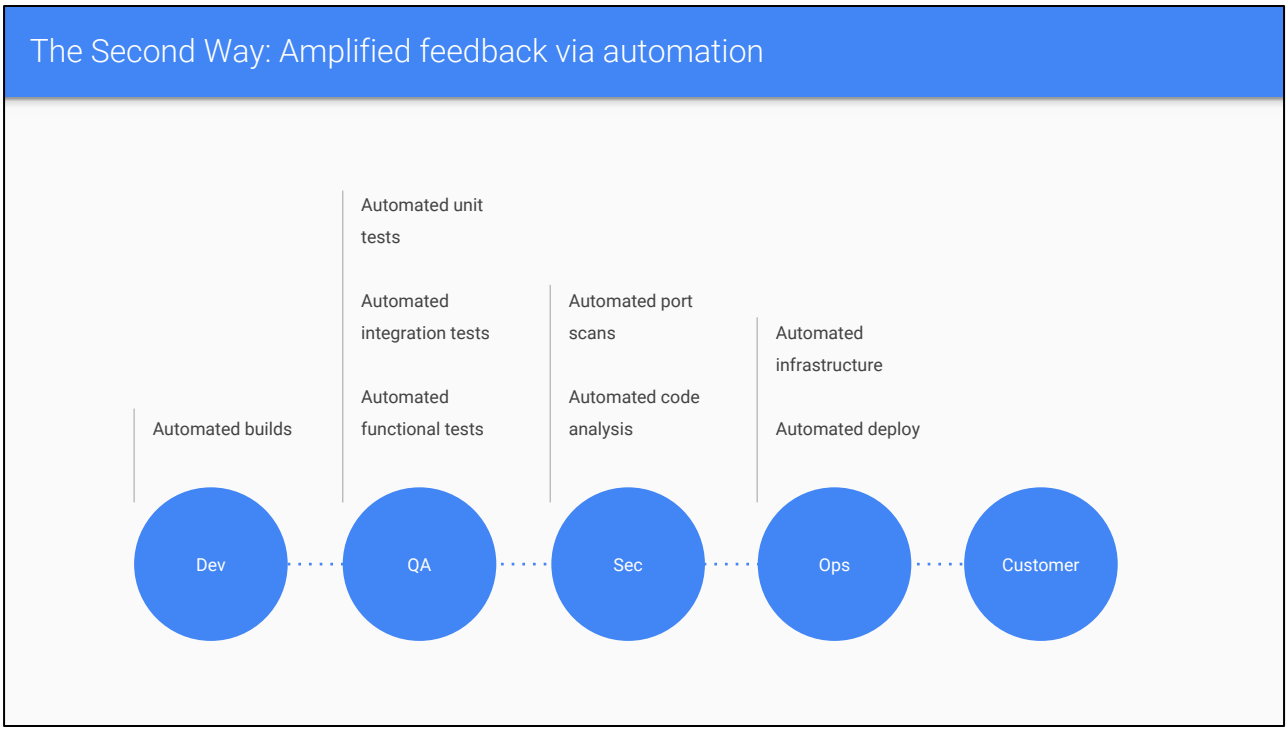
Third, the customer got the first two features as soon as the developers were done writing them. They were able to become productive from these new features more quickly. This is delivering value to the customer faster.



The Second Way of DevOps is amplified feedback loops. Simply put, this is when feedback flows quickly and efficiently from right to left, meaning from the customer all the way back to development and also to the business. This feedback needs to come constantly and quickly. Operations needs to understand what the customer is experiencing: are apps loading fast or slow? Are customers able to log in? Are any servers down? Developers need to know right away if a build is failing, or if customers are having difficulty with a redesigned feature.

Feedback can come in many forms. Customers may report issues via a help desk, via a web survey, or may even open an issue in the organization's open source repository. Analytics can provide information on how customers are actually using the app. Automated monitoring can reveal whether the infrastructure is healthy, and what parts of the code are causing a performance bottleneck. Audit logs can be scanned to reveal suspicious activity for the security team to take action on. Teams can make others aware of their status via a dashboard or opening up their issue tracking systems for viewing access. All these diverse sources of feedback need to be carefully considered, and the important metrics should be selected and acted upon.

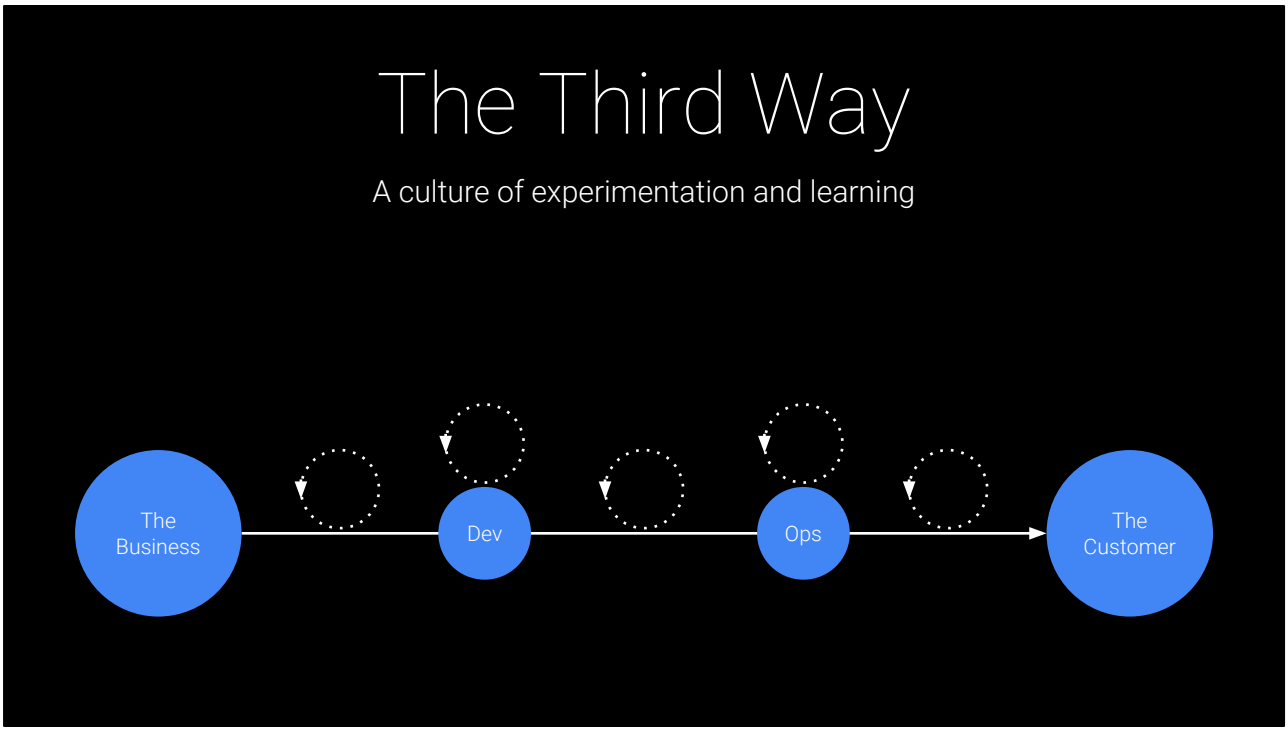
Not only is it important to constantly send and receive feedback, the IT organization must also learn from it. Useful information gleaned from feedback should be documented. Patterns should be recognized and documented.



So far, our diagrams have been rather simple, showing only the development and operations organizations. In reality, application development includes various other teams, the specifics of which vary between different IT organizations. Regardless of the name “DevOps” coming from development and operations, all of these teams have a place in DevOps. In DevOps, all these teams have the same roles, but their approach has to change. In the First Way, overall system efficiency is improved via automation. QA can ensure that any required tests are automatically run on each build, and test coverage meets their standards. Security can automatically run a static code analysis tool whenever a developer checks in new code. If an issue is discovered by any of these automated tools, the tool can halt the deployment and raise a flag, sending the work back to the dev team. By consistently running these automated checks along the way, IT can reduce risk and achieve faster delivery all at once. QA and security still hold the developers’ code to a high standard. They just take a different approach to implementing their work by thinking about how to make the overall system more efficient.

The automation implemented by the First Way helps us achieve amplified feedback. Consider all the automated tests performed by QA and security. If any one of these tests fails, it’s immediately reported to the dev team and the deployment of that particular batch is halted. The dev team can then add the issue to their backlog to be prioritized and worked on like any other ticket. The developers don’t have to wait for QA and security to manually write up test results. At the same time, QA and security can actively monitor the results and make suggestions to the developers, documenting these suggestions along the way. This builds a useful body of

knowledge.

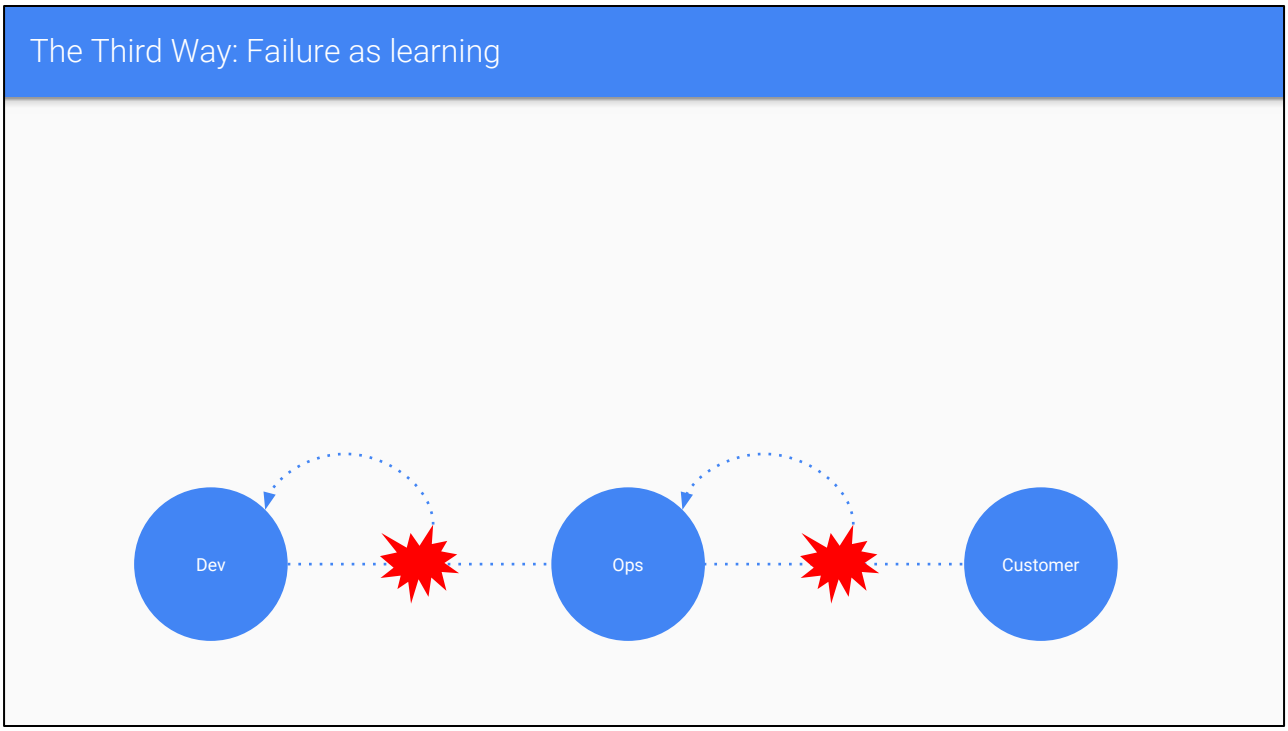


The Third Way of DevOps is creating a culture of experimentation and learning. There are two main ideas behind this practice.

The first idea is that the IT organization should be open to experimentation. They should build a culture where teams and individuals aren't afraid to take on a bit of risk and try something new. Experimentation leads to improvement. Everyone needs to understand that failure is acceptable as long as you learn something from it. We're accustomed to reducing risk in as many ways as possible, so how can an organization become accustomed to taking controlled risks? This is where the First Way and the Second Way will help: By reducing batch size and having fast feedback, you know that you can take small risks and you'll know the results very quickly. If you think about the batch size, automation, and amplified feedback from the First and Second Ways, it becomes clear that one can take small, calculated risks and recover very quickly if these experiments go wrong. In the era of commodity cloud servers that can be provisioned and de-provisioned cheaply and quickly, it's relatively easy to safely perform production deployments that can be scrapped and rolled back. If you recall, the First Way is about systems thinking, and about IT increasing the flow of value from the business to the customer. Constantly improving this flow is a key element of DevOps. An organization free to experiment will find ways to improve this flow.

The second idea is that repetition leads to mastery. You can find this repetition in doing exercises and drills, or you can find it by continuously taking small risks and recovering from them when they go wrong.

In order to succeed in the Third Way, the IT organization must have trust, transparency, tolerance for failure, and celebration of successes.



When the IT organization adopts the Third Way, failures are embraced as an opportunity to learn and practice. Some organizations even intentionally cause controlled failures to benefit from this opportunity.

Netflix came up with an unique idea: the Chaos Monkey. This is a tool that randomly causes mischief in their infrastructure. It might crash a server or it might cause network lag. The Chaos Monkey runs amok in their production environment. If you've watched anything on Netflix, the Chaos Monkey is somewhere in there taking down their servers. Netflix did this because they realized that failures are an excellent way to learn. Controlled failures keep everyone alert. The Chaos Monkey forces the developers to proactively make sure their code can gracefully recover from unexpected events, and forces their operations staff to make sure the infrastructure is robust.

Not every organization wants a Chaos Monkey running rampant in their cloud, but there are other ways to learn from failure. One is via blameless postmortems when there is a real failure. No matter how much risk is managed and no matter how carefully everyone treads, something will go wrong at some point. It could happen to anyone, so the best approach is to treat it as a learning opportunity rather than a time to assign blame.

In our scenario from the First Way, when our deployment failed, DevOps principles minimized the impact of that failure. A small batch size made it trivial to identify the cause of the failure. Automation led to an amplified feedback loop, informing teams of

the failure. Automation also made it easier for operations to recover in production. With these safety nets in place, the production environment is rugged and recovery can occur in seconds or minutes. Failure is no longer such a terrible thing. This grants organizations the leeway to experiment more via controlled risks. The ability to experiment leads to learning, growth, and optimization of the flow of value from the business to the customer. Because of the deployment failure, the development team learned something new about the framework they're using. The operations team learned something new about the infrastructure and why the upgrade didn't work in the production environment.

culture + practices + tools

=

faster delivery + better quality

We've covered a bit of the cultural change and practices that are involved in a shift to DevOps. If you can relate to the story about the zip code field at an agency far, far away, then I hope you agree that building a successful DevOps shop is a worthy endeavor. It is not easy because cultural change is hard, and there is no single product out there that gives you all the DevOps tools you need in a box. But it is absolutely doable and has tremendous benefits. The evidence is in the organizations that have implemented it.

This Tech Talk has barely scratched the surface on how to implement DevOps, but I hope it has piqued your interest in the subject. DevOps is a bottom-up approach, starting with individuals buying into the idea. There is a wealth of information out there, and I encourage you to continue pursuing more knowledge on the subject.

Thank you!

https://tech.gsa.gov/guides/

Jeff Fredrickson
jeffrey.fredrickson@gsa.gov

Get in touch with us at the Office of the CTO, and we can help you transform your IT shop.