FACILITATING THE SPREAD OF KNOWLEDGE AND INNOVATION IN ENTERPRISE SOFTWARE DEVELOPMENT

DevOps War Stories

by InfoQueue

Enterprise Software Development Community

eMag Issue 9 - January 2014

Introducing DevOps in a Traditional Enterprise

In this article we hear a very personal story on introducing a DevOps mindset at a large bank. In particular how the automation of configuration and release management processes enabled collaboration.

PAGE 3

NOKIA

DevOps @ Nokia Entertainment

DevOps@Nokia Entertainment is the first article of the "DevOps War Stories" series.

PAGE 12

PREZI

DevOps @ Prezi

Peter Neumark describes how Prezi transitioned from a centralized "devops team" to having its individual product teams have assume ownership of the infrastructure running their products or features.

PAGE 18

Spotify

DevOps @ Spotify

Mattias Jansson, Engineering Team Lead at Spotify, shows how devops has permeated the company's engineering management team.

PAGE 22

RAFTER

DevOps @ Rafter

Chris Williams explains how Rafter grew from one developer taking care of system administration tasks to a full-fledged DevOps team that supports the whole business.

PAGE 27

Contents

Introducing DevOps in a Traditional Enterprise Page 3

In this article we hear a very personal story on introducing a DevOps mindset at a large bank. In particular how the automation of configuration and release management processes enabled collaboration.

DevOps @ Nokia Entertainment Page 12

John Clapham, Software Development Manager at Nokia's Entertainment division, shows how DevOps behaviours can be introduced, and sustained, in a large organisation, through he five P's: promotion, planning, perseverance, patience, and of course, pizza.

DevOps @ Prezi Page 18

Peter Neumark describes how Prezi transitioned from a centralized "devops team" to having its individual product teams have assume ownership of the infrastructure running their products or features. Many of these teams did not include anyone with devops experience, so it became a priority for them to learn the basics.

DevOps @ Spotify Page 22

Mattias Jansson, Engineering Team Lead at Spotify, shows how devops has permeated the company's engineering management team. The result is a healthy overall team structure consisting of leam leads, product owners, and agile coaches.

DevOps @ Rafter Page 27

Chris Williams explains how Rafter grew from one developer taking care of system administration tasks to a full-fledged DevOps team that supports the whole business.

Introducing DevOps in a Traditional Enterprise

By Niek Bartholomeus

I have recently worked for a large investment bank as part of the architecture/development-tooling team where I spent a lot of time trying to solve the increasingly harmful problems of software by introducing DevOps.

My view on the subject of DevOps is highly influenced by the excellent presentation of John Allspaw and Paul Hammond of Flickr:

1. The business – whether that be the sponsor or the end-user – wants the software to change, to adapt to the changing world it represents, and it wants this to happen fast (most of the time as fast as possible).

2. At the same time, the business wants the existing IT services to remain stable or at least not disrupted from the introduction of changes.

The problem with the traditional software-delivery process (or the lack thereof) is that it is not well adapted to support these two requirements simultaneously. So companies have to choose between either delivering changes fast and ending up with a messy production environment or keeping a stable but outdated environment.

This doesn't work very well. Most of the time, they will still want both and therefore put pressure on the developers to deliver fast and on the ops guys to keep their infrastructure stable. It is no wonder that dev and ops will fight each other to protect their objectives and as a result they will gradually drift away from each other, leaving the head of IT stuck somewhere inside the gap that arises between the two departments.

This picture pretty much summarizes the position of the head of IT:

You can guess what will happen to the poor man when the business sends both horses in different directions.

The solution is to redefine the software-delivery process to enable it to support these two requirements – fast and stable – simultaneously.

But how exactly should we redefine it? Let us look at this question from the point of view of the three

layers that make up the software delivery process: the process itself; the tooling to support it; and the culture (i.e. the people who use it).

Process

The process should be logical and efficient. It's funny how big improvements sometimes can be made just by drawing the high-level process on a piece of paper and removing the obvious inconsistencies.

The process should initially take into account the complexities of the particular context (like the company's application landscape, its technologies, team structure, etc.) but at a later stage it should be possible to adapt the context in order to improve the process wherever the effort is worth it (e.g. switch from a difficult-to-automate development technology to a more easily automated one).

Tools

Especially if the emphasis is on delivering the changes fast, the process should be automated where possible. This has the added benefit that the produced data has a higher level of confidence (and therefore will more easily be used by whoever has an interest) and that the executed workflows are consistent with one another and not dependent on the mood or fallibility of a human being.

Automation should also be non-intrusive with regards to human intervention. Whenever a situation occurs that is not supported by the automation (e.g. because it is too new, too complex to automate, or happens only rarely), humans should be able to take over and return control to the automation when their job is done. This ability doesn't come for free but must explicitly be designed.

Culture

And then there is the cultural part: everyone involved in the process should have a high-level understanding of the end-to-end flow and in particular sufficient knowledge of their own box and all interfacing boxes.

It is well known that people resist change. It should therefore not come as a surprise that they will resist changes to the software-delivery process, a process that hugely impacts the way they work day-to-day. Now, back to reality. The solution that I implemented consisted of:

- The organization-wide alignment of the configuration and release-management processes.
- The automation of these processes in order to facilitate the work of the release-management team.

The biggest problems that existed in this traditional company were:

- The exponentially increasing need for release coordination, which was characterized by its manual nature, and the pressure it exerted on the deployment window.
- The inconsistent, vague, incomplete, and/or erroneous deployment instructions.
- The problems surrounding configuration management: being too vague and not reliable due to the manual process that tracked them.
- The manual nature of testing at the end of a long process meant that it must absorb all upstream planning issues, which eventually forced the removal of any change requests not signed off.

These problems were caused or at least permitted by the way the IT department was organized, more precisely:

- The heterogeneity of the application landscape, the development teams, and the ops teams.
- The strong focus on the integration of the business applications and the high degree of coupling between them.
- The manual nature of configuration management, release management, acceptance testing, server provisioning, and environment creation.
- The low frequency of the releases.

The first step we took in getting these problems solved was to bring configuration management under control.

Configuration management

It was obvious to me that we should start by bringing configuration management under control as it is the core of the whole ecosystem. A lot of building blocks were already present. It was only a matter of gluing them together and filling the remaining gaps

This first step consisted of three sub-steps:

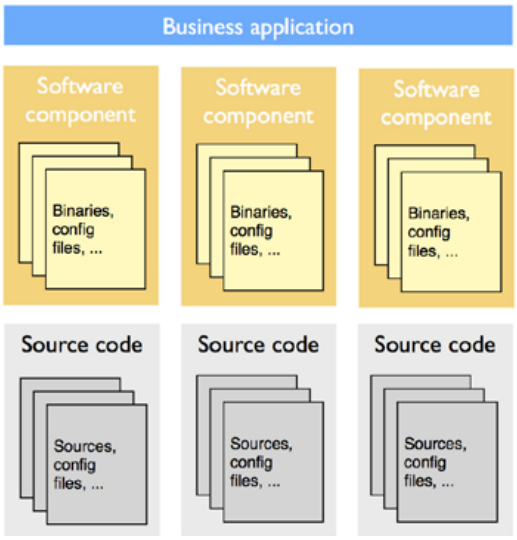
1. Getting a mutual agreement on the structure of configuration management.
2. The implementation of a configuration-management system (CMS) and a software repository - or definitive media library (DML) in ITIL

terms.

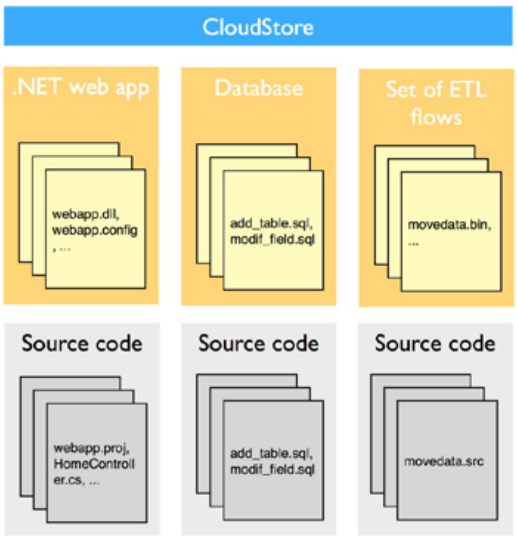
3. The integration of the existing change-management tool, build/continuous integration tools, and deployment scripts.

We decided to create three levels of configuration items (CI). On top was the level of the **business application**. Business applications can be seen as the "units of service" that the IT department provides to the business. Below that was the level of the **software component**. A software component refers to a deployable software package like a third-party application, a web application, a logical database, or a set of related ETL flows, and consists of all files that are needed to deploy the component. And on the bottom level we placed the **source code** that consists of the files needed to build the component.

Here is an overview of the three CI levels:



And here is an overview of the three levels of configuration management for a sample application called CloudStore:



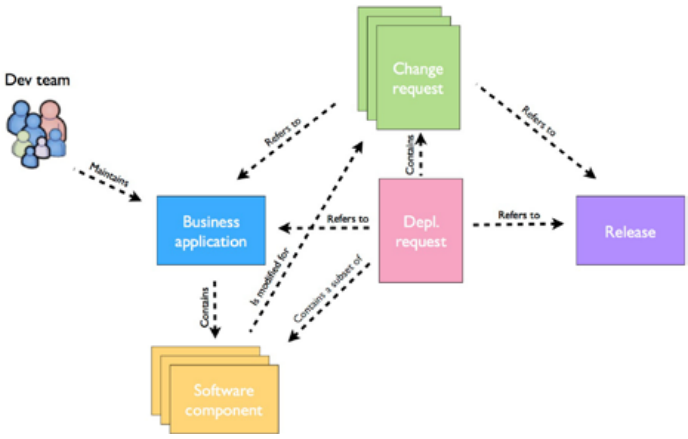
CIs are not static. They must change over time as part of the implementation of new features and these changes must be tracked by versioning the CIs. You could use a version number to uniquely identify a particular set of features that are contained within the CI.

Here is an example of how the application and its components are versioned following the implementation of change requests:

CloudStore	**2.0.0**	**2.1.0**	**2.1.2**
.NET web app	3.1.3	3.2.0	3.2.1
Database	1.7.2	2.1.2	2.1.2
Set of ETL flows	2.5.3	3.1.0	3.1.0
Change req	**CS-123**	**CS-124**	**CS-125**

Once we defined this structure, we created a complete list of business applications and statically linked each application to the components that made it up and to the development team that maintained it. Change requests that were created for a particular application could no longer be used to change components that belonged to a different application. And deployment requests created for a particular application could no longer be used to deploy components of a different application. An entirely new change or deployment request had to be created for this second application.

The resulting conceptual model looked as follows:



Following the arrows: an application contains one or more software components. There can be multiple change requests per application and release, but only one deployment request and it is used to deploy one, some, or all of the software components.

Source code is stored and managed by version-control systems. This level is well documented and well supported by tooling (git, svn, TFS, etc.) so I will not further discuss it here.

The files that represent the components (executables, libraries, config files, scripts, setup packages, etc.) are typically created from their associated source code by build tools and are physically stored in the DML. All relevant information about the component (the application it belongs to, the context in which it was developed and built, etc.) is stored in the CMS.

The business application has no physical presence. It only exists as a piece of information in the CMS and is linked to the components that make it up.

Here is an overview of the implementation view:

CI	Action	Tool
Application	Managed	CMS
Component	Managed	CMS
	Stored	DML
Source code	Managed	Version control tool
	Stored	Version control tool

Once we agreed upon these rules, a lightweight tool was built to serve as both the CMS and DML. It was integrated with the build tools in such a way that it automatically received and stored the built files after each successful build of a particular component. Restricting the upload of files exclusively to the build tools (which assures that the component has successfully passed the unit tests and the deployment test to a development server) assures at least a minimum level of quality. Additionally, once a particular version number of a component was uploaded, it was frozen. Attempts to upload a newer version of the components with a pre-existing version number would fail.

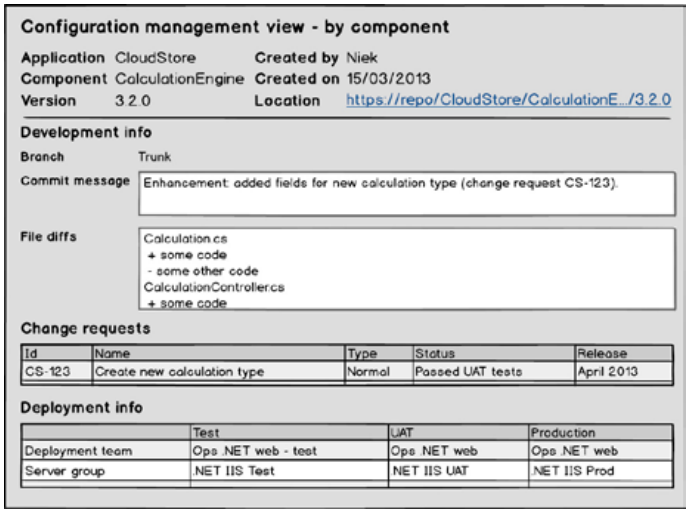
The build tools not only sent the physical files, but also all the information relevant to the downstream processes: the person who built it; when it was built; the commit messages (a.k.a. check-in comments) since the previous version; the file diffs since the previous version; etc.

With this information, the CMS was able to calculate some interesting pieces of information that used to be hard to manually keep track of before, namely the team responsible for the deployment and the logical server group (e.g. “Java web DEV” or “.NET Citrix UAT”) to deploy to. Both were functions of the technology, the environment, and sometimes other parameters that were part of the received information. As these calculation rules were quite volatile, they were implemented in some simple scripts that could be modified on the fly by the administrator of the CMS whenever the rules changed.

The CMS also parsed the change-request identifiers from the commit messages and retrieved the relevant details about it from the change-management tool (another integration we have implemented) like the summary, type, status, and the release for which it was planned.

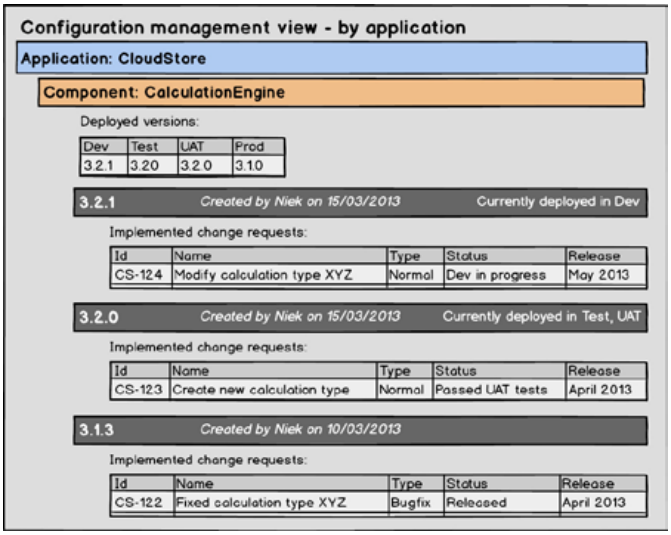
The presence of the core data that came from the build tools combined with the calculated data and especially the data retrieved from the change-management tool transformed the initially boring CMS to a small intelligence center. It became possible to see the components (and their versions) that implemented a particular change request or even those that implemented any change requests for a particular release (assuming, of course, that the commit messages correctly contain the identifiers of the change request.)

In the following mockup, you can see how the core data of a component is extended with information from change management and operations:



It’s also interesting to look at configuration management from a higher level, from the level of the

application for example:



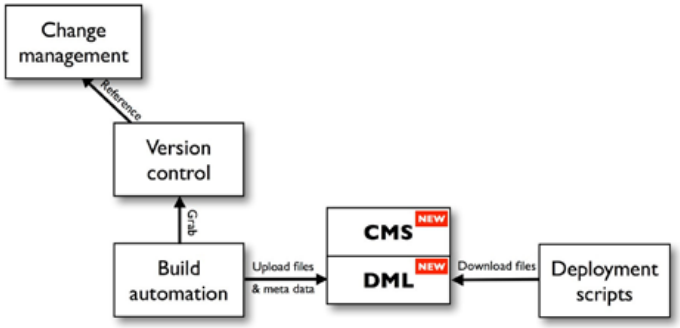
By default, the view shows all the components of the selected application and all versions of each component since the currently deployed version in descending order. In addition, the implemented change requests are also mentioned. (Note that the mockup also shows which versions of the component were deployed in which environment. I'll talk more on this later when discussing the implementation of a release-management tool.)

The CMS also contained logic to detect a couple of basic inconsistencies:

- A particular version of a component didn't implement any change requests. (In that case it was still possible to manually associate a change request with the version.)
- A component version implemented change requests that were planned for different releases. (It would be quite difficult to decide when to deploy it, right?)
- A change request that was planned for a particular release was not implemented by any components.

All of this information facilitated the work of the developer in determining the components he had to include in his deployment request for the next release. But it was also interesting during impact analysis (e.g. after a change request was removed from a release) to find out more information about an individual component or about the dependencies between components.
This ability to do impact analysis when a change request had to be removed from a release was a big deal for the company. One of the dreams of the people involved in this difficult task was the ability to get a complete list of all change requests that depend on a single change request. Although this capability was not developed initially, it would not be very difficult to do now that all the necessary information was available in the CMS. The same could be said about including more intelligent consistency checks: it's quite a small development effort for sometimes important insights that could save a lot of time.
Finally, the deployment scripts of all technologies were adapted in such a way that they always retrieved the deployable files from the DML and as such finalized the last step of a fully controlled software-delivery pipeline from version-control tool to production.

Here is an overview of how the CMS and DML were integrated with the existing tools:



You may have noticed that I have emphasized the importance of keeping track of the dependencies between the change requests and the components to enable proper impact analysis when the need arises. These dependencies are the cause of many problems within the software-delivery process and therefore a lot of effort has to be put into finding ways to remove or at least limit their negative impact. The solution I mentioned here was all about visualizing the dependencies, which is only the first step of the solution.

A much better strategy would be to avoid these dependencies in the first place. And the best way to do this is by simply decreasing the size of the releases, which comes down to increasing the frequency of the releases. When releases happen infrequently, the change requests typically stack up and in the end all change requests are dependent on one another due to the technical dependencies that exist through their components. You remove one and the whole thing collapses. But increasing the release frequency

requires decent automation and this is exactly what we're working on. Until the whole flow is automated and the release frequency can be increased, we have to live with this problem.

If we take this strategy of increasing the release frequency to its extremes, we end up with continuous delivery, where each commit to each component triggers a new mini-release, one that contains a one-step deployment of the component that was committed. No more dependencies, no more impact analysis, no more problems! Nice and easy! Nice? Yes! Easy? Maybe not, because this approach doesn't come for free.

With batch-style releases, you can assume that whenever the new version of your component is deployed into an environment, it will find the new versions of all components it depends on (remember that most of our features required changes to multiple components). It doesn't have to take into account that old versions of these components may still hang around. With continuous delivery, this assumption is not guaranteed anymore in my opinion. It's now up to the developer to make sure that his component supports both the old and the new functionality and that he includes a feature flag to activate the feature only when all components are released. In some organizations (and I'm thinking about the large, traditional ones with lots of integrated applications), this may be a high price to pay.

It's time to move on to the second step of the solution: bringing release management under control.

A need for orchestration

As already mentioned, the applications in our company were highly integrated with each other and therefore most of the new features required modifications in multiple applications, each managed by its own dedicated development team.

This seemingly innocent fact had a rather big consequence: it resulted in the need for a release process capable of deploying the modified components of each of these impacted applications in the same go, in other words an orchestrated release process. If the component would be deployed on different days, it would break or at least disrupt the applications. In fact, at least some degree of

orchestration is necessary even if a feature can be implemented by only modifying components within one application (a good example is a modification in the source code and one in the database schema). But as this would stay within the boundaries of one development team, it would be relatively easy to organize the simultaneous deployment of all components.

Things become way more complicated when more than one application and development team is involved in implementing a feature and moreover when multiple application-overlapping features are being worked on in parallel and thereby modifying the same components. Such a situation really shouts for a company-wide release calendar that defines the exact dates when components can be deployed in each environment. If we deploy all components at the same moment, we are sure that all dependencies between them are taken into account.

Unfortunately, we all know that creating strict deadlines in a context so complex and so difficult to estimate as software development can cause a lot of stress - and cause a lot of missed deadlines as well, resulting in half-finished features that must somehow be taken out of the ongoing release and postponed to the next release. A lot of the dependent features must be taken out as well. Features that stay in but were rushed may lack the quality that is expected from them, causing system outages, urgent bug fixes that must be released short after the initial release, loss of reputation, and so on downstream in the process. And even if all goes according to the plan, there is generally a lot of unfinished work waiting somewhere in the pipeline for one or another deployment (development-done but not release-done, also called WIP or work in progress).

Continuous deployment avoids all these problems caused by these dependencies by requiring the

developers to support backward-compatibility; in other words, their component should work with the original versions of the other components as well as with the new versions that include the new feature, simply because they don't know in advance in which order the components will be deployed. In such a context, the developers can work at their own pace and deliver their components whenever they are ready. As soon as all the new versions of the components are released, a feature flag can be switched on to activate the feature.

There is an unavoidable, huge drawback to this scenario: in order to support this backward-compatibility, the developers basically have to keep two versions of the logic in his code, and this applies to each feature that is being worked on. This requirement can be a big deal, especially if the code is not well organized. Once the feature is activated, they need to remember to remove the old version to avoid a code base that becomes a total mess after a while. If there are changes to the database scheme (or other stateful resources) and/or data migrations involved, things will become even more complicated. Continuous deployment is also tightly coupled to hot deployment, which introduces challenges of its own, and if a company doesn't have a business need to be up all the time, that's a bit of a wasted effort. I found a nice explanation of all the intricacies of such practices in this webinar by Paul Biggar at MountainWest RubyConf 2013.

Don't get me wrong. Continuous deployment is great and I really see it as the way to go but that doesn't take away that it will take a long time for the "conventional" development community to switch to a mindset that is so extremely different from what everyone has been used to for so long. For this community (the 99%-ers ;-)), continuous deployment appears merely as a small dot on the horizon, practiced by whiz kids and Einsteins living on a different planet. Hopefully, if we can gradually bring our conventional software-delivery process under control, automating where possible and gradually increasing the release frequency, one day the leap towards continuous deployment may be less daunting than it is today and instead just be the next incremental step in the process.

Until then, I'm afraid we are stuck with our orchestrated release process so we better make sure that we bring the processes and dataflows under control to keep all the aforementioned problems it brings with it to a minimum.

Let us have a closer look at the orchestrated release process in our company, starting from the moment the components are delivered to the software repository (typically done by a continuous-integration tool) to the moment they are released to production. The first step was for the dev team to create a deployment request for their application. This request should contain all the components - including the correct version numbers - that implement the features planned for the ongoing release.

Each team would then send their deployment request to the release coordinator (a role at the enterprise level) for the deployment of their application in the first "orchestrated" environment, in our case the UAT environment. Note that we also had an integration environment between development and UAT where cross-application testing, amongst other types of testing, happened but this environment was still in the hands of the development teams in terms of deciding when to install their components.

The release coordinator would review the contents of the deployment requests and verify that the associated features were signed-off in the previous testing stage, e.g. system testing. Finally, he would assemble all deployment requests into a global release plan and include some global pre-steps (like taking a backup of all databases and bringing all services down) and post-steps (like restarting all services and adding an end-to-end smoke-test task). On the planned day of the UAT deployment, he would coordinate deployments of the requests with the assigned ops teams and inform the dev teams as soon as the UAT environment was back up and running.

When a bug was found during the UAT testing, the developer would fix it and send an updated deployment request in which the fixed components got an updated version number and were highlighted for redeployment.

The deployment request for production would combine the initial deployment request and all "bug fix" deployment requests, each time keeping the

last deployed version of a component, unless the component was stateful like a database, in which case deployments are typically incremental and as a result all correction deployments that happened in UAT must be replayed in production. Stateless components like the ones built from source code are typically deployed by completely overwriting the previous version.

Again, the release coordinator would review and coordinate the deployment requests for production, similar to how it was done for UAT and that would finally deliver the features into production after a long, stressful period.

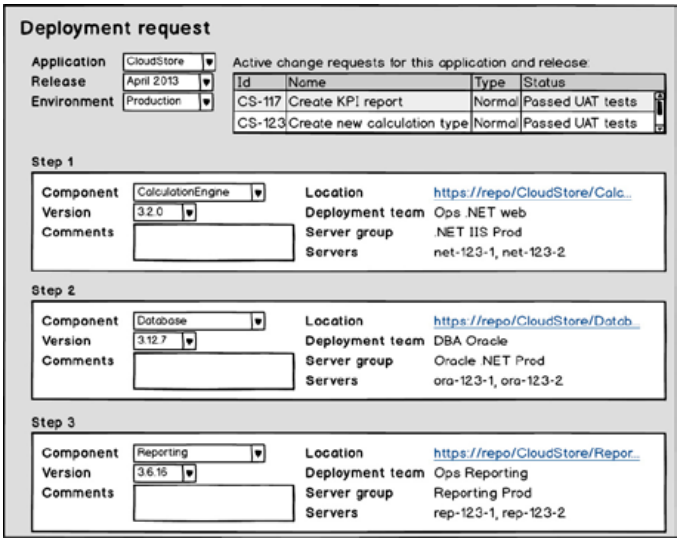
Note that I only touched the positive path here. The complete process was more complex and included scenarios like what to do in case a deployment failed, how to treat rollbacks, how to support hotfix releases that happen while a regular release is ongoing, etc. For more information on this topic you should definitely check out Eric Minick's webinar on introducing uRelease in which he does a great job explaining some of the common release-orchestration patterns that exist in traditional enterprises.

As long as there were relatively few dev and ops teams and all were mostly co-located, this process could still be managed by a combination of Excel, Word, e-mail, and a lot of plain human communication. However, as the IT department grew and spread out over different locations, this artisanal approach hit its limits and a more industrial solution was needed. On the tooling side, we chose a proprietary tool that helped us industrialize our release-management process.

First of all, it allowed the release manager (typically a more senior role than the release coordinator) to configure her releases, and this includes specifying the applicable deployment dates for each environment. It also allowed the developers to create a deployment request for a particular application and release with a set of deployment steps, one for each component that must be deployed. These steps can be configured to run sequentially or in parallel, and manually (as we had to, thanks the security team's refusing us access to the production servers) or automated.

On the deployment date, the deployment requests for a particular release can be grouped into a release plan - typically after all deployment requests are received - that allows the creation of release-specific handling like adding pre and post-deployment steps.

On the day of the deployment, the release plan is executed, going over each deployment step of each deployment request either sequentially or in parallel. For each manual step, the ops team responsible for the deployment of the associated component receives a notification and is able to indicate success or failure. For each automated step, an associated script takes care of the deployment.
Here's a mockup of a deployment request:

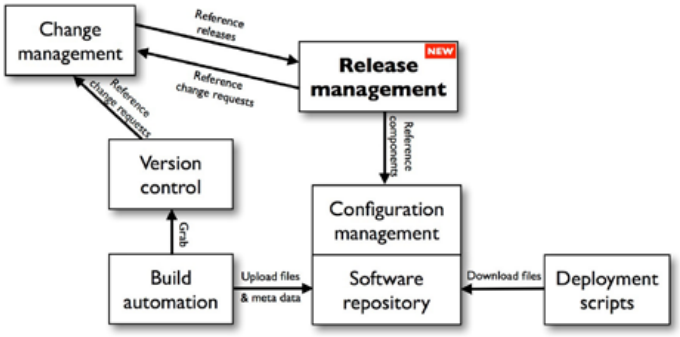


As part of the implementation project in our company, the tool was integrated with two existing tools: the change-management tool and the software repository.

The change-management tool notified the release-management tool whenever a change request (or feature) was updated. This allowed the tool to show the change requests that applied directly to that particular application and release on the deployment request, which made it possible for the release coordinator to easily track the status of change requests and for example reject the deployment requests that contain not-yet-signed-off change requests.

At the same time, the software repository notified release-management tool whenever a new version of a component was built, which allowed the tool to restrict the choices of the components and their version numbers to those that actually exist in the

software repository.
Here's an overview of the integration of the release-management tool with the other tools:



More generally, by implementing a tool for release management rather than relying on manual efforts it became possible to increase the quality of the information by either enforcing correct input data or by validating it a posteriori through reporting - and to provide better visibility on the progress of the releases.

Conclusion

Let us take a step back and see which problems identified initially were solved with implementation of a configuration-management tool and a release-management tool:

- The exponentially increasing need for release coordination, which was characterized by its manual nature, and the pressure it exerted on the deployment window. => SOLVED
- The inconsistent, vague, incomplete and/or erroneous deployment instructions. => SOLVED
- The problems surrounding configuration management: not being enough under control and allowing a too-permissive structure. => SOLVED
- The manual nature of testing at the end of a long process meant that it must absorb all upstream planning issues, which eventually forced the removal of any change requests not signed off. => NOT SOLVED YET
- The desire by the developers to sneak late features into the release, thereby bypassing the validations. => SOLVED

Looks good doesn't it? Of course, this doesn't mean that what was implemented doesn't need further improvement. But for a first step, it solved a number of urgent and important problems. It is time now for these tools to settle down and to put them under the scrutiny of continuous improvement before heading to the next level.

About the Author

Niek Bartholomeus has been a devops evangelist in a large financial institution for the last five years where he was responsible for bringing together the dev and ops teams, on a cultural as well as a tooling level. He has a background as a software architect and developer and is fascinated by finding the big picture out of the smaller pieces.

READ THIS ARTICLE ONLINE ON InfoQ
http://www.infoq.com/articles/devops-04-large-bank

DevOps @ Nokia Entertainment

By John Clapham

Introduction

I'd like to tell a story about DevOps. I'll be drawing on some of the experiences and lessons learnt in a small corner of Nokia, Nokia Entertainment Bristol in England. We are creators of fine music products and during the last couple of years our outfit has learnt to deliver server-side software, fast. In order to learn, we've had to make mistakes. I'll aim to share our successes and failures. It's not a recipe for DevOps, but you might find some useful ingredients.

There are numerous definitions of DevOps and nearly every variation I read holds something new and interesting. For the purposes of this article, I'd suggest DevOps is a way of working where:

- Agile project management that adapts with you. Provide your global development team a shared space that adapts to the way they work.
- Developers and operators work together to ensure products get built and the systems they create scale and stay stable. They understand each other's responsibilities and routinely draw on each other's expertise.

This can be contrasted with the situation where the two teams are separated, organisationally or culturally, and don't work effectively together. DevOps is both a desirable culture and a solution to a specific problem. It's not needed everywhere. When I talked to a few of the start-up people at the Future of Web Apps conference about DevOps, most of them didn't see the need. When there are just a few people in a company, responsibility is shared and there isn't room for the kind of specialism and protectionism that can prove so divisive as organisations scale up.

This article also mentions Continuous Delivery - quite a lot. That's because chasing the continuous-delivery vision of faster, more frequent changes led us to recognise the need for DevOps.

Why things had to change…

A few years ago, creating software releases really hurt. They hurt in the kind of creative ways that would a make a James Bond villain proud. Although teams were developing agile methods, changes were queued up and applied to production systems in one drop, or rather carefully built into a fragile house of cards and carefully coaxed to production. This led to a number of common problems, with all too familiar symptoms.

Quality – The rush for release deadlines could easily compromise quality of both decisions and software. The knowledge that not including a feature in this release would mean waiting another couple of months made for some tough decisions, and late nights, to get the last pieces in place.

Risk – Releases weren't routine. No two were the same; each was unique and involved complex dependencies. This meant they risked downtime, performance issues, finger trouble, and similar

problems. Some of this was mitigated with testing, rehearsals, and rollback plans, but the sheer number of changes and moving parts meant there was always scope for problems.

Motivation – Despite generous amounts of late-night pizza, there was a people side effect. Pride and satisfaction with the release contents were beaten down by bureaucracy and the difficulty of the release process. Pressure was high during releases. One mistake and the house of cards comes crashing down, and with it the chance of going home in daylight. People committed to release activities often weren't doing the roles they went to college or studied for. Working on releases was an unwelcome distraction from the areas that really added value to the business.

Cost – There were both opportunity and financial costs. While effort is going into building this giant release-shaped house of cards, it's not invested in innovation or adding value to existing products. Infrequent releases also made for long lead times and a feeling of unresponsiveness. In an environment where opportunities for change are scarce, completion for precious release slots can easily lead to escalation and the high cost of senior-management involvement.

Despite a seemingly bleak picture, we were still delivering features but knew we could do better. People were acutely aware of the symptoms but perhaps not their origins. Attempts were made to improve matters, both in terms of refining the existing process and steps towards alternative approaches, including continuous delivery and DevOps. However, the real killer was introducing change, or even creating a plan, whilst meeting our other commitments.

How we got there…

Taking stock

Instead of diving into the next release activity, we stopped. Even this was hard, and took discipline. The temptation is always to keep pressing forward, like the infamous man who won't stop to sharpen his saw. Expending energy feels good, even if it's ultimately wasteful. We stopped just long enough to consolidate our vision and plan our next steps. After all the cajoling and evangelism, the business started to listen. Particularly notable were discussions at the leadership level between operators, engineering, product, and architects. Out of this came agreement to try something new, at both organisational and technical levels.

The something new was continuous delivery with its promise of reduced lead times and costs, and an end to all that release hurt. Implementation of continuous delivery would require agile engineering teams to consider release into production, not handover to quality assurance, integrators, or ops, as the point at which their work was done. In addition, frequent deployments to live would be required, something for which neither tooling nor people were set up. It was clear that greater collaboration would be required to achieve this, and that there was a lot to learn from the growing DevOps movement. I think it's fair to say that at time we didn't appreciate all the many benefits that way of working brings.

Commitment

A team was created to help build tools and encourage progress towards continuous delivery and a DevOps style. This built on the work of individuals and enthusiasts who had been scattered across teams, enabling them to concentrate their initiatives, both technical and otherwise. It is perhaps an obvious move but it's not an easy one to sell. It requires recognition that teams need to improve capability and build product, and an acceptance that delivery may slow while new ways of working are established. In addition to the time investment, a dedicated team or project sends a strong signal: this is the approach for the future and we're committed to it. Both help people get behind the ideas and can save a lot of wasteful should we/shouldn't we discussions.

Inspect, adapt, learn

Being an agile shop, the natural place for us to start was retrospection. This is the process of looking at what is working and what isn't, and generating ideas to enhance the positive factors and suppress the negative. In engineering teams, retrospectives were routine, but continuous delivery required input and perspectives from many other areas including architects, product, quality assurance, and testers. Cross-team retrospectives generated some of the most useful insights; they encouraged conversation, a sense of shared purpose, and community. Reflecting,

I think these should have been run more regularly. When radical change is taking place, the pace of change can be startling, but it's not always positive change and therefore not always welcomed, especially if the long-term benefit is masked by short-term inconvenience.

Focus on value

Another useful concept was thinking about and occasionally obsessing over the value processes, roles, and tools to bring to the release process. This led us to challenge the interfaces between dev and ops, and also the way we assessed and managed risk. It's easy for habits and practices to form, and over time the bad and indifferent may become accepted and unchallenged. One straightforward technique is simply challenging process steps and asking, "What benefit does that actually add to us?" We also employed a more rigorous technique named value-stream mapping to visualise and understand the entire product-development process. Given the opportunity to try this again, rather than spending time and discussion trying get a single "true" map, I'd encourage separate groups to produce their own value-steam maps, and then compare the differences. This should expose differences in interpretation and quickly reveal those "Oh, we only did that because we thought your team needed that" issues.

What we did – what worked and what didn't

Just enough tools

One early initiative aimed to improve the release process by reducing errors and speeding up handling of configuration information. Handover was needed because different teams owned different environments on the path to production. Tools became complex, trying to serve both providers and consumers of data. A kind of arms race developed, with requirements added to combat specific failures. As they grew unwieldy, some of the tools felt more like a hindrance than a help. These tools tended to manage the interface between teams but this just reinforced differences and, even worse, discouraged conversation and collaboration.

To assist with releases, we introduced new tools, processes, and automation. We wanted to put responsibility for application deployment, previously the preserve of operators, into the hands of anyone competent, especially engineers. Incorporating the requirements of ops and devs, we developed a deployment tool that enforced the release pipeline, managed configuration, audited, and orchestrated deployments to different environments. It did just enough, and we relied on healthy collaboration and diligence to do the rest.

The approach to dependency management was a good example. The service-oriented architecture contained complex dependencies but software seemed an expensive way to handle the problem. We saved effort with a few simple rules. As usual, engineers took responsibility for automated testing and keeping services compatible with their consumers. Then we added the notion of a "deployment transaction", an exclusive lock across the integration and live environments. This meant only one service could be tested and deployed at a time, reducing the risk of deployment-order problems. A useful side effect was that the approach encouraged conversation between teams queuing for the transaction, promoting an understanding of the different moving parts of the system

Organisation and leadership

In the classic setup, engineering and operations were separate organisations. There was an impedance mismatch between the two organisations: their own teams; ways of working; leadership; and styles. Crucially, incentives differed. To summarise gratuitously, ops were incentivised for stability and devs for delivering change. It was evident that we needed to create a structure which encouraged more collaboration and promoted shared goals. Although we'd love to think that most problems can be solved at a technical level, in a large organisation the rate of change is often accelerated by evangelists, strong leadership, and a critical mass of people setting a good example. A forward-thinking manager stepped up to lead both operations and development teams, straddling the divide. This helped to change behaviours and, crucially, let him see and understand both perspectives, a priceless source of feedback. An org chart is just a diagram, though, and although it can signal a clear intention, it doesn't necessarily mean anything will change or

stay changed, and that's why culture is so important.

Culture

The culture in engineering was based on agile, using a mix of scrum, kanban, and lean concepts. This meant we were already some way towards the DevOps culture we wanted to create. There was already some collaboration and overlapping knowledge, but there were specific areas to enhance and be wary of. Relaxing some of the rules and letting people work out of their normal areas of expertise could lead to failures, and tempt a demoralising and energy-sapping blame culture. Key cultural principles included trust, acceptance that things would break occasionally, shared learning, and responsibility in the right place.

Adoption of the deployment tool was one of the first things to test our culture. Hitherto, ops made all the production changes; they had intimate knowledge of the systems so why on earth should they trust engineers to make unsupervised deployments? To learn more, we used a safe-fail experiment. We set up a deployment workstation in the heart of the ops space. It was the only place deployments could be made, and deployments could only be made with both an operator and a developer present. Over time, experience was gained, bugs were fixed, and the tool was refined. Trust was gained and the tool became ubiquitous.

During the times when developers and ops people weren't talking frequently, there was a kind of reactive collaboration, a tendency to talk only when things went wrong. This meant poor forward planning and encouraged a defensive stance from both teams. We aimed to encourage early conversations as collaboration is always easier if you share the same vocabulary, so we ran cross-team training courses and workshops.

It is interesting to note that the next hire is a vital opportunity not just to augment the skills of the team but to steer culture in a desired direction. We considered what skills and principles would further a DevOps culture. In an interview, it's informative to ask an ops type how he feels about software developers working directly on production systems, or an engineer how he would feel if his code went live 30 minutes after commit. Blood draining from the face and gripping the arms of the chair are not promising signs.

Where we are now

The DevOps mentality was one of the foundations for improving our release capability, but the benefits of the approach are often hidden under the continuous-delivery banner. Let's walk through the areas that were so troublesome when releases were infrequent:

Quality - In general, the quality of services has increased. The role DevOps plays is most visible when there are live incidents such as bugs, outages, or increasing response times. This shows in both how production issues are noticed and what happens when they are. Production-system monitors could be roughly categorised into two areas: infrastructure and application. Infrastructure monitors (Nagios, Keynote) are created with the needs of ops in mind; application monitors (Graphite) are built more for devs' requirements. The overlap or redundancy between monitors is desirable: two perspectives of the same systems act like a parity check. When an incident does occur, collaboration is via Campfire, directly between the people that are needed and who can add value. If you've ever been the person sweating over the console on a production server with a project manager over one shoulder and your boss on the other, you'll understand how refreshing this is.

Risk - The risk, that is the likelihood of problems arising as result of release activities, has decreased. Smaller incremental changes reduce the risk per change, and crucially make it easier to assess risk. In a healthy DevOps relationship, people either have enough knowledge to make an assessment themselves or know when to draw in expertise from another team.

Motivation - I believe people's motivation has improved. There's still plenty of frustration, but knowing that deployment times are short and that ops and engineers will support each other makes a big difference. With responsibility in the right place, people are performing the roles they signed up for and more time is spent creating, rather than delivering, product. For ops, this means less

time fighting fires and more time investing in our platform, particularly the automation side.

Cost - The time cost of delivering a change is where DevOps ways of working have really had a huge impact. Shared deployment tooling eliminates handover time. Collaboration, planning ahead, and early testing all mean that applications are almost certain to work on production infrastructure rather than hold nasty surprises. Costly escalations and investigations by management are reduced as a result of greater understanding and trust between the two areas.

Who gets the pager?

In case you're wondering about the classic "Who gets the pager?" – it's ops. They have the skills and experience to deal with live issues, not to mention ways of working geared towards call-out and fast reactions. There is something different though – involvement with R&D engineers is much quicker, and an accepted way of working.

The challenges we face now…

We now face two classic challenges that follow change: keeping it going and getting better. Sustaining a fledgling DevOps culture should be realistic. Engineers and ops are getting things done by working and learning together and the business sees clear advantages and is supportive. Considering Mitchell Hashimoto's range of DevOps, we're veering somewhat to the right, so while developers are enthusiastic to take on new challenges, we should take care that the operator's perspective is represented and acted upon.

John Wills makes a sound point that DevOps is about culture, automation, measurement and sharing. Automation and measurement could be our next improvements. Both areas need to keep pace with change in our technology and be developed alongside features rather than afterwards. The technical side of DevOps will never be done. We frequently experiment with new languages and need new approaches to provisioning, like Clojure in conjunction with Pallet.

There is also the challenge of taking what we've learnt while bringing two quite disparate teams together. Huge benefits have been gained by focusing on collaboration and understanding others' responsibilities, perspectives, and priorities. The behavioural patterns that led to our interest in DevOps lurk elsewhere in the org and similar benefits could arise by focusing the spotlight on other teams.

In conclusion…

It's pretty hard to summarise three years of graft, learning, and change, especially when so many brilliant and diverse people were involved. To assess the impact of DevOps, continuous delivery, agile, or anything else we did, and draw out useful insights, is equally tricky.

There were some key building blocks. Living agile principles enabled us to inspect, adapt, and learn. It feels like we've made progress towards becoming an organisation that learns (Jez Humble has a post on why this matters more than almost anything else.) These habits and ways of working (our culture, you might say) were crucial. Many ideas stemmed from them, as did the impetus to keep improving.

Following the thoughts and principles of people like Etsy, Flickr, Thoughtworks, and Jez Humble's awesome book let us learn quickly. The emergence of the DevOps community and its friendliness and willingness to share (typified by DevOpsDays) helped us realise the significance and benefits of the practice.

Key first steps were to acknowledge failures, stop to plan our next move, and to start to change. Things could have moved quicker, but we promoted agile and DevOps concepts and persevered. At some point, sufficient momentum gathered and we began to evolve rapidly. At the same time, our products improved and enthusiasm increased. Getting time and commitment were some of the greatest challenges. Recognising that some things can't change, yet, and waiting patiently for the right opportunity was just as important.

I think these experiences have shown that DevOps behaviours can be introduced and sustained in a large organisation, but it needs the five P's: promotion; planning; perseverance; patience; and, of course, pizza.

About the Author

John Clapham is a software-development manager in Nokia's entertainment division based in Bristol.

Previously, as product owner for the continuous-delivery team, he helped transform the Nokia Entertainment platform-release process from an expensive, once-every-three-months exercise to a once-every-30-minutes routine. John is passionate about agile, coaching, coffee, and finding new ways to build great products. John can be found on Twitter as @johnC_bristol, and onLinkedIn.

READ THIS ARTICLE ONLINE ON InfoQ
http://www.infoq.com/articles/monthly-devops-01-nokia

DevOps @ Prezi

By Peter Neumark

Learning DevOps from within

Introduction

Prezi's philosophy of running each product team like a mini-startup - as decentralized as possible - means every team needs to be self-sufficient. This is only possible using the classic DevOps recipe of mixing software developers with engineers experienced in operating large-scale systems. As the executive team at Prezi realized this, they pushed for company-wide DevOps. With this initiative, the question of how employees would develop the skillset necessary to implement DevOps became especially important.

Prezi has long had cross-functional teams consisting of designers, QA engineers, developers, user-experience researchers, and product managers. Until recently, however, there was only one DevOps team. Today, this team is still responsible for the availability of most mission-critical services like core databases and HTTP load-balancers. What changed is that individual product teams have assumed ownership of the infrastructure that runs their products or features. Many of these teams did not include anyone with DevOps experience, so it became a priority for them to learn the basics.

The traditional method of learning a technology stack is to enroll in courses typically organized by vendors, often to earn a certification. While this tried-and-true path has many benefits, it's a heavy-handed approach, requiring a significant investment of money and time from both the company and the individual employee. Those who are familiar with much of the material in an introductory course but are not yet ready for the next level will either be bored or confused. Finally, a vendor's course often woefully neglects the big picture, focusing primarily on the problems solved by the vendor's product line, not the problems faced by those enrolled. Because of these drawbacks, engineers are much better off learning from their colleagues if they can.

Prezi started experimenting with different methods of sharing information internally long before every team turned DevOps, so we already had effective ways of sharing the knowledge we had within the company when we needed them. This article shares these techniques.

Boot camp

Learning from teammates is easy if you know them well, but what about new hires? Prezi struggled to get new engineers up to speed on the codebase and company values. The solution was to scale down Facebook's impressive six-week training marathon and create our own two-week boot camp. New hires spend up to a day with each of the teams that make up the organization. This introduction week is designed to impart several different kinds of knowledge. First of all, spending some time with every team at Prezi is a great way to get to know one's new colleagues. It's much easier to remember names after a short conversation or crash course on an interesting

topic than after the usual whirlwind tour with a thousand handshakes. In addition to pairing names with faces, graduates of boot camp will also have a good idea of whom they can approach with a specific problem.

Second, boot camp gives new DevOps engineers the bird's-eye perspective of Prezi's infrastructure and code. Typically, after a short presentation, participants will be given a small but reasonably challenging task. Examples include writing a Chef recipe for configuring an Apache instance and creating a dashboard for the number of prezis exported for use with Prezi Desktop using data from Hadoop. These mini-projects are carefully chosen to be doable in the time available and to be useful in their own right, not just busywork. They also typically require talking to two or three experienced engineers to get advice on how to solve the problem at hand.

Finally, boot camp introduces new employees to what working in a startup environment is like. At Prezi, there are teams but no departments. We don't have too many layers of management, but we do have a user-experience research team. All of these differences can be a bit disorienting for newcomers. Prezi's iterative development style is probably the most difficult new concept for fresh engineers to integrate into their working habits, simply because it's so pervasive. By iterative development, I mean following the build-measure-learn cycle. At Prezi, we try to apply this lean startup approach not just to major product decisions but also to minor, 15-minute tasks and everything in between. It is the responsibility of every engineer to avoid waste by building good-enough solutions which can be tested and, if necessary, scrapped without a major investment of time and energy.

Prezi's urgency in getting working systems out the door is the exact opposite of the quest for a perfect solution the first time around that's so common in academia and large corporations. A recently joined team member remarked during boot camp that hearing every group talking about rewriting this code or rethinking that process during a presentation really drove home the iterative spirit. Now that he has graduated, not only does he know a lot more about how Prezi works, he also knows who could benefit from the knowledge he brought with him. The learning goes both ways.

Meet-ups

Meeting new people is a sure way of encountering new ideas. Meet-ups are an excellent way to find people from other organizations who face similar challenges. In addition to swapping war stories, talking to engineers outside the company can also serve as a reality check. For example, if we're having a lot of trouble with a particular database, it's important to know whether it's a configuration problem on our end or the version of the product we're using is giving others a hard time as well.

Prezi hosts several technical meet-ups, often offering free pizza and beer to participants. It's a classic win-win situation: for the organizers, having a suitable venue at their disposal free of charge is a big help. For Prezi, the arrangement has several advantages.

First, it makes it really convenient for employees to attend meet-ups related to their daily work. Many of our colleagues attended a talk one of my teammates gave on Apache Kafka at the Budapest DevOps Meetup since all they had to do was walk upstairs after work. The next day, everyone who attended was talking about how we should rethink the amount of hardware we allocate for services based on the last presentation of the night before.

Second, it's a great way to foster employees' technical interests. If I'm really into an area of research within computer science or a programming language, all I need to do is find enough people who share my interest. Since I don't have to worry about the venue, the barrier to organizing a get-together is quite low. Even if the topic cannot be put to use in Prezi today, there may come a time when that piece of knowledge is exactly what's needed to solve a problem our company faces.

Finally, hosting meet-ups brings talented engineers to Prezi. It's obvious to anyone who walks through the entrance that Prezi is not the average software company. When these people want to switch jobs, there's a good chance they'll send a copy of their resumé to Prezi.
Bouncing around an interesting idea that came up during a meet-up within the team is an excellent way to start a technical discussion that may benefit every

participant (even if the original idea turns out to be inapplicable to current problems).

Brown-bag talks

Perhaps the oldest form of sharing knowledge, the lecture is alive and well within Prezi. Brown-bag talks (named after the brown bags in which members of the audience bring sandwiches) are optional lunchtime lectures lasting 45 minutes to an hour. The can be anything at all.

We regularly invite guest speakers to hold such talks. Some of the more memorable ones include a crash course on US-EU trade relations by the former Hungarian ambassador to the US and a presentation on how the data-services team at Spotify copes with technical and organizational challenges.

Any employee is free to organize a brown-bag talk. One of the best presentations was on the Paxos algorithm and alternative approaches to data consistency. The presenter became an expert on the subject before he joined Prezi. Members of the audience walked away with a new understanding of possible tradeoffs. In our team, the engineers with operations backgrounds were frantically trying to remember if they accidently configured MongoDB with dangerous settings in light of what they had just learned about data consistency.

What sets these brown-bag talks apart from a typical internal presentation is their informal nature. If someone in the audience realizes that this talk is not for them, they are free to leave. As a result, very few people bury themselves in their laptops or smartphones during such an event. The informal atmosphere also makes it easier for the speaker to deliver his point, since his message does not have to resonate with the whole company.
At the cost of requiring more preparation than a spontaneous conversation, such talks are an efficient way to share one's expertise with colleagues.

Hackathons

The most authentic type of learning is the "on the job" kind. That doesn't mean you have to figure everything out by yourself by reading man pages, however. Solving unusual problems together is a great way for members of a group to learn from each other.

Prezi regularly organizes hackathons at its engineering headquarters in Budapest. Anyone is welcome to attend. The event begins with a brainstorming of novel ideas to implement. Mixed teams composed of both employees and guests form around popular ideas. These teams spend the next day and a half proving the feasibility of their idea through a hack.

Some of the teams work on Prezi.com related projects but this isn't mandatory. A good counterexample is the group that wrote a web app that lists who is currently in the office. The hack works by cataloguing wireless devices connected to the company Wi-Fi network through a good mix of low-level Unix hacking, Python web-application code, and some nifty JavaScript on the client side.

Hackathons are a good way to simulate real emergencies that occur in the life of a DevOps team. The parallels are numerous. In both situations, a heterogeneous team is dealing with a highly ambiguous problem. Time is a limiting factor, so the team needs fast solutions. Finally, the more effectively each teammate can contribute his or her strength, the more likely the team is to succeed. Aside from the technical aspects, the roles assumed by members of the team are also similar in both cases. Naturally, some members will take on coordinator roles, while others will prefer to do instead of talk. Needless to say, a successful team needs both roles. Finding out which of these each team member excels at is vital preparation for the real DevOps disasters, like when a MySQL master node dies, leading to downtime and data loss.

In-house courses

The techniques described so far were all introduced as knowledge-sharing alternatives to vendor courses. Beyond a certain technical depth, however, these more casual approaches often don't work, especially when the technical backgrounds of audience members are too diverse.

With their predictable schedules and moderate pace, organized courses give students an opportunity to thoroughly immerse themselves in a topic.
One example of a course held at Prezi was a short introduction to Haskell programming held on four consecutive Friday mornings. Every team in the

company was represented. There were designers and QA engineers sitting next to developers with years of functional-programming experience. By starting from the basics and moving at a steady pace, the course covered an impressive amount of material. Participants were encouraged to ask questions both of each other and the instructor (who was a colleague). As a result, there was always time for questions and suggestions on how the content of the course could be used in our daily work even if we had to deviate from the original lesson plan. This flexibility and open atmosphere made the course a success even though we use Haskell in few production systems.

Structured in a similar way to the Haskell course, a reading group was launched to focus on classical computer-science articles. Each week, a member of the group presents one of the papers from the reading list. A discussion follows in which any member of the group is welcome to comment on or challenge the presenter's interpretation of the publication.

Vendor courses are generally taught by experienced presenters using professional teaching materials. They are an efficient means of one-way information transmission from an expert to an audience, and there have been cases where it made sense for Prezi engineers to enroll. Prezi's in-house courses are not viewed as direct competition to vendors' offerings. Instead, the reason for organizing in-house courses stems from the realization that the one-way model is not the only means of communication. In fact, it may not even be the most important one. After all, the way a DevOps team solves problems is typically a much less structured process, often requiring one to defend his solution to a problem or ask the right questions in order to quickly find the root cause of problems. In addition to providing useful technical information, these in-house courses and reading groups definitely help sharpen the much-needed soft skills required to make a team operate smoothly.

Prezi being a presentation company, it's particularly fitting that emphasis is placed not just on the technical development of employees, but also their ability to share ideas. By improving the latter, engineers will also become better at helping each other in the former.

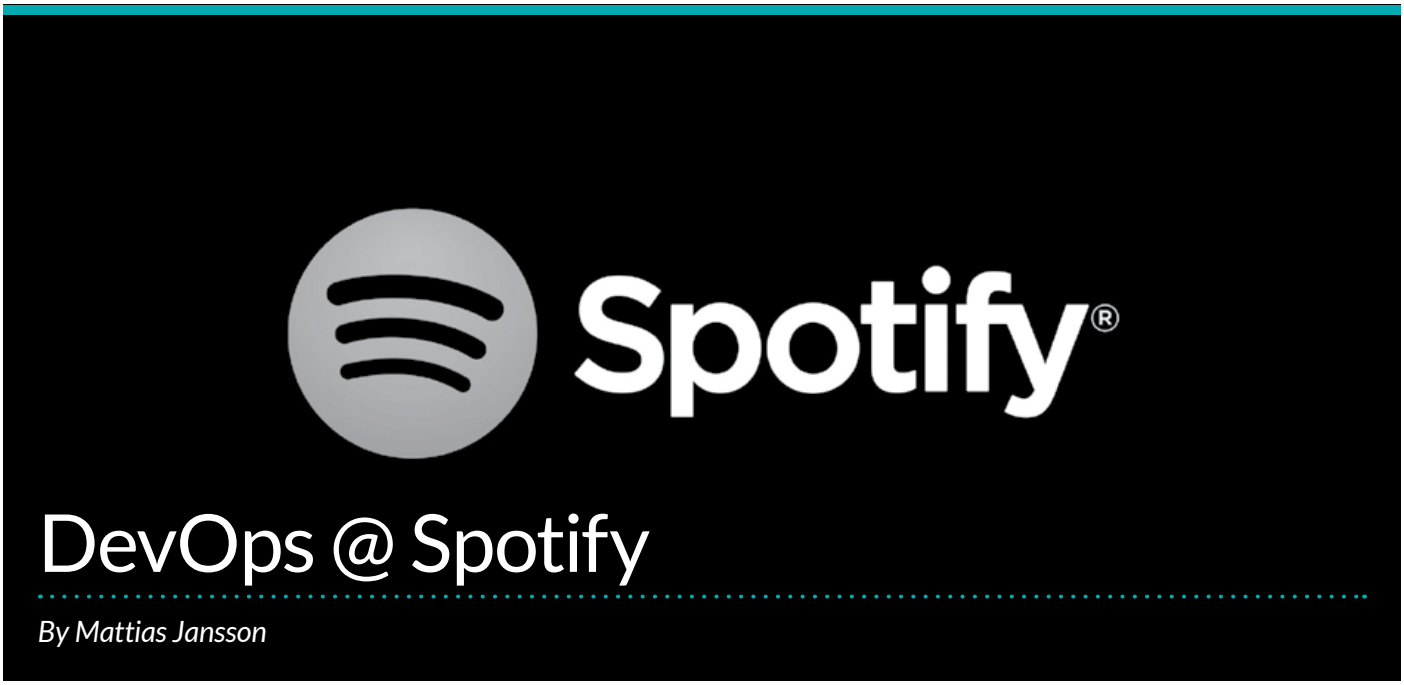
Conclusion

I have learned an incredible amount from my colleagues since I started working as a DevOps engineer at Prezi. It's also great feeling that I have helped others in solving problems and, ultimately, in becoming better engineers. Most people want to work in an environment like Prezi's where everyone is encouraged to share their ideas and any idea can be challenged. The alternative knowledge-sharing techniques Prezi employs go a long way in creating such a setting. Facilitating the unobstructed flow of information within DevOps teams is especially important since, given team members' heterogeneous backgrounds, every member of the team likely knows something others could benefit from. The same can be said for the benefits of establishing dialogue between teams. Regardless of whether or not your company enrolls employees in professional courses, chances are it would benefit from trying out some of Prezi's alternatives.

About the Author

Peter Neumark is a DevOps guy at Prezi. He lives in Budapest, Hungary with his wife Anna and two small children. When not debugging Python code or changing diapers, Peter likes to ride his bicycle.

READ THIS ARTICLE ONLINE ON InfoQ
http://www.infoq.com/articles/monthly-devops-02-prezi



DevOps @ Spotify

By Mattias Jansson

Mapping DevOps lessons to management

There are many blog posts, articles and tweets about DevOps out there on the Internet. Some of them discuss the pros/cons, some the consequences of its introduction, while others discuss how it was implemented.

Although this article refers to some aspects of DevOps adoption, its main focus is on applying DevOps principles to a different area: engineering leadership.

In this article, I'll refer to "us" every now and then. Here's a quick intro:

- Ingrid Franck is the engineering team's agile coach.
- Ramon van Alteren is the engineering team's product owner.
- Mattias Jansson (yours truly) is the engineering team's chapter lead, sometimes lazily called a team lead, like I do in this article.

The engineering team itself consists at the time of writing of nine people.

DevOps culture at Spotify

Many parts of DevOps culture have pervaded Spotify from its early beginnings.

The first six people employed by Spotify were engineers, one of whom had an operational role. This was back when Spotify was just another startup in an apartment. This background and admission of the importance of operational thinking from the very beginning has heavily influenced the relationship between Dev and Ops.

We've come a long way since then. There are now hundreds of engineers at Spotify, spread over four cities and three time zones. Although the DevOps mentality does not permeate the hearts and souls of all individuals in the engineering team and although it is not actually mentioned by name anywhere, one can see it show up everywhere in the day-to-day workflow as well as in conversations by the coffee machine.

Many startups are staffed by developers and the odd business guy. In those firms, the operations engineer is hired once the code has been written and there is a need for someone to deploy and maintain the system. At Spotify, the two camps overlap in responsibilities, skill sets, and interests. We have some uncommonly ops-ish developers here, and likewise many of our ops engineers have a strong developer background. The advantage of this overlap is immense in the day-to-day, solving potential blockers long before they arise. Backend developers deploy their code in production by themselves, with or without an ops engineer to hold their hand. This, in turn, more often than not encourages the devs to think seriously about traditionally operations-focused problem areas such as monitoring, logging, packaging, and availability. Since we have thousands of servers in production,

our ops engineers have moved from thinking in terms of individual servers to clusters of servers. One-off manual fixes on individual servers is avoided when possible; instead, the ops engineer, through code, modifies the state of the authoritative data model of the backend, which in turn reflects onto reality via our configuration-management system, backed by Puppet.

Backend services typically have two so-called system owners, one from Dev and one from Ops. Their core responsibilities reside in their respective dominions: the dev system owner owns the code, design, and architecture while the ops system owner owns the service once it is deployed and is serving traffic. However, these two areas have great overlap, and thus the two system owners regularly discuss scalability, changes in neighbouring backend topology, new products that will affect service behaviour, etc.

A final example is how we have a dedicated team working on automation tools and services. Developer and operations staff work side by side for weeks at a time to solve specific problems raised and prioritised by Ops themselves.

All this being said, our organisation still has a long way to go. The numbers of customers, servers, data centers, services, offices, and staff are growing all the time, and yesterday's solutions have a marked tendency to scale badly into the present. On top of this, the ratio of Dev to Ops staff has changed in a way that has diluted the DevOps mentality in some ways.

Lessons learned

The lessons the DevOps movement has taught us are many, but one of the most important is the value of aligning the goals of Dev and Ops. Get them to work side by side, give them space to learn from each other. By getting the two groups to communicate regularly, the developer will have a chance to understand why Ops need to act like a blocker at times and will learn how to plan ahead and produce changes in alignment with the requirements of the operational environment. Also, once Ops start to see their hardware and running services as malleable data structures upon which one can apply code, the developer suddenly has a different reach. He/she will be able to affect not just the code as it exists in packages but will have much more flexibility with respect to how the packages are applied in production.

Likewise, by injecting operational thinking into the development process, the frequency with which operations engineers need to spend time on interruptions and cleanup is lowered, and their time can be spent on longer-term projects.

Generally, the DevOps work methods have helped both sides of the organisation think of the entire system - the value stream - not just the components they are traditionally responsible for.

Problems in management-land

In many modern tech companies, one can find three distinct responsibilities that touch an engineering team. In some smaller firms, and indeed even in larger ones, these responsibilities are typically gathered in one or two roles. At Spotify, we try to separate them so that three separate people own these distinct responsibilities:

- The product owner (PO) is accountable for delivering products to one or more stakeholders in a timely fashion.
- The team lead's (TL) mission is to maintain the team so its members are fit for the challenges expected of them, and is responsible for the architectural soundness of the internals of the product.
- The agile coach (AC) is dedicated to nurturing an environment of engaged and healthy team members who continuously improve themselves as team members, their product deliveries, and their team collaboration.

These three roles are at times at odds with each other. In most organisations, the people who have these roles have differing missions and can pull the team in different directions.

Have you ever experienced a conflict between the PO pushing for an essential new feature and the TL who is concerned with the team's frustrations over the mountain of technical debt in the existing codebase? Or between the AC who feels that the team needs to stop and reflect more often in order to figure out how to improve and the PO who hesitates and worries that the rhythm of the team will be disrupted? Or when the TL feels that the team is agile enough and actively

blocks the increasingly agitated attempts by the AC to help the team help themselves?

These problems are in many ways similar to the conflicting goals of Dev and Ops groupings in archetypical firms.

Companies new to DevOps often discover blockers (structural, social, cultural, etc.) inside their firm, making adoption difficult. Even people inside firms where DevOps prevails will find plenty to disagree on. Often, the entire problem set resembles two people in a single bed with a blanket that is too small to cover both. The result is a lot of pulling/shuffling of this blanket to try and cover all the exposed parts.
It is not unusual to find old-school operations engineers who refuse to see infrastructure as code, preferring to manually modify configuration files on target machines, or developers who look down on operational work, who feel that they their job is done once the build completes and that whatever happens when the code hits bare metal is someone else's problem. Dev and Ops leadership can be at odds with each other because of a mismatch of missions (the number of features shipped vs. keeping downtime at a minimum).
But what makes DevOps so attractive is that it's all about encouraging developers and operations engineers to talk to and learn from each other. It's all about communication, about aligning goals. Once we start listening to each other, we will have taken the first step towards some sort of DevOps synergy.

So what happens when you do the same thing with an engineering team's closest leadership figures?

What if we get the PO, TL, and AC to regularly talk about their concerns, to discuss their short and long-term goals for the team, and to teach each other the realities within which they live? Will we find similar synergetic effects among these three roles? Will we not only eliminate conflicts between them but also find something... more?

PO TL AC to Potlac

That's what we did, six months ago. The three of us had never worked together in this particular constellation before and we were willing to do some serious experimentation. Our aim was to minimise misunderstandings and to possibly get some sort of synergy.

So what did we do? How did we apply the lessons of DevOps to our work?

Weekly sync meeting

Half an hour every week, we discussed the current state from each of our perspectives. Each brought at least one topic to the session, which we then discussed and digested together. Example topics would include increasing stakeholder involvement, upcoming conferences, or the theme of the next retrospective.

Quick chats and sync before key meetings

Before one of us held a critical meeting, we would have a quick chat with the others to get last-minute feedback. We would reiterate the purpose of the meeting, determine if the goal(s) were realistic, or discuss strategies that would drive the involvement of the meeting attendees.

Regular one-on-ones

Each of us had one-on-one meetings with each of the others once a week, either at the office, over lunch, or with a quick phone call in the evening. In limiting the discussion to two of us, the tone of the conversation and the problems raised became more personal, but all the while orbitied our common goals.

Mock meetings

If a meeting was special or the agenda experimental, we would hold mock meetings with each other, practicing the tricky parts to check for holes in reasoning or to expose unexplained assumptions.

These are the four most obvious ways in which we worked together. The emergent property of this group of three was that we started to think in each other's shoes, and in some sense each of us was suddenly wearing all three hats - although our original one remained larger than the other two. It made us stronger as a group and the more we discussed, the further our cooperation deepened.

A critical success factor for our setup was a shared commitment to the team as opposed to any single engineer. In our opinion, this is largely what made this work: the shared idea that the team is bigger than the sum of its parts.
Some time after we had begun to work in this manner, we accidentally came upon a name for ourselves. I

had at some point placed our photos on a board with our role initials under the pics. it spelled "PO TL AC". When Ingrid (AC) realised that the sequence could be pronounced "potluck", the Potlac name stuck. (A potluck is a meal to which everyone brings food to share with the others. A successful potluck requires coordination among the people who come lest the meal have all desserts and no main courses.)

Benefits of Potlac

"That's all very fine and sounds nice, but what do you get out of it?" you might ask. Well, it depends a bit on what responsibilities you have. Below, we each state the main unforeseen benefits we gained by working together in this fashion.

Mattias: team lead

Management can be a lonely job. While engineers can swarm around a problem, I often cannot. I can ask the team to do many things for me, but some things simply cannot be delegated or shared (I'm thinking career goals, personal confidences, salaries, etc.). Though I cannot share these things with my Potlac colleagues either, there are other topics I can and do share. Examples include discussing new and different ways of solving conventional problems or discussing how to scale our team in a sustainable way. Often, these discussions gave me a piece of the puzzle that helped me understand some problem I was working on.

Since we have an ongoing dialogue in Potlac, we have grown to know each other's visions and our respective views on the state and history of the team. Through this, together with our different networks both inside and outside the company, I benefit in that I see things on the horizon long before I would have otherwise. I can then prepare in time, and snuff out many problems before they become big ones.

Ingrid: agile coach

The Potlac gave me an opportunity and platform to have discussions around agile. It is a stage to dialogue about servant leadership, a forum to find a consensus on what it means. It allowed the leadership team to focus on results and to hold each other accountable. It soon became a sandbox where we hashed out conversations of empowerment, impediments, and conflict. It was also a classroom where we talked about approaches to stakeholder meetings, planning meetings, retrospectives, and one-on-ones. It also developed into coaching sessions where we talked about our failures and what we learned. Instead of three individuals, each working toward our respective goals, we became a team: a leadership team with a united mission of supporting our engineers.

Ramon: product owner

Pushing for delivery can be just as lonely as managing a team. By working so closely together with both a team lead and an agile coach, I gained a multitude of benefits. One of the most important ones is focus. I can focus on delivery of enhancements to the products I am responsible for because I know that my two colleagues are covering the other two equally important aspects of team leadership. Mystifying incidents of the past such as, for example, a sudden lack of commitment by an engineer for some time became a lot clearer with the added information from Mattias on the personal situation of that engineer. Ingrid opened up entirely new ways of handling typical issues with the team, which helped me a great deal.

The second most important benefit I see is the typical thorny problem of avoiding (or repaying) technical debt. An open discussion between people representing the different interests involved makes it easier to approach this problem. Otherwise, it's just an internal debate in a single person's head.

We have seen how our initial experiments with Potlac brought new insights into our day-to-day work dynamics - unforeseen yet somehow expected. This way of putting ourselves in each others' shoes has broadened our horizons and given each of us more context when considering a problem.

At the end of the day, it's really all about exposing context. Context of why a product is necessary now rather than later; context of the background to a team conflict; context to help select the right combination of agile methodologies for this particular team at this particular time.

DevOps helps the engineers practicing it to better understand the points of view of tangential groups of people. It exposes their needs and the requirements set upon them. In a similar way, Potlac has helped the three of us by giving us context in a focused high-bandwidth channel.

So what now?

Though this has been an amazing ride, the environment within which we have worked is changing and we will need to adapt our methods in some way. The team, which consists of nine people (plus us), will probably double in size during the coming year. The company is, as always, expanding and with this comes changing focus. Ingrid has been asked to work as an agile coach elsewhere and Ramon has taken on broader PO responsibilities. Mattias will have a flurry of new engineers in his team to manage. Each of us will need to form brand new Potlac groups in our new surroundings.

One question we ask ourselves is how Potlac will scale with the growth of an engineering team. With a larger number of people in the team, we will most likely have more products to develop and maintain. This implies more product owners. The team lead will not be able to manage this many people; some sort of team split is on the horizon. Finally, we might need two agile coaches if the team grows to this size. Will Potlac function if its members increase from three to six?

Another important question we have considered is how difficult it would be to try to duplicate this leadership model. In software, if a hack helps solve an immediate problem, it is a good thing. However, to really get true value from the hack, it must be documented and portable. Is Potlac portable?

It would be convenient if we could produce a Puppet recipe that covers how to deploy Potlac in other teams. Alas, Puppet does not cover this particular feature, so perhaps this article will help iron out what we did to get the results we described above.

Happy org-hacking!

P.S. If you want to more about how we organise our whole tech organisation, see Henrik Kniberg and Anders Ivarsson's paper on Scaling Agile at Spotify.

About the Author

Mattias Jansson is an engineering-team lead at Spotify. He has worked as a university lecturer and course coordinator, software engineer, site-reliability engineer, and has lately been focusing his energies inside of management. Mattias's current passions include agile, servant leadership, and bridging the gap between tech and non-tech people. You can contact Mattias via Twitter.

READ THIS ARTICLE ONLINE ON InfoQ
http://www.infoq.com/articles/monthly-devops-03-spotify

DevOps @ Rafter

By Chris Williams

The Bootstrap Phase

Over the last six years, I've had the unique opportunity to watch our company grow from just a couple of fresh college grads with an idea of renting textbooks to a large, mature company. When I look back, I tend to see the growth we went through in two distinct phases – pre and post Series-A funding. Unlike most startups you hear or read about, we had a rather long pre-Series A period (almost three years).

In this phase of our growth, we didn't spend a lot of resources or brainpower thinking about DevOps and instead focused on building our product. For most of those first few years, I considered myself a software engineer who sometimes had to spend a few days a month performing systems administration.

We always joked that I was the one who drew the short straw and got stuck managing the servers but, in all honesty, I enjoyed it. It never occurred to me at the time that my love of software development could be applied to managing our servers.

Keep it simple (while you can)

There are a couple of reasons why in the early days we were able to get away without focusing much on DevOps. A lot of it has to do with being a small company and keeping the amount of change low. We chose a simple architecture and had a small number of applications, so while our products were evolving, there was little change exerted on the underlying infrastructure.

Overinvest

Secondly, we overinvested early on in server hardware. We probably could have run our entire site on two physical servers but we had 10. This allowed us to spend very little time worrying about performance or growing our infrastructure, since it took several years for us to hit the limit of the initial hardware investment. There was, of course, up-front cost in configuring these servers, but once they were set up, we made very few changes to them.

Pick good tools

Lastly, because we adopted Ruby on Rails as the framework for our applications, we were exposed to tools that allowed us to adopt good practices early on for release and deployment such as Git, Capistrano, and TeamCity. We also built on top of well-tested, and stable open-source solutions such as Nginx, MySQL, and Memcached.

The exposure to all of these tools and frameworks kept us from having to roll out complex and proprietary solutions, which I think all too often ends up slowing down development as the company grows.

Growing up

As our company entered the second growth phase, we grew our engineering and product teams substantially. The days of only two engineers were gone, and I simply had to blink and we soon reached 10 and then 20 people working on current and new products. As one can imagine, the amount of

change also grew substantially. The sheer number of applications we needed to host on our hardware doubled, then tripled, and developers began wanting to use new types of application frameworks, programming languages, databases, queuing systems, cache servers, etc.

Along with this added complexity, the cost of mistakes and outages also grew. It quickly became apparent that the old days of configuring our infrastructure manually was not going to scale and that our lack of maturity and flexibility at the infrastructure layer would start to cause problems by both slowing the release of new products and features and hurting our stability. With this realization, I stopped working on our products, and switched to focusing on developing automation, monitoring, and release processes full time.

Cooking with Chef

Luckily, as we were recognizing the need to develop DevOps practices at our company, we were introduced to OpsCode Chef and with it, the whole philosophy of "infrastructure as code". We initially spent several months writing all the recipes for automating each piece of our existing infrastructure. Once we were finished and could rebuild all our servers from these automated recipes, we felt an incredible weight lifted off our shoulders. All of our servers were now set up consistently and we finally had a place where anyone on the team could look and see exactly how a piece of infrastructure was set up and configured. Equally important, it also gave us the ability to quickly spin up additional resources with relative ease.

A DevOps team is born

DevOps began to have unique responsibilities in the organization, providing critical application support for our product lines and ensuring our infrastructure could continue to scale. Along with these priorities, we were building sets of tools and products for managing the infrastructure that needed to be continually supported and improved. Especially early on, every new application in the infrastructure often generated changes that needed to be made in our underlying automation. The number of DevOps-related requests by internal customers (engineers and operations folks) also increased substantially as more people began to rely on our work.
Because our engineering organization was already split into separate product teams each focusing on distinct pieces of the business, we decided to make DevOps another team in our engineering organization. This allowed us to dedicate engineers that could solve infrastructure problems full time. In addition, availability and reliability of our application platform is extremely important to the business, as outages and issues can have a big impact on the bottom line. We needed to ensure there were always dedicated and well-trained engineers available to assist and investigate issues, especially when ownership of the issue might be shared across several teams or not immediately obvious.

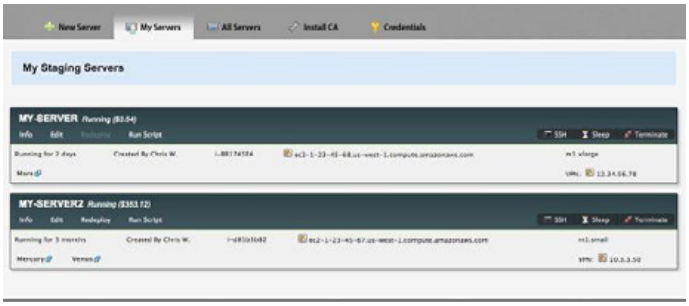
The fruits of automation

On-demand provisioning

One of the things we did almost immediately after adopting Chef and automating our production infrastructure was to improve our test and staging systems with the same level of automation. A common complaint we heard was that engineers could not easily demo to the business people what they were currently working on. In response, we built a 100% self-service portal that allows anyone in the company to spin up a preconfigured server running our full stack on EC2.

A user can choose which of our applications they want installed on this server at a specific revision, and they can choose a test database or use a scrubbed snapshot of the production database from a time of their choosing. One of the big wins from this system is that we simply reuse the exact same Chef recipes that we already use to build our production servers. This allows us to flush out many potential issues before they ever land in production. Our engineering and QA teams can also feel more confident they are testing their new features on servers that are set up identically and run the same versions as our production servers.

This staging system has been immensely popular at our company. Many employees have remarked how it took weeks and lots of paperwork to get demo servers requisitioned at their previous companies. On our system, anyone can have a staging server up in as little as 15 minutes. Building such a system would be impossible without having a strong automation foundation in place.



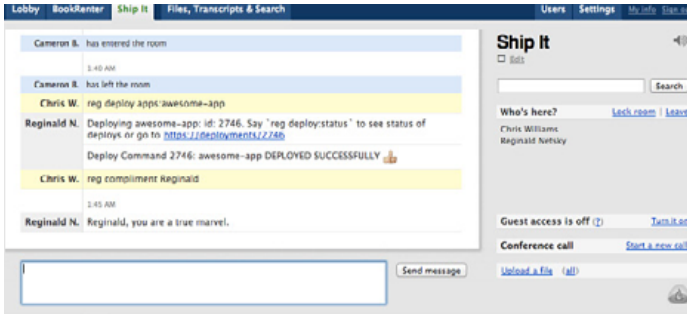
Datacenter failover in minutes

Another difficult task that DevOps automation has helped us solve at Rafter is performing datacenter failover. About two years ago, we decided that we wanted to have the ability to switch at any time to a secondary datacenter to reduce the impact and cost of outages. Since we are in the education industry, our business is seasonal and suffering a long downtime during back-to-school seasons can be costly. This also benefits our operations folks tremendously since they can perform risky datacenter maintenance when there is no live traffic going to that particular datacenter.

Having an automated infrastructure in place has allowed us to meet this difficult goal with relative ease. When we first started out, I would have been shocked if someone told me one day we would be able to failover entire datacenters in a matter of minutes. But relying on our automation to do all of the heavy lifting, we are able to design an infrastructure that supports this goal.

Shared deployment

Another notable area of work for our DevOps team has been on improving our deployment processes. From the beginning, we'd used a great tool called Capistrano to manage our deployment and we were generally happy with it. One improvement we did make to it was to teach our Campfire bot (named Reginald) how to deploy using Capistrano. When our product engineers want to deploy their applications now, they simply ask Reginald to do it for them.



The biggest win here has been that it makes deployment more of a shared experience. Anyone in the Campfire chat room can see a deployment going on and if there's a problem, someone can immediately jump in and help. All errors and deploy logs are stored in our database and are viewable from a web application which Reginald points developers to. This makes it much more convenient to share any potential issues. Previously, when the developers were deploying from a server, it was more of a private operation. When you have to deploy publicly, it's a lot easier for everyone on the team to know what's going on.

Self-service tools

We've always designed the products our DevOps team builds and uses to be self-service whenever possible. The success of automation is all about removing manual roadblocks, especially ourselves! Giving people the tools and platform for managing parts of the infrastructure that they care about makes the entire organization perform more efficiently. Especially where others can actually do the work for you (and probably do it better), your tools should empower them to do it. We try to keep this philosophy in the tools and practices we adopt.

Our team makeup

I've found that there tends to be a wide range in the industry of where a DevOps organization falls in terms of development vs. operations. Our DevOps team was formed out of the product-engineering team so it has always been staffed with traditional software engineers. All current team members have spent time building features on our main product lines before switching to DevOps. We've also been blessed with a fantastic operations team that handles the care and feeding of our hardware and datacenters, so we have the luxury to focus solely on software.

Focus on development

I believe the heavy focus on the dev side has worked well for us. As we build and support a more complex application infrastructure, our need for traditional software development will only grow. That said, we have found hiring to be difficult, as many software engineers do not have a lot of interest in ops/infrastructure and many operations engineers don't have enough experience in traditional software-development roles. It definitely requires a special individual to fill these shoes. At Rafter, I've noticed that DevOps seems to attract those with deep knowledge in certain areas as opposed to generalists. Currently, our DevOps team consists of three engineers, and we are always looking for more talented people.

One aspect that makes our DevOps team unique is that we often take on tasks that normal engineering product teams or site-reliability engineers might accomplish. For example, we've led projects such as upgrading our applications to work with new major versions of Ruby and Ruby on Rails. If you've ever been involved in a major Rails upgrade on a large application, you know this is no easy undertaking and requires a lot of expertise with both Rails and the underlying codebase. We also often assist product teams in debugging and solving performance and scalability issues in their applications. This type of work has helped expose our team to all parts of the software platform that we have to support.

Day to day

Support

A typical day for members on our DevOps team is a combination of resolving support requests, investigating alerts, and working on longer-term projects. On average, we spend about 50% of our time on support and alerts and 50% on project work, with requests and alerts taking priority because they are usually very time-sensitive. The support requests cover variety of issues but typically relate to application support. For example, an engineer would like to add a new application to the platform or would like to make an infrastructure change to an existing application such as spinning up more servers. Sometimes, these requests can be large in scope and require significant changes and testing such as hosting a new programming language or a new database.

Monitoring and alerts

Another common set of requests involves investigating application alerts (either automated or reported by engineers). Often, DevOps acts as the coordinator for alerts, performing the initial investigation and finding the right team that can best resolve the issue. This is not to say, however, that we handle all application alerts. We investigate alerts that affect infrastructure (e.g. issues involving over-utilization of network, I/O, CPU, and memory) or areas that have shared responsibility and usage among the engineering team (e.g. shared libraries, databases, and legacy applications).

Project work

Often, the work we do in supporting engineering requests and investigating alerts exposes areas that need more investment. Specifically, these are areas where we aren't providing engineers with enough tools or information to solve the problem themselves. One current example of this is that our engineers cannot deploy their own cronjob changes. When an engineer needs to modify a cronjob for their application, they need to open a support request with us. After doing 10 of these requests a week, it quickly becomes apparent that we need a tool that allows our engineers to deploy their own cronjobs. Thus, a significant portion of our project work comes from deficiencies that we discover through support and alert requests.

We also spend a significant amount of time upgrading our systems and performing the testing required to be confident in the upgrades. I could fill up several pages with all of the different types of servers, databases, applications, libraries, and operating systems in our

platform. Keeping everything running recent and secure versions is enough work to occupy several engineers all year round.

Another source of project work comes from direct external requests. For example, management might request that we need to be able to fail over to another datacenter, an auditor asks for a specific security feature, or an engineer requests infrastructure changes that might affect many applications.

DevOps as a platform

As our company has grown, DevOps has found a place in between our engineering product teams and operations team. I think of it as a three-layer cake. At the bottom layer is our operations team, which is responsible for acquiring and setting up the physical hardware. In the middle is our DevOps team, responsible for providing a platform to use these hardware resources. The top layer is our engineering product teams that use the DevOps platform to deploy, monitor, and host their applications.

I think platform is a key concept here when talking about DevOps. When I first started in DevOps, I used to think of our tools as separate, independent items. However, once you start building out these individual pieces, you realize that each component fits together in a larger platform and requires access to a common set of information. Each of these tools needs to be integrated, otherwise a lot of complexity and duplication will slowly creep in.

Building a DevOps Platform

Transforming our DevOps tools into a platform of services is a current and ongoing project for us. This might seem strange, but when you think about it, there are many different clients that need information about our infrastructure. For example, some of the applications described in this article might need to access the platform in the following ways:

- Our staging-server portal needs a list of applications that can be installed on staging servers.
- Our deployment framework needs to know which applications can be deployed and where to deploy them.
- To switch datacenters, we need to know how to bring up applications in the other datacenter.
- Our Chef recipes need to know how to configure servers properly based on the applications we ask it to install.

From these examples, there's a set of common information that can be exposed via services to answer basic questions about the infrastructure.

One can take the idea even farther and develop additional services that affect change in the infrastructure. Examples might be a service for putting a server or application into maintenance mode, a service for adding new applications to the platform, etc. Some DevOps organizations might implement these features as ad hoc scripts. This has a downside, though, since usually only a DevOps engineer can run these kinds of scripts. A service has the benefit that other tools and applications can interact with it in a common way, and we can build applications on top of these services for further flexibility. This also supports the idea of building tools that are self-service whenever possible.

Chef as a database

We have also shifted how we see Chef in our ecosystem. It is far more than just a tool for automating our servers. We see it as the data layer in our DevOps platform. By taking advantage of Chef's platform and APIs, we can store and query information about all of the servers and applications in our infrastructure. While Chef provides the data-storage layer, we are building our own set of services on top to access this data and provide a flexible means of interacting with it.

Our job is never finished

All of this comes down to ensuring we can continue to scale, and that the infrastructure is never the roadblock that prevents a new product from being launched. Even after there is significant automation in place, you have to keep iterating on top of that existing automation.

One example of this readily comes to mind. Our team spent a lot of effort building out a framework for how our applications are set up on each server. The only piece now required is to write a small JSON file that describes basic information about the application and its dependencies. We then patted ourselves on the back and went to tackle the next problem.

However, we noticed we were adding a lot of new applications and spending a decent amount of time writing these configuration files usually through several inefficient back-and-forth conversations with the product engineers. It soon became apparent that we had become a bottleneck in our own automation. The job of the DevOps team is never finished and what may seem like enough automation today can still slow you down tomorrow.

About the Author

Chris Williams is a co-founder of BookRenter.com, the first online textbook rental service, which became Rafter Inc. in 2012. Today, he manages the DevOps group at Rafter and oversees infrastructure automation, deployment and release processes, and platform availability.

READ THIS ARTICLE ONLINE ON InfoQ
http://www.infoq.com/articles/monthly-devops-04-rafter