DevOps
Making It Easy to Do the Right Thing

Matt Callanan and Alexandra Spillane, Wotif

// Wotif used DevOps principles to recover from the downward spiral of manual release activity. By defining deployment standards for development and operations teams and making them easy to adopt, the company drastically improved the average release cycle time. //

WOTIF GROUP IS one of Australia's largest travel e-commerce platforms, encompassing Wotif.com and several other brands (acquired by Expedia in late 2014). In 2013 and 2014, the organization overhauled its software release processes, reducing the average release time from weeks to hours. A DevOps and continuous delivery (CD) transformation enabled this overhaul.

Beyond the mechanics of building CD pipelines, this article focuses on the technical and cultural issues Wotif faced and how it overcame them. In particular, Wotif faced the dilemma that many engineering departments face: teams want to collaborate on infrastructure, test, and deployment automation but can't find a way to do so. To deal with this dilemma, Wotif embraced the idea of "making it easy to do the right thing."

The Context

In 2012, Wotif was in the midst of what Gene Kim might call a downward spiral of problems,[1] with an unwieldy release process that no longer met its needs. The engineering department consisted of approximately 170 staff. A centralized operations team comprised 11 system and database administrators. Approximately 60 developers, 30 testers, and 15 business analysts were spread across 12 development teams in three cities. (Approximately 20 percent of those teams were involved in brand-specific feature development and thus are outside this article's scope.) The development teams were experiencing increasing frustration with delays and problems releasing software to production, while the operations team felt overloaded and constrained. Although many people were outspoken in their opinions of where the biggest problems lay and how to fix them, no discussion forum or avenue for improvement existed.

Whereas the engineering organizational structure was fairly flat, with only three management levels, the software release process was overly bureaucratic. Releases had to be booked weeks in advance and were often deferred at the last minute in favor of higher-priority releases. Releases also had to contend with a cumbersome gatekeeping process to reduce the perceived risk of change. (For instance, all releases had to undergo up to 48 hours of performance testing, even for cosmetic changes.) Because the prep-for-release process was difficult, teams would bundle multiple crosscutting changes into each release, causing release batch sizes to trend upward. This, in turn, further complicated prep-for-release, in a self-perpetuating cycle. These difficulties meant that teams were hard-pressed to keep up with business demand for features and manage technical debt. Like many organizations, Wotif suffered from all the problems of the Deploying Software Manually release antipattern in Jez Humble and David Farley's book *Continuous Delivery*.[2]

In the midst of these issues, Wotif's engineering department made the architectural decision to move to microservices[3] based on the Dropwizard framework (www.dropwizard.io) for Java-based Web services. Until then, we had built most applications as Java EE (Enterprise Edition) applications on GlassFish (https://glassfish.java.net) servers. Most functionality resided in several monolithic code

0740-7459/16/$33.00 © 2016 IEEE

bases up to eight years old. Although we found microservices advantageous architecturally, this architecture required significantly more release effort because we had to manually deploy and test each microservice. This compounded our release problems.

Additionally, the adoption of Dropwizard by several distinct development teams resulted in inconsistent application behavior and deployment mechanisms. Within three months, a dozen new microservices were being readied for production, all with slightly different behavior (startup scripts, configuration files, administration endpoints, logging locations, and so on). These diverse services were increasingly difficult for the operations team to support and time-consuming to release. Operations staff found themselves spending most of their week releasing software, and both internal- and external-facing projects suffered.

Defining the Right Thing

Partly to address these problems, Wotif formed a CD team comprising two developers, a system administrator, and a process improvement business analyst. The team's overarching goal was to reduce the average release cycle time from weeks to less than a day. (Cycle time was the time for a software change to progress from "development and testing complete" to "released").

We identified that the crucial bottleneck in the release process was the relatively small operations team, a situation akin to the fictional one Kim and his colleagues presented in *The Phoenix Project*.[4] We opted to improve consistency and predictability in the development and release process by standardizing application packaging and application deployment mechanisms—the "operational interface"—while allowing development teams the freedom to innovate within each application code base.

We also decided that, in terms of infrastructure, our solution should be an evolutionary change achieved by extending, streamlining, and automating our existing infrastructure and tools. This decision let us avoid or mitigate many infrastructure and platform change costs traditionally associated with CD transformations. For example, we operated all our environments in colocated datacenters, on machines running Red Hat Enterprise Linux. Changing this basic platform (for instance, moving to a public cloud) was outside the CD team's mandate.

Having identified packaging and deployment standardization as the best focus areas, we continued work that had already started as part of a grassroots initiative: collecting ideas from various teams and individuals about their preferences and requirements. Initially, we gathered ideas through one-on-one discussions with developers and operations staff. These discussions served to not only gather initial requirements but also reduce friction between development and operations. We incentivized the discussions as much as possible. We reminded developers that their input would result in more frequent, easier releases. Similarly, we encouraged operations staff to contribute ideas that would make life easier during after-hours support calls.

As the development and operations team members became more accustomed to collaborating on the standards, we held larger group meetings, including representatives from development, operations, testing, architecture, and management, to flesh out our ideas. Using Atlassian's Confluence software (www.atlassian.com/software/confluence), we collected the ideas raised in these meetings and called them *lightweight application deployment standards*. In this first pass, we identified and captured approximately 30 standardization targets, including log file location and format, initialization script location and invocation, configuration file location, packaging conventions, and SSL (Secure Sockets Layer) configuration and certificate packaging. Where possible, we included implementation guidelines suitable for Dropwizard applications.

We trialed and refined these initial standards over a few months. Components deemed infeasible or unsuitable for immediate implementation were moved to a section we called Future Considerations. When we felt that the standards were sufficiently mature, we finalized that version and released it to the development teams. We then continued working on refinements that would eventually be released as new standards versions.

Making It Easy

Although establishing the standards was crucial to finding common ground between development and operations, we recognized that implementing those standards would slow teams down. Developers and operations staff would need to be intimately familiar with the standards before every software release—and would need to refamiliarize themselves every time a new standards version was published. However, many development teams lacked the time or business priority to fully comprehend and implement the standards. In addition, operations staff felt significant pressure to release software regardless of any seemingly trivial operational issues. So, we realized we

needed the ability to automatically verify standards compliance.

Automated Verification

We developed an automated compliance test suite using the Fabric Python SSH (Secure Shell) library (www.fabfile.org). The suite remotely executed basic Linux commands (such as `ls`, `lsof`, `jar`, `ps`, and `rpm`) to verify that the system and application looked and behaved as expected after each application deployment. The same set of 24 compliance test cases would run on all deployments to all environments.

This brought operational considerations into the development process. The compliance tests would run on every commit, during deployment to the acceptance test environments, blocking that release candidate if it didn't meet one or more of the standards criteria. Developers thus received fast feedback on how their services behaved operationally and could rectify potential issues well before release. They could run the compliance tests for the target standards version in a test environment and use the failures to test-drive[5] the updates required for compatibility.

We also annotated test cases with the standards versions to which they applied, to make the test suite backward-compatible. This ensured we could still check compliance for applications built according to old standards versions, allowing us to continue supporting those applications without forcing teams to update.

The Reference Implementation

We created `helloworld-service`, a reference implementation compliant with the latest standards version. This served three purposes. First, we deployed `helloworld-service` to all environments (including production) to demonstrate and test end-to-end pipeline automation. Second, every time our Fabric platform was updated, the `helloworld-service` build pipeline would be invoked, which would serve as an automated acceptance test of the automated compliance testing and deployment. Finally, `helloworld-service` acted as a template code base that developers could easily duplicate to create new microservices based on the latest standards version.

> We decided to make all microservice releases independent by building in backward compatibility with each release.

Deployment and Test Automation

A crucial step in building our CD pipelines was automating deployment. The prevalent opinion was that automating this process would be impossible because it involved so many manual steps and required the involvement of several staff members.

We created a DevOps toolchain[6] consisting of Git, TeamCity, Yum, Hiera, and Puppet. We orchestrated it with a Fabric code base we named Rolling Upgrade. The Rolling Upgrade tool duplicated the manual steps performed during production releases, with extra checks and assertions to fail the upgrade should any issues be detected.

To automate testing in Rolling Upgrade, we mandated (as part of the standards) the deployment of a self-testing *smoke test* script with every instance of the service. If that script was missing on a node, the deployment (and thus the release) would fail. The script was a simple Linux shell script named `smoketest.sh` that assumed the service it was testing was already running on `localhost`, indicating failure via a nonzero exit code if errors occurred. Different applications had different smoke test styles depending on the application's risk profile. The simplest examples might access known HTTP endpoints on `localhost`; more advanced scripts might invoke a subset of the project's Cucumber (https://cucumber.io) acceptance test suite. Smoke tests were to be "nondestructive"—that is, using read-only access or writing only to previously approved test data. These tests were also available for monitoring tools to execute or for operations staff to use during maintenance or firefighting.

In Rolling Upgrade's first utilization during a release to both preproduction and production environments, it reduced deployment time by approximately 85 percent. Deployments that previously took hours could now finish in minutes.

Release Independence

Even after we automated deployments, our gatekeeping and release-scheduling issues still caused long delays in our CD pipelines. We also faced another problem: ensuring that the versions of those microservices that had been tested together were the same versions released to production.

We considered an approach that kept track of which versions were deployed and tested together and then deployed those applications together as "snapshots" or "release sets." This approach felt cumbersome and

counter to the loose coupling we wanted between microservices. We decided instead to make all microservice releases independent by building in backward compatibility with each release. Teams had the autonomy to release whenever they were ready but also had to support existing calls made by client microservices.

SLIPway: A New Pathway to Production

We reviewed our release methodology, concluding that we needed a radically new release process. This process would be an alternative pathway to production, rewarding those teams that wanted to embrace the CD and DevOps principles of reduced batch size and global optimization[7] embodied in the standards. We called our new process SLIPway (Simple Lightweight Independent Pathway).

SLIPway rules. SLIPway optimized for release speed by constraining complexity, through a set of simple rules:

- *Keep changes independent*. We restricted SLIPway releases to a single application with no dependencies. Any prerequisite database, networking, or OS changes had to occur ahead of time through change requests. Applications had to be backward-compatible with current clients.
- *No manual testing occurs during release*. We facilitated automated testing through the deployment framework. Manual testing was permitted in a given environment only after the release to that environment was complete.
- *Release slots can't be reserved in the calendar*. Time-critical functionality had to be released ahead of time, hidden behind a feature switch.
- *Applications must comply with the latest standards version*. This ensured that teams kept their code bases up to date with the DevOps improvements represented by the standards. Applications not implementing the latest version were prevented from releasing through SLIPway.
- *Operations staff can roll back releases after hours*. If an after-hours production issue related to a SLIPway release emerged, operations staff could roll back that release at will.

After we introduced these simple rules, the average release cycle time decreased from almost two weeks to around one day. Teams could now deploy on the same day they were ready to deploy—diminishing the inclination to bundle many changes into large batch sizes. This reduced risk in two ways. First, the reduced batch sizes made identifying defects much easier than in large change sets. Second, hotfixes could be released much faster owing to the streamlined release process.

SLIPway details. Development teams requested a SLIPway release by adding their release to the SLIPway release queue. The operations team would service this first-in, first-out queue within 24 hours. This eliminated prioritization issues and allowed multiple releases in quick succession with relatively little waiting.

For load testing, we put the onus back onto the development teams, giving each team the responsibility to determine how much load testing to perform for a given release. Teams generally preferred creating smaller, more focused performance test suites using tools such as Gatling (http://gatling.io), rather than the heavyweight, centralized, proprietary tools they had used previously. This let them identify and react quickly to any performance issues in test or production. This also meant that load testing no longer caused a bottleneck in the release process, because the previous centralized load-testing tools and environments were no longer in contention. Compared to our staging environment, the teams' test environments were less representative of production capacity and load. However, we accepted this trade-off for the ability to fail fast in production and roll forward with performance fixes.

Other forms of nonfunctional testing, such as security testing and disaster recovery testing, were conducted out-of-band at regular intervals both before and after SLIPway's introduction. Altering these forms of testing or integrating them into SLIPway was outside the CD team's mandate.

Because staff were familiar with using Confluence to manage and view releases, we based the SLIPway process on the manual Confluence release-notes page creation process. We built Fabric `slipway` tasks to create release-notes pages from Confluence templates and annotate them with sign-offs, time stamps, and labels during various release activities. This let us generate cycle time metrics for each release and measure SLIPway performance and adoption over time. Additionally, using Confluence's Task Canvas Blueprint, all teams could visualize every application's progress through the SLIPway process.

To further increase the visibility of release progress, we built automation that published progress messages to a shared chat room. The shared chat room made every release's progress and outcome more

visible and eliminated the need for release-specific, ad hoc chat rooms and unpredictable coordination of roles. We also captured release transcripts using the Linux **script** command and attached them to each release-notes page. This made precise release execution history available to a wide audience for troubleshooting and historical comparison.

We facilitated rollbacks by appending the relevant Fabric commands to every SLIPway release-notes page. Rollback involved executing the rollback command from the release notes for a problematic release. This command released the previously installed version and annotated the release notes with a "rollback" label.

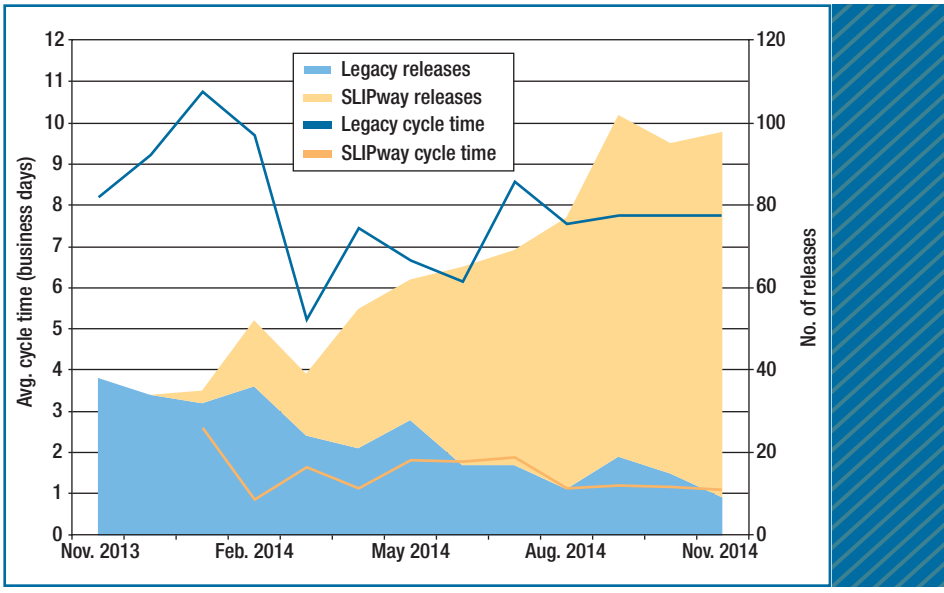

FIGURE 1. The introduction of the SLIPway (Simple Lightweight Independent Pathway) release process in January 2014 produced an immediate drop in cycle time for applications using it. In addition, the total volume of releases increased.

Summary

Defining the right thing—our deployment standards—and making it easy to adopt through compliance testing and deployment automation paved the way for SLIPway. With the simple rules underpinning SLIPway, we created an alternative release path that encouraged DevOps practices such as fast feedback, small batch sizes, and independent releases, emphasizing increased team autonomy.

Results

As Figure 1 shows, in one year, more than 90 percent of releases were performed through SLIPway, with an average cycle time of 1.1 days per release. This represented a 95 percent reduction in person-hours spent releasing, an 86 percent reduction in release cycle time, and a 2.6-fold increase in release count per month.

Comparing 2013 to 2014 (that is, before and during CD adoption), we saw the rate of releases that potentially introduced a customer-impacting problem halve, from 15 to 7 percent.

Continuous Deployment

In late 2014, building on our maturing deployment pipelines, we trialed the adoption of continuous deployment for a single application. We released every code commit to production with no manual intervention—as long as it passed successfully through the required automated testing and deployment gates.[8] We achieved this by elevating the permissions of hardened TeamCity agents and building bots that automatically performed releases for the operations team.

The trial results were immediately positive. On the first day, 10 releases occurred with no manual intervention. The average release cycle time was 21 minutes (approximately 200 times faster than pre-SLIPway releases), indicating that further improvements in delivering software were possible. Fast production feedback from tiny batches meant that development teams could adjust course throughout the day on the basis of real-world metrics.

Costs

Over that year, the CD team allocated approximately 50 percent of its effort to implementing and rolling out the tools and processes required to achieve these results. This effort was the equivalent of approximately two full-time employees. Additionally, before the CD team's formation, some of its members had already laid the groundwork for Rolling Upgrade and standardization—approximately three months' effort.

By the end of 2014, the overall ongoing effort to perform releases dropped by the equivalent of approximately three full-time employees. Implementing Rolling Upgrade, SLIPway, and continuous deployment required no extra investment in

ABOUT THE AUTHORS

MATT CALLANAN is a senior software engineer at Wotif and Expedia. He organizes DevOps Meetups and conferences and is passionate about helping organizations realize their potential through automation and reducing feedback cycles to quickly deliver value to customers. Callanan received a B.I.T. from the Queensland University of Technology. Contact him at matt@mattcallanan.net.

ALEXANDRA SPILLANE is a senior system administrator at Wotif and Expedia. She's passionate about DevOps, configuration management, Web operations and management, and the cultural challenges facing organizations at all levels as they move to more rapid, predictable software delivery. Spillane received a B.I.T. from the Queensland University of Technology. Contact her at alexandra@spillane.me.

physical infrastructure or software licensing. By implementing CD and continuous deployment, we achieved release rates that with our original processes would have required a significantly higher head count and hardware investment.

Lessons Learned

We learned three lessons in particular as we adopted CD.

Warranty and Adoption Periods

The rollout of new standards versions to the development teams frequently highlighted practical issues that meant the new standards needed revision during the adoption period. To strike the right balance between consistent standards and flexibility in meeting real-world conditions, we introduced a two-week "warranty" period. During this period, teams could surface any revisions that needed to be made before the standards were finalized. Any issues raised after that time would be addressed in a subsequent standards version. After the warranty period came a two-week adoption period before the new standards version became the minimum requirement for SLIPway.

Migration Notes

As the standards evolved, teams found that identifying the necessary changes for an application update became burdensome. This was because not all developers were involved in or familiar with all releases of their team's applications or the standards development process. To aid the development teams, we provided migration notes with each version of the standards. These notes described the differences from the previous revision and listed the steps required to update applications.

Semantic Versioning

Our microservices shared several in-house Java libraries, and we implemented many standards requirements by adding functionality to these libraries. Teams could adopt a large portion of a new standards version "for free" simply by depending on new versions of these libraries.

However, some of these libraries used simple date-based versioning. This meant that upgrading to a new library version required for a new standards version involved adopting all changes since the previous library version. This might have included major changes that weren't required for standards adoption. In some cases, development teams updating applications had to unexpectedly spend days or weeks integrating major but ultimately irrelevant changes.

Semantic versioning[9] helped correct this issue. We identified significant non-standards-related changes as breaking changes that warranted a major version number increase. Library maintainers then made standards-related changes as backward-compatible minor revisions to any older major versions still in use by applications, without forcing development teams to adopt breaking changes.

Remaining Challenges

Aligning the Confluence pages, Fabric test suites, reference implementation, and configuration management code bases while smoothing the adoption process required significant ongoing coordination. We needed to find a way to automate not only the testing and deployment of the standards but also the standards themselves. We needed them to become "executable documentation" in the form of "infrastructure as code."[10]

We began research into replacing standards documents with a standardized microservices platform

built on Docker containers.[11] A layered approach to building Docker containers would let operations control the base image and maintain a standardized operating environment. Development teams could then build microservices out of any technologies on the basis of those centralized base images, balancing standardization and innovation.

Other directions included using products such as Mesosphere DCOS (Data Center Operating System; https://mesosphere.com) to abstract clusters of machines as a single OS, on which Docker containers could be deployed. Teams wouldn't need to be concerned about where the applications were physically located. This would free operations staff from manually allocating resources to individual applications.

Another challenge lay in managing database schema upgrades. We encouraged teams to adopt the incremental-change approach outlined in the "Managing Data" chapter of *Continuous Delivery*.[2] This approach decouples a database change from an application release by making changes backward- and forward-compatible with each other. Although some teams started adopting this approach, most releases requiring database changes employed the traditional release process. We started automating the incremental-change approach using Liquibase (www.liquibase.org) to reliably perform automated database manipulation during releases.

Standardizing our deployment mechanism and building a platform centered around the DevOps ideals of collaboration and fast feedback broke us out of our downward spiral. These initiatives noticeably improved team morale, particularly because teams could release frequently and with confidence. They also provided an avenue for practically implementing DevOps-driven changes that could be adopted across the entire organization.

Should all microservices write logs using a common request-tracing mechanism? Should we move all applications from using configuration files on disk to a central feature switch service that operations staff can easily maintain? Previously, the barrier to rolling out such changes was too high. Interrupting business priorities to address seemingly trivial operational concerns was considered unconscionable, owing partly—ironically—to long release times.

By making the right thing obvious and easy to implement, we removed many of the frustrations and hindrances experienced by those developing and deploying applications. The lightweight application deployment standards and SLIPway made it easy for development teams to adopt new operational functionality. We also encouraged the flow of new ideas by providing a fast-path alternative that incentivized adoption of DevOps practices.

Acknowledgments

We thank our Wotif Group colleagues (particularly Danny Williams and Andreas Grossnick) and this article's reviewers. This article represents our own views and doesn't necessarily reflect those of our employer. Wotif.com launched in 2000 and was listed on the Australian Securities Exchange in June 2006 as Wotif.com Holdings Limited, under the ASX code WTF (Wotif Group). Wotif Group has been wholly owned by Expedia Inc. (NASDAQ: EXPE) since November 2014. Today, the Wotif Group comprises the Wotif.com, lastminute.com.au, and travel.com.au brands in Australia, and Wotif.co.nz and lastminute.co.nz in New Zealand.

References

1. G. Kim, "How to Better Sell DevOps: The Construction of 'The Phoenix Project,'" blog; http://itrevolution.com/construction-phoenix-project-and-selling-devops-downward-spiral.
2. J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Addison-Wesley, 2010.
3. J. Lewis and M. Fowler, "Microservices," 25 Mar. 2014; http://martinfowler.com/articles/microservices.html.
4. G. Kim, K. Behr, and G. Spafford, *The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win*, IT Revolution Press, 2013.
5. K. Beck, *Test Driven Development: By Example*, Addison-Wesley Professional, 2003.
6. D. Edwards, "Integrating DevOps Tools into a Service Delivery Platform," video, 23 July 2012; http://dev2ops.org/2012/07/integrating-devops-tools-into-a-service-delivery-platform-video.
7. K. Donley, "TGO & Print Media in the Digital Age," blog, 24 Oct. 2014; https://multimediaman.wordpress.com/2014/10/24/tgo-print-media-in-the-digital-age.
8. C. Caum, "Continuous Delivery vs. Continuous Deployment: What's the Diff?," blog, 30 Aug. 2013; https://puppetlabs.com/blog/continuous-delivery-vs-continuous-deployment-whats-diff.
9. T. Preston-Werner, "Semantic Versioning 2.0.0"; http://semver.org.
10. "Infrastructure as Code," Puppet Labs; https://puppetlabs.com/solutions/infrastructure-as-code.
11. "Understand the Architecture," Docker; https://docs.docker.com/engine/understanding-docker.