The Transformative Technologies Driving DevOps

CloudBees®

Contents

Introduction

Microservices, Containers and Everything as Code - The Perfect Trio for DevOps

More and more organizations recognize the value of adopting DevOps practices to accelerate delivery cycles while improving quality, reliability and security. As a result, the disparities between organizations who have begun a DevOps transformation and those who have not yet are becoming more pronounced. The growing advantages of the haves over the have-not-yets are highlighted in a recent State of DevOps Report.[1] In addition to noting that DevOps practices "improve organizational culture and enhance employee engagement," this report cites several key findings on high-performing organizations. The study found that compared to their lower-performing peers, high-performing organizations have better employee loyalty and spend less time on unplanned work and rework. They also deploy 200 times more frequently, recover from failures 24 times faster and have 2,555 times shorter lead times for changes.

Many of the high-performing organizations surveyed for the report started down the road to DevOps by bridging the chasms that often exist between upstream development and downstream delivery across three planes:

- People and culture
- Process and practice
- Tools and technology

The three planes in this trinity are interdependent such that establishing an effective DevOps culture requires addressing all three.

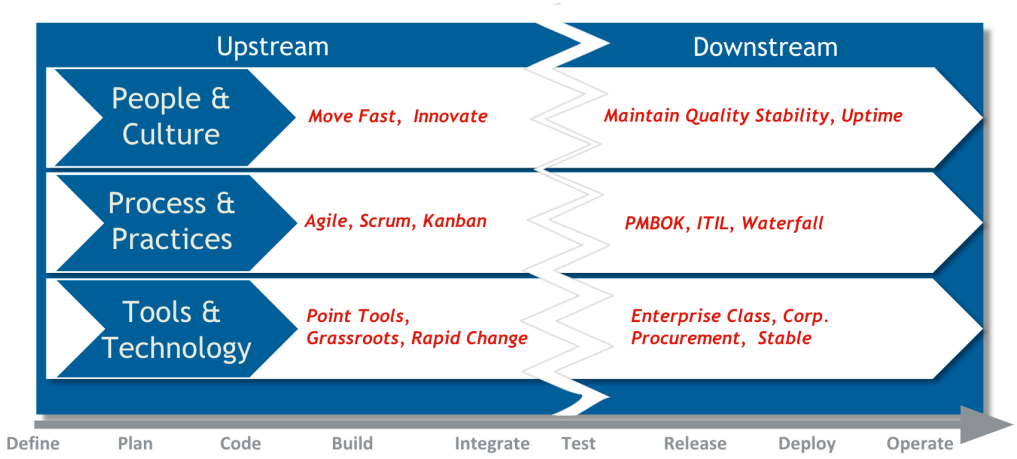

Figure 1: Bridging the chasm between upstream (development) and downstream (operations) across the three planes of the DevOps trinity

On the third plane – tools and technology – high-performing organizations have long recognized the value of using tools to automate the delivery process, as with continuous integration and continuous delivery. But these organizations eventually find that automating legacy architectures, using legacy tools and technologies will only get them so for.

Therefore, there is a growing interest among organizations in enhancing their software stack with a microservices architecture and container technology. In many ways, microservices and containers are a perfect fit for DevOps because they support most of the core concepts that lay at the heart of DevOps, including frequent builds, rapid deployment and smaller, more agile teams.

In addition to the growth of microservices and containers, organizations who have transformed to continuous delivery and a DevOps culture have also adopted the practice of versioning everything: environment configurations, hardware configurations and - of course, code. This is the only way to be sure that a development environment exactly mirrors a test environment and that it, in turn, exactly mirrors production.

In this whitepaper, we will take a closer look at several rapidly evolving trends: microservices, the expanding container ecosystem and everything as code, and explore how organizations can marry these pieces together within a DevOps culture. Together, these technologies enable organizations to deliver software at the speed of ideas without sacrificing quality, reliability or security.

Not Your Father's Architecture: Microservices Are Not Just a Mashup of SOA

From one perspective, microservices may appear to be a new version of the Service Oriented Architecture (SOA) trend from years past. Because the microservices concept has its roots in SOA the two concepts do indeed share many characteristics, but there are important distinctions that set them apart. Like SOA, microservices work by decoupling components of a complex system and defining interfaces or contracts between those components. With microservices, the communications between components tend to be lighter weight and the interfaces and contracts less rigid, often implemented through RESTful APIs. Many also view microservices as more focused on user-facing functionality rather than back-end services, but that is not a hard-and-fast rule.

Microservice components can also be deployed independently, making it easier for relatively small teams – including those with expertise in a single domain – to apply iterative processes to build, test and deliver a microservice as an individual component.

Microservices are relatively new to the industry, but they are rapidly gaining popularity and acceptance because they serve as an enabler for organizations to deliver software faster.

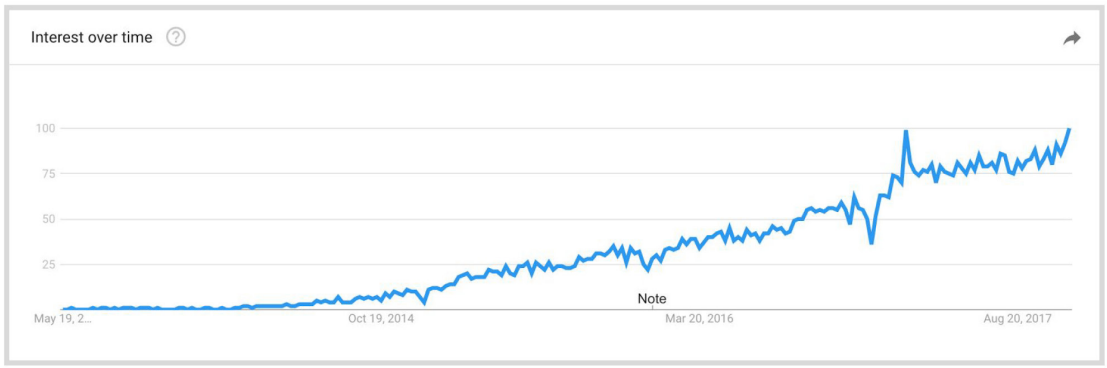

Figure 2: Google Trends shows increasing interest in microservices May 17, 2013 - November 17, 2017

Several factors contribute to this acceleration. Small components can be built independently by teams of 8-12 developers who have end-to-end control over development and delivery. Further, decoupling system functionality into smaller components makes it possible to reliably and frequently update individual components with little or no impact on the overall system. (In fact, systems designed for resiliency and based on microservices can carry on, despite the unavailability of a given service.) The decomposition of monolithic applications into microservices also facilitates easier rollbacks when issues are detected in production, as well as rolling deployments and blue-green deployments. In short, a cross-functional scrum team with development, QA and operations expertise can rapidly develop, test and deploy a complete microservice component, and then react faster to unexpected issues once it is deployed.

How Do Containers Fit In?

Much like microservices have breathed new life into some of the core concepts that animated SOA, Docker has revitalized decades-old container technology. The first Linux containers were available 15 years ago, and the technology's roots stretch back past that. Today, however, it is the Docker implementation of containers that has captured everyone's attention. Docker is now, by far, the most popular and most widely adopted container technology. By defining a standard image format and establishing a tooling and provider ecosystem, Docker made container technology widely accessible and helped bring containers into the mainstream of IT. It is difficult to find a mainstream development and delivery tool provider that has not adopted some level of Docker support.

What is the appeal of using Docker containers over separate servers or even virtual machines, which can be used in a similar way? Containers enable you to avoid building and configuring an entirely new physical server or spinning up a new virtual environment with processor emulation, an operating system and installed software. Instead, the Docker container lets you encapsulate an entire environment into a single lightweight image. As such, Docker containers provide fast access to infrastructure, a fundamental requirement of DevOps and continuous delivery practices.

Immutability is a key characteristic of the environments provided by containers As John Willis, director of ecosystem development at Docker, explains, "Immutable is a methodology where you create nothing new on a running infrastructure - generally, production...the base operating system, middleware and application are bit-for-bit identical.[2]" Because a Docker image maintains the application, all its dependencies and its environment, the entire deployment is baked in and can move through the software delivery pipeline intact. The application has a consistent environment from development through QA, staging and production. This consistency removes the risk of developers chasing a moving target.

As the industry continues to move towards the ideal of building and testing software with every change made, we need environments available on-demand to support the increased number and frequency of builds. Docker containers give us this capability and remove what used to be a huge bottleneck – waiting for environments to be provisioned.

Pulling It All Together

More than half of the participants in the 2016 Jenkins Community survey[3] said that their company is already using container technology, and more than 90% of those were using Docker. The study also showed upticks in the frequency of deployments and in deployment automation. Of those participants who were performing continuous delivery more than 40% were deploying code once or more per week and about 30% were performing fully-automated production deployments.

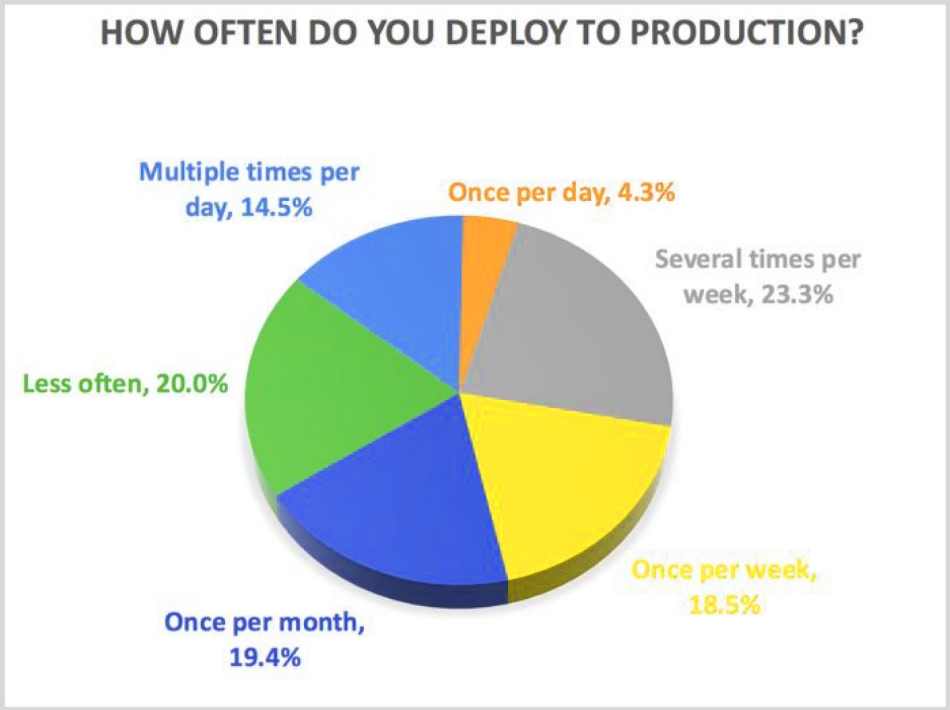

Figure 3: Responses to the *2016 State of Jenkins Community Survey*, published in January 2017

These findings correlate with the growing popularity of microservices and Docker among companies that have adopted DevOps practices. Breaking an application into discrete functional components makes it possible for small cross-functional teams to build, test and deploy them as microservices in containers.

Containers are a perfect fit for small agile teams because they provide fast access to immutable infrastructure without interfering with other development streams. Containers are also a perfect fit for microservices because they are well-suited to hosting smaller, self-contained components. Microservices and containers combined make it possible for agile teams to work more independently, procure their own infrastructure and ultimately deliver software at an accelerated pace - especially when the entire software delivery pipeline is orchestrated with Jenkins and other automation tools that support Docker.

There are, of course, some caveats to consider before jumping into microservices and containers with both feet. For one thing, container technology is maturing and evolving rapidly, and new Docker releases arrive frequently. If you're going to be aggressive about container adoption, keep vigilant about changes that may affect your specific use cases. Also, rather than spending time and resources breaking a legacy monolithic system (that works adequately) into microservices, you may want to consider leaving that software in place and use a microservices architecture only when implementing new capabilities, gradually replacing your legacy architecture.

DevOps requires a marriage of culture and process as well as tools and technology. An organization's ability to employ tools (including Docker) and technology (including containers and microservices) in support of a collaborative culture and proven practices is a leading indicator of its ability to differentiate itself by developing software more quickly, from concept to customer, and delivering that software with increased quality, reliability and security.

Everything Became Code (and Why that's Good)

The debate – if there ever was much of one – is now over. The IT industry is clearly moving to a world of everything as code. More and more, organizations are embracing the idea of modeling the components of software development and delivery programmatically in code. If they can't treat literally everything as code, they are defining everything in code as much as possible, including infrastructure, networks, delivery workflows, environments and more. Why? Because capturing these elements as code enables them to be versioned, shared, reused and refined through collaboration. In short, organizations that adhere to the everything as code concept are applying proven software development practices to other domains, enabling them to increase automation and repeatability, reduce errors and accelerate delivery pipelines.

In a world where businesses have to deliver high quality software at the speed of ideas in order to remain competitive, IT organizations are turning to practices such as continuous delivery (CD) and DevOps. These practices are focused on establishing the ability to deliver software reliably and repeatedly. We will take a detailed look at how codifying the delivery through everything as code makes the entire process repeatable, enables a DevOps approach and allows the business to realize value by getting innovation to market faster!

Applying Proven Software Development Best Practices in New Domains

Code is, at its heart, a set of text files. Over the decades, the industry has developed a wide array of tools and best practices for creating those text files as quickly as possible and with as few errors as possible. For example, virtually every development team uses GitHub or another version control tool along with common versioning practices to manage multiple variants of their code as it is developed. These tools and practices are designed wholly to enable teams to track differences in the code and to collaborate more efficiently and with greater transparency. Further, they support auditability and traceability, so teams know who changed what and when. They also support roll-backs and repeatability, enabling teams to reliably construct any version of their application from their versioned code at any time.

Everything as code extends the benefits of versioning tools and practices – as well as numerous other software development tools and practices – to virtually every other aspect of the software development and delivery process. For example, once you have captured a configuration, workflow or environment via YAML, JSON or any human-readable syntax, you can always recreate what you have captured. Using traditional software development tooling and practices, you realize the same auditability, traceability, repeatability, reuse and related benefits that you get for your application code across other software delivery domains.

What other domains can be managed using an everything as code approach? More than you might expect:

- **Networks** - With software-defined networks, teams are programmatically initializing, controlling, changing and managing network behavior dynamically via open interfaces.[1]
- **Build and deployment workflows** - Development and operations teams model software delivery pipelines in code to enable continuous delivery and speed release cycles.

```
34          stage('Checkout'){
35
36             checkout scm
37          }
38
39          stage('Test'){
40
41            env.NODE_ENV = "test"
42
43            print "Environment will be : ${env.NODE_ENV}"
44
45            sh 'node -v'
46            sh 'npm prune'
47            sh 'npm install'
48            sh 'npm test'
49
50          }
51
52          stage('Build Docker'){
53
54               sh './dockerBuild.sh'
55          }
56
57          stage('Deploy'){
58
59            echo 'Push to Repo'
60            sh './dockerPushToRepo.sh'
61
62            echo 'ssh to web server and tell it to pull new image'
63            sh 'ssh deploy@xxxxx.xxxxx.com running/xxxxxxx/dockerRun.sh'
64
65          }
```

» **Infrastructure** - Infrastructure as code using Chef, Puppet, Ansible and similar tools enable organizations to orchestrate system and environment configurations at large scale, quickly.

» **Environments** - Using container technology, developers define entire environments in code as self-contained systems. For example, Docker uses the text-based Dockerfiles to build shareable images that fit specific infrastructure requirements.

```
# Create a PostgreSQL role named ``docker`` with ``docker`` as the password and
# then create a database `docker` owned by the ``docker`` role.
# Note: here we use ``&&\`` to run commands one after the other - the ``\``
#       allows the RUN command to span multiple lines.
RUN    /etc/init.d/postgresql start &&\
    psql --command "CREATE USER docker WITH SUPERUSER PASSWORD 'docker';" &&\
    createdb -O docker docker

# Adjust PostgreSQL configuration so that remote connections to the
# database are possible.
RUN echo "host all  all    0.0.0.0/0  md5" >> /etc/postgresql/9.3/main/pg_hba.conf

# And add ``listen_addresses`` to ``/etc/postgresql/9.3/main/postgresql.conf``
RUN echo "listen_addresses='*'" >> /etc/postgresql/9.3/main/postgresql.conf

# Expose the PostgreSQL port
EXPOSE 5432

# Add VOLUMEs to allow backup of config, logs and databases
VOLUME  ["/etc/postgresql", "/var/log/postgresql", "/var/lib/postgresql"]

# Set the default command to run when starting the container
CMD ["/usr/lib/postgresql/9.3/bin/postgres", "-D", "/var/lib/postgresql/9.3/main", "-c", "config_file=/etc/postgres
```

» **Tests** - Both non-functional tests (including unit tests, performance tests and load tests) as well as functional tests (including acceptance tests and tests created for behavior-driven development) are defined and managed programmatically via a wide variety of tools and scripting languages

Harnessing the Power of a Programmatic, Normalized Representation

Aside from making it possible to reap the rewards of time-tested development tools and best practices, everything as code provides a normalized representation of the software delivery process. Normalized, means a shared view across stakeholders that defines the process clearly and unambiguously, providing insights that can otherwise be difficult to come by, as well as a mechanism for stakeholders to collaborate more effectively.

Nearly all organizations have a wide range of domain-specific tools which drive the development and delivery pipeline. Traditionally, each tool must be accessed via its own dedicated user interface (UI) in order to configure aspects of the process. In this environment, the entire process is fragmented across disparate UIs, making it difficult to achieve a common view of the delivery pipeline.

In contrast with an everything as code approach, tools are controlled and configured programmatically via application programming interfaces (APIs), SDKs and text-based representations, enabling the process to be codified into a shared, normalized representation. Such representations can be readily shared with other teams – either within the organization or outside it via GitHub – who can then provide feedback and make improvements. Defining the production environment programmatically – with the use of containers, for example – makes it possible to model the end state of the process early on, which in itself reduces errors. Any aspect of development and delivery implemented in code can be validated using tools that employ syntax checks or higher level logic. And finally, because all changes are captured and versioned, when an error is identified downstream in the delivery process, teams are equipped to immediately roll-back to a known safe version, diagnose the error and correct it.

Caveats and Coming Attractions

Of course transitioning from just code as code to everything as code will be easier for some organizations and more difficult for others, and there are considerations to bear in mind. Adopting an everything as code approach, for example, tends to shift organizations to a more collaborative culture, much like a DevOps transformation. Bringing configurations and processes out of UIs and exposing them all as code that is shared and versioned increases transparency. Agile shops may already be comfortable with this transparency and culture, but more traditional software organizations may find the initial change more jarring.

Another caveat to be taken into account is that not everyone takes to code naturally. UIs were developed for a reason – specifically to make it easier for users to complete tasks without editing files and running tools from the command line. It's true that some organizations will have a longer learning curve as they transition from UIs that guide users through steps to understanding how to implement those steps in code. Fortunately, this potential stumbling-block was recognized at the earliest stages of the everything as code movement, so all along there has been an effort to provide solutions and technologies that make the transition easier.

One example in the Jenkins space of making technologies more approachable for users of all skill levels is the Blue Ocean project, which enables teams to visually create a code-based representation of their continuous delivery pipeline, without detailed coding. The user experience provides guidance on defining CD pipelines while retaining the benefits of everything as code. In a similar way, the new Declarative Pipeline provides teams with a way to define how they want their pipelines to work using configuration files rather than scripts. The idea is that you express in a flat text file format what you want to accomplish, and then make use of tools that interpret the declarations in the file, convert them to more complex operations and then execute those operations.

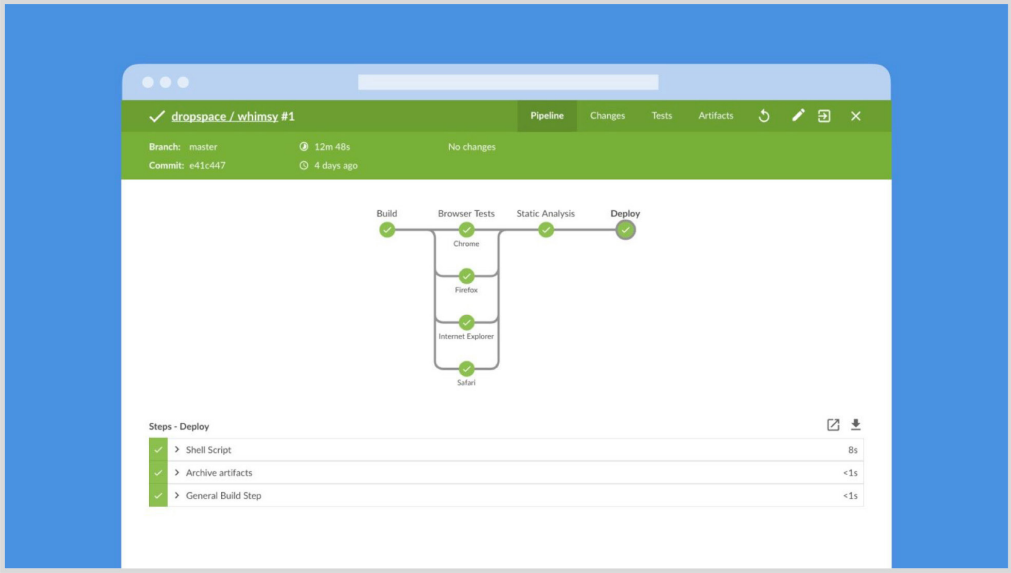

Figure 4: Example of a Jenkins Pipeline visualization using the Blue Ocean UX. Tools such as this make it easy for users of all skill levels to understand what is going on without having to delve into hundreds or thousands of lines of code.

In the coming months and years, we will likely see more of this hybrid approach that combines the best of both worlds. The various components of software development and delivery will be represented as code in their final state, but user interfaces will provide an accessible entry point to creating that code. This will ease the learning curve for everything as code, while maintaining its key advantages: more transparency and increased automation leading to fewer errors and faster deliveries.

Conclusion

In the software industry, organizations must deliver updates faster and faster. The only way to achieve speed and retain quality is to automate the software delivery process, leveraging select technologies to do so.

Organizations are enhancing their software stacks with a microservices architecture, container technologies and versioning everything as code. These technologies and practices are a perfect fit for DevOps environments because they support the core concepts that lay at the heart of DevOps, including frequent builds, rapid deployment and smaller, more agile teams. Teams that transition to continuous delivery processes and DevOps practices, while adopting these technologies see tremendous acceleration in software delivery cycles.

In addition to microservices and container adoption, organizations who have transformed to continuous delivery and a DevOps culture have also started versioning everything: environment configurations, hardware configurations and - of course, code. This is the only way to be sure that a development environment exactly mirrors a test environment and that it, in turn, exactly mirrors production. If software delivery is to be automated and accelerated, the old days of incompatible environments must be left behind.

Together, these technologies and practices enable organizations to deliver software at the speed of ideas without sacrificing quality, reliability or security.

[1] *2016 State of DevOps Report*, Puppet website
[2] *DevOps and Immutable Structure*, a presentation given at Cloud Expo 2015 by John Willis, Docker
[3] 2016 State of Jenkins Community Survey, September 2016

Learn More

Visit the CloudBees DevOps Resource Center
www.cloudbees.com/devops

Read the Whitepaper
Assessing DevOps Maturity Using a Quadrant Model

Read Case Studies and See How Real-World Enterprises Are Using DevOps to Transform Software Delivery
www.cloudbees.com/customers

Learn more

www.cloudbees.com/products

CloudBees CI is built on top of Jenkins, an independent community project. Read more about Jenkins at: www.cloudbees.com/jenkins/about

1020v03

CloudBees, Inc.
4 North Second Street | Suite 1270
San Jose, CA 95113
United States
www.cloudbees.com
info@cloudbees.com