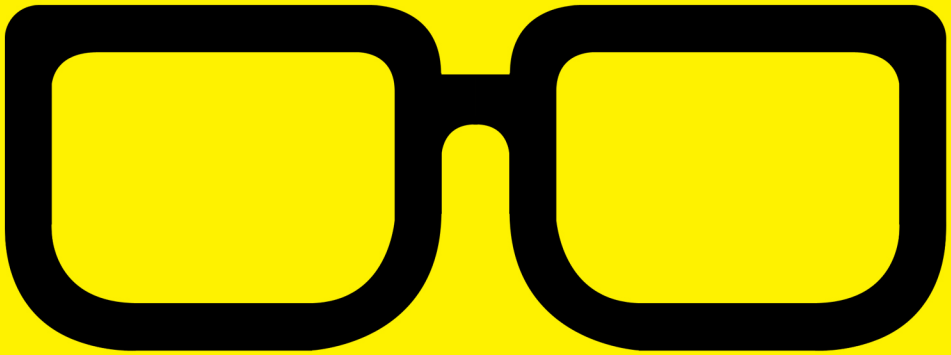
SPONSORED BY

puppet

GEEK GUIDE



DevOps for the Rest of Us

LINUX™ JOURNAL

Table of Contents

JOHN S. TONELLO is the Director of IT and Communications Manager for NYSERNet, New York's regional optical networking company, serving the state's colleges, universities and research centers. He's been a Linux user and enthusiast since building his first Slackware system from diskette more than 20 years ago.

GEEK GUIDES:
Mission-critical information for the most technical people on the planet.

About the Sponsor

Puppet

Puppet is driving the movement to a world of unconstrained software change. Its revolutionary platform is the industry standard for automating the delivery and operation of the software that powers everything around us. More than 33,000 companies—including more than 75 percent of the Fortune 100—use Puppet's open source and commercial solutions to adopt DevOps practices, achieve situational awareness and drive software change with confidence. Based in Portland, Oregon, Puppet is a privately held company with more than 520 employees around the world. Learn more at http://puppet.com.

DevOps for the Rest of Us

JOHN S. TONELLO

Introduction

No matter what industry you're in, you probably find yourself on a perennial quest to build, modify, test and release software rapidly, frequently and more reliably—without driving everyone loony in the process. That's the aim of organizations large and small, all of which have long sought ways to bring developers and IT folks closer together and benefit from better collaboration and communication.

These efforts have taken many forms and names over the years, and chances are, you've already taken steps to simplify and standardize the way you build and deploy things, whether it's built in-house or bought off the shelf.

If so, you're already doing some sort of DevOps—even if that's not what you call it.

The benefits are many, but with the rise of the term DevOps has come the rise of confusion and fear. The full-court press on your inbox isn't helping. If you're like most, you just want ways to improve what you do continuously, convince your bosses and C-level executives that you can make changes without blowing up your existing operations, do away with boring routine, and maybe make your team members and customers happier—and more willing to stick around.

For some, part of the answer is an automation tool like Puppet, a straightforward open-source tool that enables you to start simply and grow, keep all your legacy systems and development tools in place, retire a bunch of shell scripts and manual tasks, and quietly become more efficient, predictable and productive—that is, more "DevOps-y".

This Geek Guide describes practical ways to use Puppet across all your platforms—new and legacy—and deploy a few manifests and modules that will help you automate configuration management, improve security, fortify your team and convince the higher-ups that DevOps is a lot closer and achievable than they might think.

Who's Skeptical of DevOps?

It's becoming harder to avoid mention of DevOps anywhere you go these days. It's talked about at conferences, in blogs, job postings and everywhere IT practitioners live and work. As a result, some shrug it off as the latest fad.

Wait long enough, and it will pass. But unlike true fads, this one has been adopted in ways small and large by companies and organizations of all sizes, many of which are likely to be your customers and competitors.

Still, skepticism is understandable, particularly in organizations that have invested time and money to build, modify and deploy what they now have. After all, as a going concern, you've done pretty well on your own, right? You've earned a right to be doubtful and your bottom line bears you out.

Skepticism is also understandable among those who've been burned by organizational fads in the past. Anyone who remembers the push for Total Quality Management in the late 1980s will recall that, as a management practice, it was easy to wind up with well-planned garbage. It was the ultimate garbage in, garbage out management philosophy, because organizations, anxious to deliver customer-defined quality, easily could misinterpret the TQM principles or falter when they struggled to implement them.

Perhaps less credible are the DevOps doubters who have invested a lot in maintaining the status quo. These are the people in your organization who are quick to say, "we've always done it this way", or recoil from any kind of change. These engineers and IT practitioners have time on their side. Momentum is with them, particularly if they manage a wide array of legacy systems with an experienced, legacy staff.

So, who's skeptical of DevOps? In a word, just about everyone—who hasn't tried it.

But you don't need a "DevOps Initiative" to do DevOps, because at its core, DevOps is fundamentally about communicating early and often across teams.

DevOps by Any Other Name

A certain amount of skepticism is healthy, but when it comes to DevOps, a lot of the skepticism may be rooted in the name itself. It's vague and doesn't mean the same thing to everyone. But you don't need a "DevOps Initiative" to do DevOps, because at its core, DevOps is fundamentally about communicating early and often across teams. If you're already doing that—or working toward it—you're already improving how you work, even if you don't call it DevOps.

For example, think about how your IT operations team currently collaborates with your security, network and developer teams. Do you follow the same company-wide standards? Do you adhere to common organizational goals of customer service? You probably can answer with an easy, "Yes."

Harder to answer are questions like the following:

- How do you share information?
- How do you devise company-wide standards?
- How do you respond to security breaches or system outages?

- How do you collaborate?
- Is the IT team a bottleneck or seen as the department of the perpetual "No"?
- What else prevents you from working with and trusting other teams?

Answers to all those questions impact how you get things done—or don't. Collaboration can be especially tricky between operations and developer teams because they have opposing priorities. IT's goal is to create and maintain reliable and secure systems. The developers need to be fast-moving, innovative and cutting-edge. This friction may be masked in scorn as one team fulfills its bit, tosses the task over the wall to the next team and moves on. Maybe the tools you use don't mix and match well, and that limits even well intended collaboration.

Less typical are teams that sit down regularly and hash things out, even though most know that baking in security, measurement and deployment requirements is best done up-front, not at the end of each step when it's too late or too costly to do anything about it.

At the end of the day, how well your organization works—and how well it delivers products and services—depends on how well you're positioned to share information. Data-sharing was one of the best bits of TQM and ultimately became the base for successful Lean Management, which itself took on the shape of DevOps when it was coupled with Agile methods. That evolution

If you've ever tried to devise your own common platform to communicate or share information across teams, you know it's no small task.

requires buy-in, visibility and communication, which you can't just pull out of a hat and wish it to be so.

Start by Establishing a Universal Language

If you've ever tried to devise your own common platform to communicate or share information across teams, you know it's no small task. Ticketing and project management tools—sometimes known as ChatOps—can help, but they go only so far. They're certainly useful, so long as they encourage interaction and don't replace it.

Though initially conceived as an automation tool, Puppet and Puppet code can be an effective way to establish a common language among your operations, developer and security teams, because it's essentially executable documentation. Instead of documenting the steps you take to run other tools and scripts, you could just run that documentation and automatically bring up systems that match your needs to the letter.

The key to Puppet's capabilities is the Puppet domain-specific language (DSL), which is easy to read, understand and share. The code describes the desired state of a resource, whether it's running on one node or a thousand. This is a declarative approach; you declare the configurations you want

and Puppet maintains them. Compare that with an imperative approach, which doesn't keep your systems in a consistent state. Once you make an imperative change, the system is no longer the same. That can make for some complicated if…then loops in your bash scripts or simply break them.

Want all your Linux machines to have the same base firewall rules and your web servers to have something else? Define the rules once in a few lines of Puppet code, and Puppet makes sure each node is in compliance. You don't need to tell Puppet how to execute something; you just tell Puppet what state you want your infrastructure and applications to be in, and Puppet does it for you. This can be as simple as making sure OpenSSH is enabled on every server or as sophisticated as deploying fully containerized clusters.

At its heart, Puppet is open source and easy to download and install. Most of what I reference in this Geek Guide can be accomplished with open-source Puppet, but if you want to take it a step further, you can deploy Puppet Enterprise, which features a browser-based graphical interface and some handy visual tools for keeping track of things, plus out-of-the-box automated workflows. It's also free to download and use for up to 10 nodes.

Regardless of how you get started, Puppet can manage packages, services, files, Dockerfiles, users and a wide array of system parameters and settings. For example, to enable SSH on your Linux servers, this bit of code will do it for you:

```
package { 'openssh':
        ensure          => present,
}
```

To create a user, you could do something like this:

```
user { 'username':
        ensure          => present,
        home            => '/home/username',
        shell           => '/bin/bash',
        managehome      => true,
        gid             => 'username',
        password        => '$1$zi13KdCr$zJvdWm5h552P8b34AjxO11'
   }
```

You don't have to tell Puppet how to do something, just that you want it done. It figures out how on each OS, including CentOS, Debian, Red Hat, SUSE, Ubuntu and Windows. Compare this with a non-Puppet way, which would require you to create different scripts for each OS. There are enough differences between Red Hat and Debian to require separate directory paths, for example, and Windows is another animal entirely. Puppet abstracts away the differences and enables you to use one common manifest, not three different ones to manage your resources.

Most people deploy Puppet with a master, which communicates with Puppet agents on other Linux and Windows machines, VMs or containers. With a master, you can deploy common standards everywhere you want them and nowhere you don't. You also can run Puppet on a single machine without a master, but this is more typical of a testing environment than development or production.

Puppet code is built with key-value pairs, and you can bundle your directives into classes, which can form modules that can be

shared easily. You also can take advantage of thousands of existing modules on the Puppet Forge, such as puppetlabs/firewall, a snippet of which is shown here to describe some firewall rules:

```
class my_fw::pre {
 Firewall {
   require          => undef,
 }
  # Default firewall rules
 firewall { '000 accept all icmp':
   proto            => 'icmp',
   action           => 'accept',
 }->
 firewall { '001 accept all to lo interface':
   proto            => 'all',
   iniface          => 'lo',
   action           => 'accept',
 }->
 firewall { '002 reject local traffic not on loopback interface':
   iniface          => '! lo',
   proto            => 'all',
   destination      => '127.0.0.1/8',
   action           => 'reject',
 }->
 firewall { '003 accept related established rules':
   proto            => 'all',
   state            => ['RELATED', 'ESTABLISHED'],
   action           => 'accept',
 }
}
```

One of the advantages of Puppet code is that it's easy to read, which makes it a good option as a cross-team universal language.

One of the advantages of Puppet code is that it's easy to read, which makes it a good option as a cross-team universal language. You don't have to be a firewall expert to understand the result of the above manifest. Each firewall rule is expressed with a title and values that are straightforward and familiar. As a result, you can share it with other teams readily without having to include a lot of extra explanation. It is executable documentation, and the DSL code is concise directives that you might otherwise articulate in runbooks or on a staff wiki.

Sharing Because You Can

If you're looking for a way to break the ice with other teams, you might sit down with some of their more enthusiastic members and identify resources you'd like to standardize and automate. It doesn't have to be complicated, but by working together to write a few sample manifests, you'll start to recognize other ways you can collaborate, gather feedback and simplify moving from development to production. As you become more expert, you can help other teams identify what they'd like to standardize and automate.

Of course, nobody likes to be told they need to change

the way they do things. They are, however, often open to learning how *you* changed the way *you* do things, and how those changes might impact what they do. So, after you've changed how you do some things, approach other teams and share with them the problems you were facing, how you addressed them, and what the results were—especially the part about how it made your life better. Then, ask how what you did might affect what *they* do, and offer to collaborate on tackling any problems they might decide need tackling.

Getting a Taste with a Few Sample Cases

You can create Puppet manifests to accomplish a wide range of tasks, from deploying Apache and Docker to enforcing user and group rules. That means everyone can find ways to use it, particularly if they browse the nearly 5,000 existing modules in the Puppet Forge.

Whether you're using open-source Puppet or Puppet Enterprise, you'll discover ways to deploy resources confidently that can make it far easier to make development environments match production environments. In fact, you can use a wide range of conditionals and system-specific facts to extend and customize your Puppet code and classes further. This can be useful in software too, regardless of whether your team wrote it from scratch or modified an off-the-shelf solution.

Classes—not to be confused with classes in a programming sense—are the preferred Puppet way of defining resources for use in your manifests. They're almost like functions found in traditional

programming—snippets that can be maintained in one place and used over and over again.

For example, the following apache class defines an httpd package, an httpd.conf file and the httpd service, and parameters to make it work. This will install the latest version, make sure the configuration file exists and run it with an Apache template:

```
class apache (String $version = 'latest') {
 package {'httpd':
   ensure => $version, # Using the class parameter from above
   before => File['/etc/httpd.conf'],
 }
 file {'/etc/httpd.conf':
   ensure  => file,
   owner   => 'httpd',
   content => template('apache/httpd.conf.erb'), # Template from a module
 }
 service {'httpd':
   ensure    => running,
   enable    => true,
   subscribe => File['/etc/httpd.conf'],
 }
}
```

By adding conditionals, this manifest could be modified to install the proper configuration files for each OS. When combined with the system information-gathering tool Facter from Puppet, you can enable your classes and manifests to work nicely across platforms.

Facter returns facts about any host—everything from its hostname and OS family to IP address and available memory. Puppet can gather and use these facts in real time, giving broad visibility into your systems that can be shared and acted upon.

Facter returns facts about any host—everything from its hostname and OS family to IP address and available memory. Puppet can gather and use these facts in real time, giving broad visibility into your systems that can be shared and acted upon.

Here's a bit of output from Facter, executed on a Linux virtual machine:

```
$ facter
os => {
 architecture =>[a] "amd64",
 distro => {
   codename    => "trusty",
   description => "Ubuntu 14.04.5 LTS",
   id => "Ubuntu",
   release => {
     full  => "14.04",
     major => "14.04"
   }
 },
```

Here's the same output reported by Facter executed on a Windows machine:

```
os => {
 architecture => "x64",
 family       => "windows",
 hardware     => "x86_64",
 name         => "windows",
 release => {
   full  => "7",
   major => "7"
 },
 windows    => {
   system32 => "C:\Windows\system32"
 }
}
```

These values can be captured in Puppet code as `$facts[]`, and the following example shows how you might put this into practice to install ntp, the network time package for Linux servers, which doesn't happen to work on Mac or Windows machines:

```
if ($facts['os']['name'] == 'Darwin') or ($facts['os']['name']
 ➥=='windows') {
 warning('This NTP module does not yet work on our Macs and
 ➥Windows boxes.')
}
else {
 # Normal node, include the class.
```

```
 include ntp
}
```

For the operations team, you could use regular expressions to define types of nodes to manage, such as the names of Linux nodes, to deploy common packages or services more quickly. If you want certain rules to apply only to web hosts, which you've given names like web1, web2, web3 and webN, you can apply rules in a single manifest description instead of listing each node separately. So, instead of writing this:

```
node 'web1.example.com', 'web2.example.com', 'web3.example.com' {
 include common
 include apache, squid
}
```

or this:

```
node 'web1.example.com' {
 include common
 include apache
 include squid
}

node 'web2.example.com' {
 include common
 include apache
 include squid
}
```

```
node 'web3.example.com' {
 include common
 include apache
 include squid
}
```

you could just write this to cover all nodes that begin with "web" followed by one or more digits:

```
node /^web\d+$/ {
 include common, squid, apache
}
```

Now, each time you deploy a new web server for the developers, you simply have to name the host in this way for Puppet to install everything in a consistent, reportable way that meets your standards and serves the needs of your Ops customers.

Using Puppet to Establish Standards

For anyone who has ever managed systems and the types of settings described above, there's a certain joy in letting Puppet do the work. As your nodes check in every 30 minutes, they get updated automatically. If they don't align with your manifests, they're corrected. If they do align, they're left alone.

You probably can begin to imagine how this core Puppet functionality can be useful beyond the IT shop and across the organization as a way to create and

enforce baseline system configurations collaboratively. If your security team has complex security rules that vary from platform to platform or from on-premises to off, you can begin to sketch out, or model, those rules as manifests that guarantee they're applied every time.

For example, say the security team wants you to close port 22 on every Linux VM host you have in your Amazon Web Services S3 cluster, but you want the port open for everything in your own data center, particularly everything on a local subnet. If you've given each VM a handy hostname prefix like on-www.example.com or off-www.example.com, you could set up a conditional statement in a manifest that sets the rules appropriately. Chances are, though, you haven't named everything like this, and instead you use VM templates to make sure the host configurations are appropriate to each setting.

With Facter, though, you can build your security manifests around available system facts, such as `is_virtual`, `domain` or `ipaddress`. If the IP address of everything in your Amazon cloud begins with 172.128, you could use the built-in variable `ipaddress` as your trigger:

```
if $::ipaddress =~ /^172.128/  {
        firewall { '000 drop ssh':
                port            => '22',
                proto           => 'all',
                action          => 'drop',
        }
}
```

```
elseif $::ipaddress =~ /^10.128/  {
        firewall { '001 accept ssh':
                port            => '22',
                proto           => 'tcp',
                action          => 'accept',
        }
}
else {
        # Take no action on hosts not defined by the above
        fail("No rule for ssh has been defined for $::ipaddress.")
}
```

This is both actionable Puppet code and documentation you can share and extend. You could add more conditionals to detect and act on the OS of each system you want to secure, and you could translate the security team's rules into manifests that guarantee the desired state on existing hosts and any new ones. In the example above, that means any new VM in your 172.128 scope has SSH access blocked by default.

From a DevOps perspective, this solves three problems in one fell swoop. First, you can engage your security team up front in defining the manifests. Second, you can automate the enforcement of the rules everywhere reliably, and third, you can provide feedback in the form of real-time system audits that can help you collaboratively develop and deploy new rules.

In a similar way, you could create Puppet manifests to define users and groups consistently, something that's useful across every team. Again, if you and the security team want to add certain users to the on-site VMs that are

different from the off-site ones, you can define user classes and include those in your manifests. And because Puppet code lets you use password hashes—the same found in /etc/shadow—you can define all your users on all your VMs—on-premises and off—with a few lines of clear, sharable code.

From an operations standpoint, this also means you can get out of the business of manually adding users to each new machine or removing them later. That free time may enable you to tackle back-burnered projects sooner, beef up your skills or simply take your time over lunch for a change.

Create and Maintain Baseline Configurations

You can begin to see how you can use this process among your organization's teams to establish baselines for security, OS configurations, users, packages, services and even files. Instead of using custom (and perhaps separate) scripts to do this work, you could begin to standardize using Puppet code. Compare this with what you're probably doing now, such as creating "golden images" or a proliferation of custom bash scripts and tools that aren't standardized.

With Puppet, you don't suddenly have to retire what you're currently doing; you can start small—perhaps just setting up a few manifests that establish users and set firewall rules or even set a common time server on all your Linux hosts. As you become more comfortable, you can begin to phase out some of the custom stuff and grow your baseline Puppet manifests. At the same time, you and your Ops team will spend less time doing low-value rote tasks and spend more time on high-value work.

Standards Across Platforms

If you're like most, you don't have just one kind of server or system to manage, you have a variety. Some data centers may even look a bit like the inside of a Jawa Sandcrawler, filled with machines old and new. You may have new blades running the latest Windows, Linux, KVM, VMware or Hyper-V, and a handful of towers or older stuff that you just can't get rid of. Keeping it all running is hard enough, let alone trying to superimpose some sort of baseline or standard across everything.

Again, you may have different teams tending different resources, whether they're Windows or Linux, legacy or new, on-site or cloud. If you have separate Linux and Windows groups, you can come together and talk about ways to automate and standardize common actions, using Puppet as a way to more readily understand each other's needs and goals.

For example, the puppetlabs/windows module contains a number of tools you can use to manage Windows boxes, from access control to Windows Server Update Service (WSUS). It can interact with PowerShell, manage registry keys, build IIS sites and virtual applications, manipulate environmental variables and manage the installation of software with puppetlabs-chocolatey, a sort of apt tool for Windows.

This module runs under Puppet Enterprise, and it's a good way to become comfortable with Puppet while bringing separate Linux and Windows teams closer together, something that's never easy even in the best of times. You won't solve their differences overnight, but you can get them talking for a change.

Use Puppet to Deploy Development Environments

If you're not quite ready to reach out beyond your team, but you want to start finding ways to more confidently collaborate, you might consider using Puppet to build and deploy a development environment. With automation, you then can replicate it as a fully functional production environment—or a model of one you can share.

For example, you might set up a single Puppet master that your teams share, but deploy separate nodes for web and database servers. For Apache, you could deploy the puppetlabs/apache module, which uses a single manifest to install the default configuration appropriate to your operating system, applying unique defaults for Debian, Red Hat, FreeBSD and Gentoo. You can quickly set up virtual hosts too, with or without certs.

Adding a virtual host user.example.com can be as straightforward as this:

```
apache::vhost { 'user.example.com':
 port           => '80',
 docroot        => '/var/www/user',
 docroot_owner  => 'www-data',
 docroot_group  => 'www-data',
}
```

In one small block of code, you can enable this website to run on port 80 and set ownership of the document directory. To deploy SSL and non-SSL sites simultaneously,

you might do something like this:

```
# The non-ssl virtual host
apache::vhost { 'mix.example.com non-ssl':
 servername => 'mix.example.com',
 port       => '80',
 docroot    => '/var/www/mix',
}

# The SSL virtual host at the same domain
apache::vhost { 'mix.example.com ssl':
 servername => 'mix.example.com',
 port       => '443',
 docroot    => '/var/www/mix',
 ssl        => true,
}
```

When you spin up new nodes, you can use their unique hostnames and IP addresses (drawn from Facter) to deploy web servers. Check the Resources section at the end of this ebook for details on how to do this and a wide range of customizations. From a development standpoint, this is a clean, fast way to deploy new websites on nodes. When everything works the way you want, you can use the same Puppet manifests to deploy to production.

From a DevOps perspective, building basic rules for deploying Apache (or other services and packages) will save a lot of time and give developers a shorter, clearer path to development environments. Instead of

Instead of cloning a VM and then modifying it to accommodate unique virtual hosts, which can be a trick to keep straight, you can use Puppet to define the parameters and the nodes in clear, readable and shareable code.

cloning a VM and then modifying it to accommodate unique virtual hosts, which can be a trick to keep straight, you can use Puppet to define the parameters and the nodes in clear, readable and shareable code.

At the same time, with vRealize Orchestrator and Puppet's plugin for it, you can automate and control release cycles of entire application suites—even complex ones—from dev to test to production. This sort of app-level automation will lead to increased agility for any IT team.

If you want to install, configure and manage MySQL, MariaDB, MongoDB or another database, you can standardize that too. Instead of manually creating users or default databases and permissions, define them with Puppet code to confidently—and quickly—create a development environment that easily can be deployed in production and suit the needs of developers.

Bringing It All Together with Containers

If you're working with Docker and containers—or want to—there are some great resources for deploying a complete

working environment with Docker Compose. Again, this is a perfect way to break the ice with other teams and work together on new solutions.

The Puppet modules garethr/docker and puppetlabs/docker_platform are a great place to start. Together, these modules have been downloaded from the Puppet Forge nearly two million times because they provide a fast, nearly painless way to deploy Docker. Adding Docker Compose enables you to build out entire well-defined container clusters.

Docker Compose uses YAML files to describe a set of containers, and those files build and run those containers automatically. The puppetlabs/puppet-in-docker-examples docker-compose.yml file will deploy a fully functional Puppet master and a handful of nodes, including puppetboard and puppetexplorer, two dashboard components. You'll end up with an environment that's perfect for Puppet experimentation. If you mess up something, you can deploy a whole new containerized Puppet environment in a few minutes—and so can other teams. It's a great way to try Puppet with zero risk and very few resources. In fact, you can spin up the whole thing on a modest Linux VM.

Environments on Demand

Whether or not you're using containers, you can begin to see how Puppet can help move your organization closer to trusted self-provisioning, the Holy Grail for developers using DevOps.

If you ask developers about DevOps, they're the ones to argue that it falls down because they still have to wait for operations to give them servers. Everything Ops has done to

improve speed, communication and visibility up to this point can easily bottleneck here, because developers often just don't want to—or feel they can't—wait for operations to do its thing.

Despite that, operations and security teams don't want to just hand over the power of provisioning to developers. There's simply too much risk, including the risk of losing track of everything out there.

But, even trusted Ops-built machines aren't risk-free. Maybe someone testing a new app will turn off firewall rules and forget to turn them back on. Maybe they'll use a quick password that's a hacking risk.

Part of the fear Ops teams hold for self-provisioning is what happens after a machine is rolled out. Something created for a quick, day-long test isn't the issue; it's the machine that's spun up and left running long after it's needed.

By establishing baseline Puppet manifests, you can ensure that each machine adheres to specific standards. If those standards are amended, you don't have to hunt down each node. Puppet will make the updates and changes automatically everywhere. If developers need new parameters on all their MongoDB nodes, you can roll them out by modifying a single manifest. This adds speed and confidence on both sides, a key to thinking in a DevOps way.

By working together to define Puppet manifests, you can move much closer to self-provisioning, because you can define how firewalls, packages, services, networking and patches are deployed and maintained. If you're running Puppet Enterprise, you can discover your infrastructure automatically—on- and off-site VMs and bare metal—and provision machines based on the policies you define. Instead

of building images and running manual scripts, you'll have more control over the entire process and more confidence to enable on-demand nodes.

If you're a VMware shop, check out Puppet and VMware vRealize Automation, a way to put together a complete self-provisioning solution. Read more in the Resources section.

Conclusion

DevOps is about changing how you think about work first and changing how you work second. Organizations that create software—or customize off-the-shelf software to suit their needs—have evolved to do that task as well as possible, and there's a lot at stake when contemplating even the smallest change. But before you set out to change what you do, you have to think about how you do it. Without that self-reflection, you won't convince your team, your peers, your bosses or yourself that it will be sustainable.

Puppet can give you a place to start the conversation and think anew about the work you're already doing. Puppet helps because it's not one-size-fits-all, but a unique tool that can be used across teams and within them. It doesn't require anyone to shelve tools they like and trust, but it works alongside them to make them better. Regardless of what you call the process, it can help you collaborate and build clearer understanding of functions across teams, which will make for better outcomes.

At the end of the day, your teams will have new ways to work together to solve shared problems, and that will lead to faster and better deployments, greater agility to handle change and a greater ability to position your organization

and company to meet the next wave of challenges. You'll also have more fun doing it, and that's great for all of us.■

Resources

Puppet Firewall Module: https://forge.puppet.com/puppetlabs/firewall and https://docs.puppet.com/pe/latest/quick_start_firewall.html

Puppet Tools: https://www.puppet.com/puppet-tools

Writing Puppet Manifests: https://www.digitalocean.com/community/tutorials/configuration-management-101-writing-puppet-manifests

Regular Expressions: https://docs.puppet.com/puppet/4.9/lang_node_definitions.html

Puppet Classes: https://docs.puppet.com/puppet/4.9/lang_classes.html

Puppet Conditionals: https://docs.puppet.com/puppet/4.9/lang_conditional.html

Puppet on Windows: https://forge.puppet.com/puppetlabs/windows

Puppet Enterprise: https://puppet.com/product

Docker Compose: https://docs.docker.com/compose and https://puppet.com/blog/docker-compose-and-docker-network-support-puppet

Puppet Apache: https://forge.puppet.com/puppetlabs/apache

Puppet MySQL: https://forge.puppet.com/puppetlabs/mysql

VMware vRealize: http://www.vmware.com/products/vrealize-suite.html and https://puppet.com/blog/a-guide-to-puppet-integration-vmware-vrealize-automation