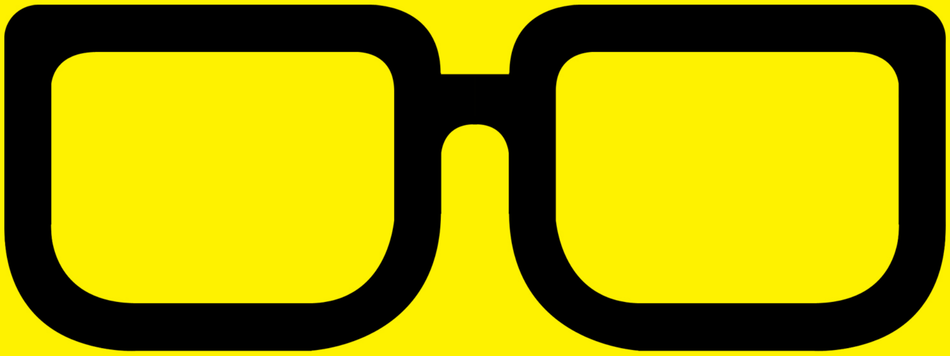
SPONSORED BY

puppet labs

GEEK GUIDE



Combating Infrastructure Sprawl

LINUX™ JOURNAL

Table of Contents

BILL CHILDERS is the Senior Development Operations Manager for MobileIron, a mobile device management company. Bill has worked in IT and DevOps since before the DevOps term was coined, and he has performed a variety of roles in software organizations: systems administrator, technical support engineer, lab manager, IT Manager and Director of Operations. Bill co-authored *Ubuntu Hacks* (O'Reilly and Associates, 2006), and he has been Virtual Editor of *Linux Journal* since 2009. He's spoken at conferences, such as Penguicon and LinuxWorld, and is enthusiastic about DevOps, IT and open source. He blogs at http://wildbill.nulldevice.net and can be found on Twitter at @wildbill.

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

About the Sponsor

Puppet Labs

Puppet Labs, Inc. is the leader in IT automation. Puppet Labs software provides system administrators the operational agility, efficiency and insight they need to proactively manage dynamic infrastructure, scaling from tens of servers to thousands, both on premise and in the cloud. Thousands of the world's leading organizations use Puppet Labs software to configure and manage their IT infrastructure, including Bank of America, Cisco, NYSE, Salesforce and WebEx. To learn more, please visit http://puppetlabs.com.

Combating Infrastructure Sprawl

BILL CHILDERS

Introduction

Do you have an environment in your company that seems to be a time sink and management headache for you? Does this environment have many machines, each running a slightly different patch level or completely different operating system? Do you find yourself and your team stuck in a quagmire of varying point releases, RPM conflicts and dependency hell? Your environment may be suffering from infrastructure sprawl, but relief is on the way. This guide will help you understand infrastructure sprawl, how it occurs and steps you can follow to eliminate (and ultimately prevent) it.

What Is Infrastructure Sprawl?

Merriam-Webster Dictionary gives one of the definitions of "sprawl" as the following: *"a group of things (such as buildings) that cover an area in an uneven and ugly way."* A common use of this word is in the phrase "urban sprawl," where a city's expansion seems to grow unplanned and unbounded (a great example of this is the Los Angeles City Basin).

FIGURE 1. Don't let this become your data center.

Applied to server infrastructure, the word "sprawl" denotes an operational nightmare for the folks who are tasked with the responsibility of keeping those systems available and healthy. Sprawl happens organically, and in an unplanned, disorganized environment, it can become the natural state of the environment.

At one point in my career, I inherited an environment that suffered from a bad case of infrastructure sprawl. This environment spread across several hundred servers from three different hardware vendors, two different major versions of CentOS Linux and seven different minor versions. There was no configuration management scheme, no regular patching schedule, and all the servers had unique and different configurations as they had all been hand built by different people at different times, with no documentation or process. The simplest tasks became a very large pain point over time, due to the lack of ongoing attention and organization.

As an example, I needed to install the net-snmp package to each of the machines so I could collect monitoring data from each system using SNMP (Simple Network Management Protocol). Typically, you just install the net-snmp package with a simple `yum install net-snmp`, and then configure the package via a configuration file and start the service. In this case though, some machines were far out of date on updates, and due to a semi-broken yum configuration, net-snmp refused to install on approximately half of the servers. These machines were in such poor shape, it cost a lot of time and effort, and completely slowed down our team.

Through the use of process, automation and infrastructure-as-code concepts — and by using configuration management tools — my team and I were able to bring order and sanity to this environment. Users were pleased because the environment's availability improved, and my team was happier because the nightmare had been tamed through our hard work.

Now that you know what infrastructure sprawl is and have heard a story about it, let's talk about tools and techniques you can use to fight it. The best way to fight sprawl is to keep it from occurring in the first place. That environment I mentioned above was not particularly large or complex, but it wound up being extremely time consuming and difficult to keep going, simply because of a lack of planning and appropriate processes and tools. If you stay on top of your infrastructure and use tools and processes appropriately, you limit the opportunity for sprawl to gain a foothold in your environment.

Process

Process has been defined as *"a systematic series of actions directed to some end."* I prefer to think of process with a DevOps or system administrator's mindset: *"Process is nothing more than a script or program that's run in meatspace rather than in software."* Process never should be feared or fought against, as it's a tool in your toolbox that will make ongoing maintenance of a given environment easier, more predictable and more methodical.

As you start to design your process, take care that you are thinking of your business policies as well. Your process

When you implement process into your environment, you will bring consistency and repeatability to your operations.

is probably not going to last long if it bypasses corporate security policies or provides a way for a developer to launch an unlimited amount of cloud servers. Any process you create should act as a blueprint for how you want your operations to run, but it also should act as a guide for what not to do. This way you can make sure that when you build your automation on top of your process, you are thinking about what you want to happen as well as what you want to avoid.

Repeatability: When you implement process into your environment, you will bring consistency and repeatability to your operations. I've met engineers who fear process and fight it every step of the way. This is generally due to a lack of understanding and a fear that the process will wind up being inefficient or inflexible. Your process never should get in the way of getting things done — it should be your roadmap, illuminating the path toward your end goal. If the process becomes inefficient, bloated or for some reason stops being what your team needs, the process needs to be revised or completely reworked. You can avoid this to some extent by doing a lot of due diligence and planning before you implement your process. If you do all the work up front, you can ensure that your process is something your organization can live with in the long term. When planning your process, always do so with an eye toward embedding quality into it — and doing quality assurance checks — throughout the process, so errors are caught early, and you "fail fast."

Once you get a particular task or workflow to be repeatable — so that it's on "autopilot" so to speak — that task tends to take much less time to complete. The faster you and your team can move work from the "to-do" pile to the "done" pile, the more time you'll have to work on more awesome things that can help the business move faster, as well as showcase your team's abilities to the larger organization. The key here is to get the mundane, repeatable tasks out of the way so that you and your team can concentrate on the really interesting (and fun) work that moves the business forward.

Recently, my team completed a company-wide dashboard display that rotates through various statistics around what the business is doing. That dashboard has since been picked up by other business units and is being displayed on TVs throughout the company — and it resulted in a shout-out for us at the company all-hands meeting. We were able to take that on because we had the time to do it, thanks in large part to our process, which makes sure that everyday tasks are repeatable.

As an example of what can happen when you don't follow process, let me refer back to the environment I mentioned earlier. Once we got the environment under control and had it running smoothly for a little while, a decision was made to hand off that environment to a different team. We performed a handoff that included some training and documentation, and we made sure that the

Your process is what will allow you to start thinking of servers as nearly disposable objects, rather than custom-crafted computers.

team taking stewardship of this environment knew how the configuration management system worked and how to maintain it going forward. However, the new owners of the environment decided that the overhead of maintaining the configuration management system "wasn't worth their time," so they simply turned it all off. Not long after that, unplanned configuration changes began to happen, and each server slowly started varying from its neighbor servers in small but significant ways. A year later, the environment ended up in much the same state as when I first inherited it — sprawl had taken a foothold and no one did anything to stop it. As a result, my team and I got to straighten out that environment a second time. The lesson learned here is once you establish a process, stick to it. It's okay to change the process as your environment changes and grows — just make sure you continue to follow it. Having nothing in place is a sure-fire recipe for entropy (and sprawl) to take hold and undo all of your hard work.

Scalability: Having a solid process is a key part of being able to scale how many servers you can manage effectively. Your process is what will allow you to start thinking of servers as nearly disposable objects, rather than custom-crafted computers. Some people say you should think of servers as "cattle, not pets." What is meant by this is that instead of thinking of a server as a single, unique entity with a name that your team manages, it should be a member of a fleet of identical machines where everything about those machines is the same. The analogy boils down to this: Pets are loved. Pets have names. Pets are unique. When pets get sick, you nurse them back to health. Cattle are given numbers and are considered identical to other cattle. They are not unique. When a cow gets sick, it gets replaced.

This analogy really holds true if your operations are in the public cloud. Servers in the cloud are capable of being decommissioned by the cloud provider at any time. A particular cloud server may wind up on a flaky piece of hardware, or it may be on an over-provisioned physical host. By thinking of these servers as part of a uniform, identical fleet of machines — or cattle — you then can start making intelligent decisions as to whether to replace machines that aren't performing well or auto-scale to add more capacity. Having a solid process in place will allow you to build the scalability the business needs.

Architecting your process for scalability is important too. Even if you don't need scalability today, it's worth thinking about and making sure you've planned for it while you're putting your process and workflow together. By thinking about it up front, you'll ensure that you won't have to go back and try to hack it in later — or worse, totally scrap your process and begin building a new one from scratch right when you need it the most.

Quality: Your process needs to achieve two things in the end. It should achieve your desired outcome, with the highest quality possible. Your process should have quality assurance tests along the way, so that anything that happens to go wrong will be caught as early as possible.

One vital tool that can be used as part of your process is code review. When you deploy a configuration management tool, the actual configuration management code is just like any other software deliverable. As such, you can do a code review process on it, and you should. Having another person (or group of people) check work before it is committed and pushed into production is an easy and inexpensive step to implement in any workflow, and it can help ensure that the configuration management code is free of errors and does what you want it to do.

An equally vital concept to engineer into your process is test driven development (TDD). If you can write a small test for your configuration management code first — before you write the configuration management code itself — then actually write the code and put it through that test, you'll have a way to check the code as it continues to evolve and grow.

Using the same example environment from the scenarios I mentioned earlier, one of the best things we implemented as part of our process was a code review step. Our source code management system (Atlassian's Stash product) allows for in-line code reviews and makes collaborating extremely easy. Having another person take a look at something before it goes live has kept us from rolling out inefficient (or flat-out incorrect) infrastructure code more than a couple of times.

Automation doesn't have to be a super-complicated set of programs and triggers.

Automation

Once you have a process sorted out and running successfully, you can start to think about how to automate steps of that process — or ideally, even automate the whole process, end to end.

Automation doesn't mean that you can eliminate portions of your team or that you're out of a job. Automation, done reliably, is an incredibly important part of increasing your team's capacity to get stuff done. Again, if you have a task that's being done manually (but this time, your process describes how that task is being done), you seriously should consider automating that task. If you do a task once, that's either a fluke or the first time you'll ever do it. If you're doing it a second time, you should be thinking about how you'll automate it (not if, but *how*), because by the third time you have to do that task, you already should have turned it into something that's automated that you or your staff no longer needs to worry about.

Automation doesn't have to be a super-complicated set of programs and triggers. Automating something can be as simple as a one-line cron job that's then controlled by your configuration management system and pushed to all the servers in your environment that need it. Don't forget about other types of hooks that some software packages have either. For example, it's possible to trigger Jenkins jobs by hitting a URL and presenting a token, which means you could have a script reach out to your continuous integration server and automatically build or deploy something with one line of code.

Once you automate a task, it can fade into the background as part of the machinery, allowing your team to move on to other things — like learning new technologies and building skills in other areas — and this makes your team even more valuable to the larger organization. This upleveling over time is a huge effectiveness multiplier in the long term.

Automation can seem like a daunting task if you are managing an environment that has none. The best bit of advice here is to start small. Identify one irritating pain point that's very visible but relatively small in scope. Automate that one pain point so that it simply disappears, and people rapidly will buy in to the concept of further automation.

Configuration management systems can be very important tools for automating your team's work. These tools provide ways to manage a fleet of machines and not only automate the provisioning of those machines, but also allow for ongoing maintenance of those machines in an automated fashion as they proceed through their life cycle.

Referring back to my anecdotal environment example: When I inherited that messy environment, the second thing I did (after establishing an easy-to-follow consistent process) was to implement Puppet. We started off small by automating just the configuration of the NTP time sync dæmon. By doing this, we found that approximately half the deployment was referring to the corporate NTP server, and the other half was referring to the publicly available NTP servers on the Internet. This slight difference in time sync was enough to cause issues with a couple time-sensitive applications. Once we got NTP squared away, we moved on to automating and controlling our SSH configuration, and then we started getting more advanced with Apache configurations.

Automation doesn't have to be a big switch that you switch on all of a sudden. It can be rolled out in a small fashion at first, and gradually added to as time goes on and you and your team gain more confidence with automation tools and techniques.

Infrastructure as Code

"Infrastructure as code" is a DevOps term that's been used quite a bit in the DevOps community. It sounds fairly complex, but what infrastructure as code means is that the system configurations, automation and other things needed to run your environment are treated exactly like source code.

This can be a little baffling if you and your team never have worked with a source code management (SCM) system like Git, Perforce or SVN. At its simplest, an SCM will give you a centralized place to put your team's code. All scripts, configuration management code and *almost* anything else that is required to deploy servers, software or code should live in the SCM. (A good exception to this is anything requiring security credentials. Because an SCM is by nature a place for sharing, you probably don't want to put security credentials in your SCM.)

Once you start using an SCM in your process and treating your infrastructure as code, you can do a lot of interesting things with it. As an example, you can branch off your code and make modifications to the branch without affecting the mainline (or released versions that may be running). This can allow a team member to experiment in a sandbox with a different way of doing something, or it can enable your team to create a new type of test environment quickly and on the fly. Those branches can have a life cycle of their own, where they can branch off, live for a bit and die — or they can be merged back into the mainline, and those changes can live on in future versions of your infrastructure code.

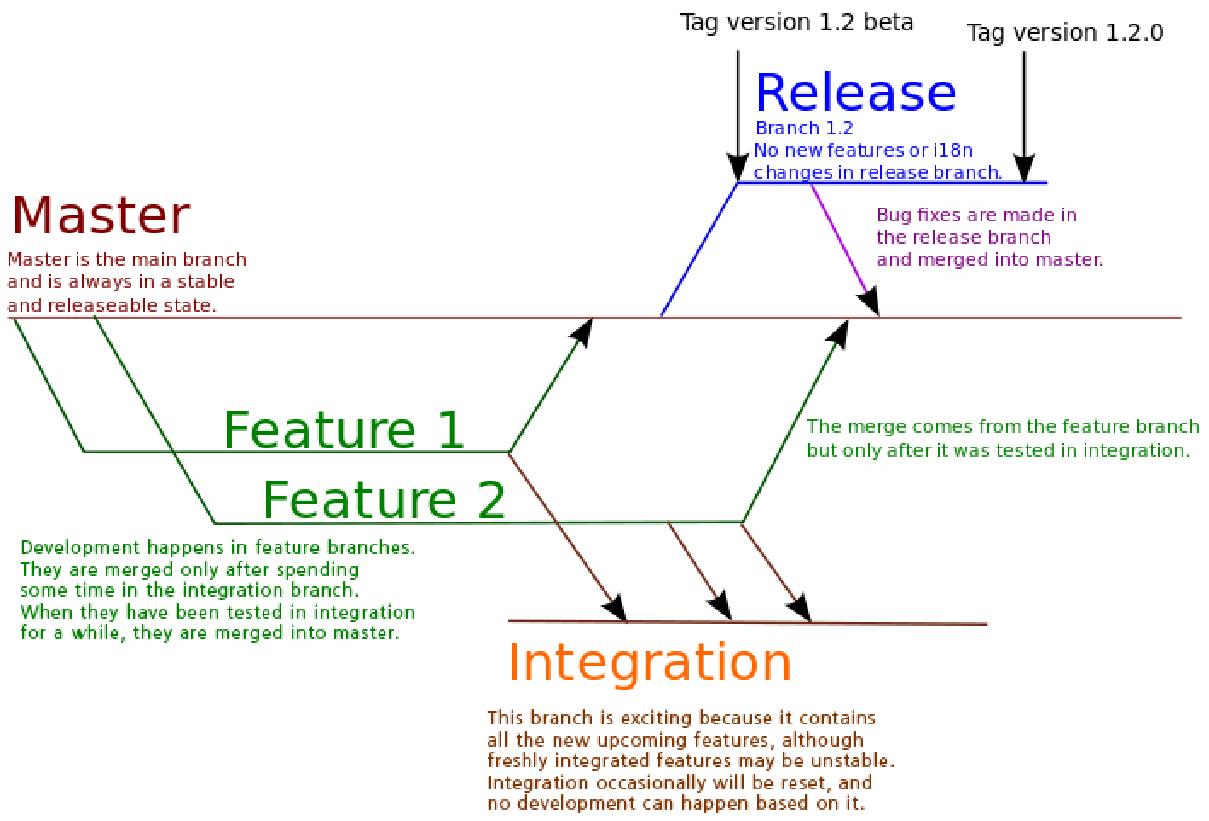

Tag version 1.2 beta
Tag version 1.2.0
Release
Branch 1.2
No new features or i18n changes in release branch.
Master
Master is the main branch and is always in a stable and releaseable state.
Bug fixes are made in the release branch and merged into master.
Feature 1
Feature 2
The merge comes from the feature branch but only after it was tested in integration.
Development happens in feature branches. They are merged only after spending some time in the integration branch. When they have been tested in integration for a while, they are merged into master.
Integration
This branch is exciting because it contains all the new upcoming features, although freshly integrated features may be unstable. Integration occasionally will be reset, and no development can happen based on it.

FIGURE 2. Example Git Workflow

A great advantage of treating infrastructure as code is that you no longer need to maintain a separate document repository (unless you wish). Your team's infrastructure code is a living, breathing example of "executable documentation." So long as the configuration management tool and its code exist, you can re-create a given server or sets of servers reliably, as well as describe how they are built and maintained. This is excellent for auditing and compliance purposes, as well as for disaster recovery.

SCM packages have a great feature that is designed for developer use, but it can be put to advantage in an operations environment too. Every check-in to the SCM is logged, and the code can be rolled back to any check-in, at any point in time. This means that it's possible to roll back an environment to a known good state in the event that a change triggers something going wrong.

If you have a directive to operate in a transparent fashion within your company, adopting infrastructure code concepts will make it easy to do that. If others want to collaborate or view how your team operates, you simply can grant them read-only or read-write access to the SCM repository that contains your infrastructure as code. Developers have been doing this kind of thing for a long time, but only recently has this concept been used in operations teams.

The example environment I've been describing here was a place where we managed our infrastructure as code. Because we had a baseline of what the environment was when we handed it off to the other team, we were able to pull that old configuration out of the SCM, update it for new system updates and packages, test that configuration on a sample of machines, and then roll out the new configuration management code to all the servers in a much shorter time than it originally took.

Ownership and Delegation

Once you develop a solid process, get your automation squared away and adopt infrastructure as code, an interesting thing can happen. You'll notice clear ownership of certain areas of your environment by various team members, thanks to the fact that their contributions can be identified easily by the SCM. This will become clear to your team's members as well, and they'll start identifying the subject-matter experts in a given space and asking them for advice and help when needed, without management's intervention.

Delegation is an interesting phenomenon you may encounter as well. Team members will begin to offload work they can't do effectively to other team members, creating a load-balancing effect within your team that may increase efficiency overall. In some cases, if your SCM repos are open to other teams, it's possible to engage other people in the larger organization for help, without needing to have a lengthy training and on-boarding process. Once other teams understand the use of the configuration management and automation tools you're using, you'll be able to collaborate with them and solve problems more quickly.

Having a solid, documented process with automation in place and infrastructure managed as code will make delegating (or in some cases, flat-out transferring ownership) easy.

Using the processes and tools listed throughout this ebook, my team was able to hand off a documented, functioning, reliable environment to another team — and put it back to that known good state without too much hassle. Those tasks simply would not have been possible otherwise. Putting the environment back to a known good, sustainable and supportable configuration would have been prohibitive without those tools.

Conclusion

Infrastructure sprawl is a very real thing, particularly in a heterogeneous environment or a cloud deployment. Sprawl can be like a tidal wave too — if you're not on top of it, you're under it, and it's sweeping away your ability to invent and innovate along with your team's valuable time and sanity. Like most things, there's no single silver bullet that can be used to combat sprawl. But if you take the time to understand your environment and business objectives, you can create your own custom process and automation that works for you and your team — and then you can start to develop your infrastructure as code. Over time, you'll start to slay the multi-headed hydra that is infrastructure sprawl, and your team will reap the benefits in terms of increased productivity, higher quality and more time to work on other more strategic projects. ■

Resources

Puppet Blog: https://puppetlabs.com/blog

Puppet Forge: https://forge.puppetlabs.com/

Infrastructure as Code:
https://puppetlabs.com/solutions/infrastructure-as-code

2015 State of DevOps Report:
https://puppetlabs.com/2015-devops-report