2023-04-23

Master of Puppets

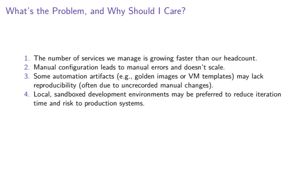
Master of Puppets

Infrastructure as Code and Configuration Management on an Open Source Budget

Mike Renfro (renfro@tntech.edu)

Tennessee Tech University

2023-04-24



What's the Problem, and Why Should I Care?

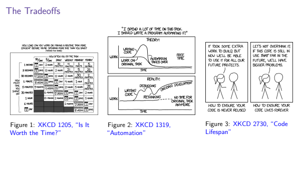
1. The number of services we manage is growing faster than our headcount.
2. Manual configuration leads to manual errors and doesn't scale.
3. Some automation artifacts (e.g., golden images or VM templates) may lack reproducibility (often due to uncrecorded manual changes).
4. Local, sandboxed development environments may be preferred to reduce iteration time and risk to production systems.

So the main reason I'm interested in this sort of topic is that we're being asked to manage and integrate more services without a corresponding increase in headcount.

Lots of the old manual methods can be error-prone and don't scale.

We sometimes make gold images or VM templates to capture a known good state, but we can't necessarily reproduce that artifact.

I'd also like to accelerate new service development in a way that reduces exposure to production systems.



The Tradeoffs

HOW LONG CAN YOU WORK ON MAKING A ROUTINE TASK MORE EFFICIENT BEFORE YOU'RE SPENDING MORE TIME THAN YOU SAVE?
HOW OFTEN YOU DO THE TASK
HOW MUCH TIME YOU SHAVE OFF
"I SPEND A LOT OF TIME ON THIS TASK. I SHOULD WRITE A PROGRAM AUTOMATING IT!"
THEORY:
WORK
WRITING CODE
WORK ON ORIGINAL TASK
AUTOMATION TAKES OVER
FREE TIME
TIME
REALITY:
WORK
WRITING CODE
DEBUGGING
RETHINKING
ONGOING DEVELOPMENT
NO TIME FOR ORIGINAL TASK ANYMORE
TIME
IT TOOK SOME EXTRA WORK TO BUILD, BUT NOW WE'LL BE ABLE TO USE IT FOR ALL OUR FUTURE PROJECTS.
HOW TO ENSURE YOUR CODE IS NEVER REUSED
LET'S NOT OVERTHINK IT; IF THIS CODE IS STILL IN USE THAT FAR IN THE FUTURE, WE'LL HAVE BIGGER PROBLEMS.
HOW TO ENSURE YOUR CODE LIVES FOREVER

Figure 1: XKCD 1205, "Is It Worth the Time?"

Figure 2: XKCD 1319, "Automation"

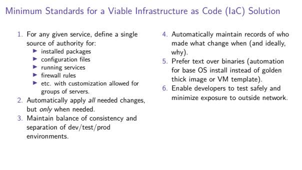
Figure 3: XKCD 2730, "Code Lifespan"

And here are the things we're constantly balancing in IT automation.

On the left, there's a clean simple equation comparing how much time you'll save in the future if you automate something now,

but the reality is that it's hard to know when to stop development,

and it's hard to predict if your new thing (whether automated or manual) is going to be holding up the entire infrastructure for decades or if it'll be abandoned within a year.



Minimum Standards for a Viable Infrastructure as Code (IaC) Solution

1. For any given service, define a single source of authority for:
   - installed packages
   - configuration files
   - running services
   - firewall rules
   - etc. with customization allowed for groups of servers.
2. Automatically apply *all* needed changes, but *only* when needed.
3. Maintain balance of consistency and separation of dev/test/prod environments.
4. Automatically maintain records of who made what change when (and ideally, why).
5. Prefer text over binaries (automation for base OS install instead of golden thick image or VM template).
6. Enable developers to test safely and minimize exposure to outside network.

But if we're going to take the time to automate, let's set out some goals for what success looks like.

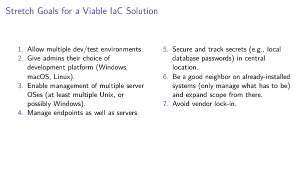
I'd like a single source of authority on system configuration related to a service.

I want to reduce configuration drift, but not keep burning CPU time doing work that's already been done.

I want to know who changed what when, and why.

I want to reduce the number of big binary artifacts I carry around.

And I want to be able to let developers test and break things without risking production services.



Stretch Goals for a Viable IaC Solution

1. Allow multiple dev/test environments.
2. Give admins their choice of development platform (Windows, macOS, Linux).
3. Enable management of multiple server OSes (at least multiple Unix, or possibly Windows).
4. Manage endpoints as well as servers.
5. Secure and track secrets (e.g., local database passwords) in central location.
6. Be a good neighbor on already-installed systems (only manage what has to be) and expand scope from there.
7. Avoid vendor lock-in.

If I can, I want to have multiple areas of development and testing going on at once.

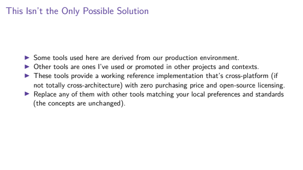
I want people to develop on their choice of platform.

I'd like to manage multiple operating systems, both endpoints and servers.

I'd like to securely store, track, and distribute secrets.

I want to be a good neighbor on existing systems. If all I want to do is deploy the new antivirus and nothing else, I should be able to.

And I want to avoid vendor lock-in. Vendors are great, but business models change, companies get bought, and I'd like to reduce my exposure to those.

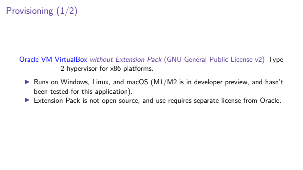


This Isn't the Only Possible Solution

- Some tools used here are derived from our production environment.
- Other tools are ones I've used or promoted in other projects and contexts.
- These tools provide a working reference implementation that's cross-platform (if not totally cross-architecture) with zero purchasing price and open-source licensing.
- Replace any of them with other tools matching your local preferences and standards (the concepts are unchanged).

Fundamentally, what I'm covering here are *concepts* of infrastructure as code with a reference implementation using cross-platform open source tools.

But if you've got a strong preference or established practices for a competing product, stick with it if you like.



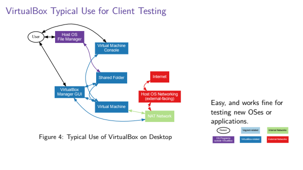
Provisioning (1/2)

Oracle VM VirtualBox *without Extension Pack* (GNU General Public License v2) Type 2 hypervisor for x86 platforms.

- Runs on Windows, Linux, and macOS (M1/M2 is in developer preview, and hasn't been tested for this application).
- Extension Pack is not open source, and use requires separate license from Oracle.

So on the administrator or developer's client system, they'll need a hypervisor.

I'm showing Oracle's VirtualBox without the Extension Pack to reduce licensing barriers.



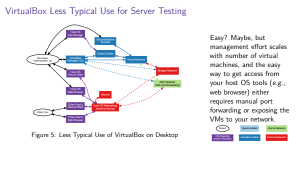
VirtualBox Typical Use for Client Testing

Figure 4: Typical Use of VirtualBox on Desktop

Easy, and works fine for testing new OSes or applications.

So basically everyone here has probably used a desktop hypervisor for testing new operating systems, or setting up a sandbox environment.

For VM quantity 1 and a goal of “get to the Internet”, this isn’t a big deal in VirtualBox.



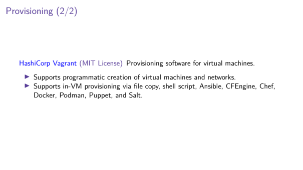
VirtualBox Less Typical Use for Server Testing

Figure 5: Less Typical Use of VirtualBox on Desktop

Easy? Maybe, but management effort scales with number of virtual machines, and the easy way to get access from your host OS tools (*e.g.*, web browser) either requires manual port forwarding or exposing the VMs to your network.

Sometimes we use the same hypervisor to set up servers.

Here things get a bit trickier, as you need to manually configure and install multiple VMs, configure multiple port forwardings or bridged network connections, and it doesn't scale well.



Provisioning (2/2)

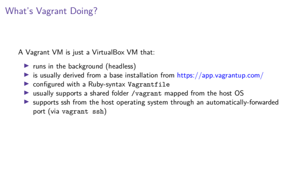
HashiCorp Vagrant (MIT License) Provisioning software for virtual machines.

- Supports programmatic creation of virtual machines and networks.
- Supports in-VM provisioning via file copy, shell script, Ansible, CFEngine, Chef, Docker, Podman, Puppet, and Salt.

So in concert with your hypervisor, I'm adding in HashiCorp's Vagrant.

Vagrant will let you programmatically create, configure, and delete VMs and networks.

And rather than just using OS installation media, they make it easy to run provisioning scripts in several languages.



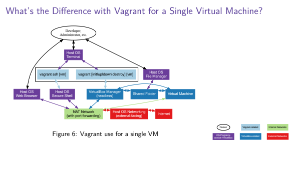
What's Vagrant Doing?

A Vagrant VM is just a VirtualBox VM that:

- runs in the background (headless)
- is usually derived from a base installation from https://app.vagrantup.com/
- configured with a Ruby-syntax `Vagrantfile`
- usually supports a shared folder `/vagrant` mapped from the host OS
- supports ssh from the host operating system through an automatically-forwarded port (via `vagrant ssh`)

Keep in mind that anything you can do in Vagrant, you can do in VirtualBox if you put in enough effort.

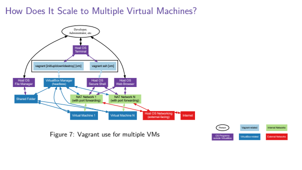
But a Vagrant VM fits a lot of what I want in service development in that it puts the VMs in the background, lets me pick from my choice of operating systems, lets me exchange files through a shared folder, and lets me ssh from my host into the VM easily.



What's the Difference with Vagrant for a Single Virtual Machine?

Figure 6: Vagrant use for a single VM

The Vagrant architecture on a single VM isn't too different from the manual VirtualBox one, except that our person now only has to interact with a terminal, their file manager, and the `vagrant` commands. All the VirtualBox complexity is hidden behind Vagrant.

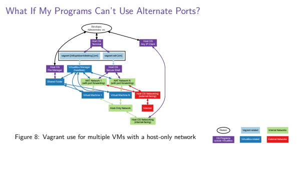


How Does It Scale to Multiple Virtual Machines?

Developer, Administrator, etc.
Host OS Terminal
vagrant [init|up|down|destroy] [vm]
vagrant ssh [vm]
Host OS File Manager
VirtualBox Manager (headless)
Host OS Secure Shell
Host OS Web Browser
Shared Folder
NAT Network 1 (with port forwarding)
NAT Network N (with port forwarding)
Virtual Machine 1
Virtual Machine N
Host OS Networking (external-facing)
Internet
Person
Vagrant-related
Internal Networks
OS Programs outside VirtualBox
VirtualBox-related
External Networks

Figure 7: Vagrant use for multiple VMs

The advantages get more apparent when you add in multiple VMs. The person still only has three primary tools to work with, and managing extra VMs is a matter of adding extra lines in a Vagrantfile.



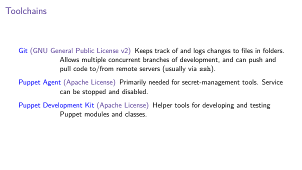
What If My Programs Can't Use Alternate Ports?

Developer, Administrator, etc.
Host OS Terminal
Host OS Any IP Client
vagrant [init|up|down|destroy] [vm]
vagrant ssh [vm]
Host OS File Manager
VirtualBox Manager (headless)
Host OS Secure Shell
Shared Folder
NAT Network 1 (with port forwarding)
NAT Network N (with port forwarding)
Virtual Machine 1
Virtual Machine N
Host OS Networking (external-facing)
Host-Only Network
Internet
Host OS Networking (internal-facing)
Person
Vagrant-related
Internal Networks
OS Programs outside VirtualBox
VirtualBox-related
External Networks

Figure 8: Vagrant use for multiple VMs with a host-only network

Another thing Vagrant makes easier is more complex network configurations. Maybe you've got a client on your host OS that doesn't let you change port numbers, or you really need to see something running on port 80 or 443 for reproducibility.

Because you can set up a host-only network where each VM is a first-class citizen with its own IP, all of that is feasible.



Toolchains

Git (GNU General Public License v2) Keeps track of and logs changes to files in folders. Allows multiple concurrent branches of development, and can push and pull code to/from remote servers (usually via `ssh`).

Puppet Agent (Apache License) Primarily needed for secret-management tools. Service can be stopped and disabled.

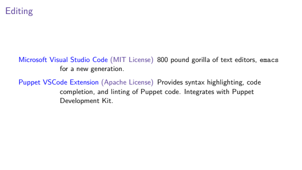
Puppet Development Kit (Apache License) Helper tools for developing and testing Puppet modules and classes.

Other things we'll need to install on the administrator's client system are:

Git for verstion control

Puppet Agent, mostly for managing secrets, so you can definitely stop and disable its background service.

The Puppet Development Kit that makes it easier to create modules and classes with correct documentation and testing.

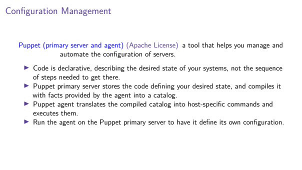


Editing

Microsoft Visual Studio Code (MIT License) 800 pound gorilla of text editors, `emacs` for a new generation.

Puppet VSCode Extension (Apache License) Provides syntax highlighting, code completion, and linting of Puppet code. Integrates with Puppet Development Kit.

For editing, I do a lot of Visual Studio Code, especially because there's a Puppet extension that will do syntax highlighting, code completion, and linting.



Configuration Management

Puppet (primary server and agent) (Apache License) a tool that helps you manage and automate the configuration of servers.

- Code is declarative, describing the desired state of your systems, not the sequence of steps needed to get there.
- Puppet primary server stores the code defining your desired state, and compiles it with facts provided by the agent into a catalog.
- Puppet agent translates the compiled catalog into host-specific commands and executes them.
- Run the agent on the Puppet primary server to have it define its own configuration.

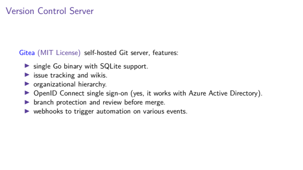
Inside the VMs that we'll build with Vagrant and host in VirtualBox, we'll need the Puppet agent on everything, and the Puppet server programs on the designated Puppet server.

If you've never seen it before, Puppet code is declarative rather than procedural. Similar to Desired State Configuration in Windows, it describes a configuration rather than a particular sequence of steps to realize that configuration.

When Puppet agents check in, they send over a bunch of attributes including IP addresses, operating system versions, and other "facts" about themselves.

The Puppet primary server integrates those facts into a compiled catalog of resources that the agent system should have.

And the agent will then order those resources as needed to realize that configuration.



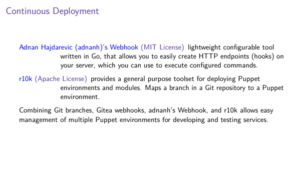
Version Control Server

Gitea (MIT License) self-hosted Git server, features:

- single Go binary with SQLite support.
- issue tracking and wikis.
- organizational hierarchy.
- OpenID Connect single sign-on (yes, it works with Azure Active Directory).
- branch protection and review before merge.
- webhooks to trigger automation on various events.

Gitea is a Git server that's incredibly easy to deploy.
It's a single Go binary that doesn't require an external database server
and has a lot of the features I've grown to expect with GitHub and other services.

In particular, we're going to make use of its ability to trigger automation on other systems through a webhook request.



Continuous Deployment

Adnan Hajdarevic (adnanh)'s Webhook (MIT License) lightweight configurable tool written in Go, that allows you to easily create HTTP endpoints (hooks) on your server, which you can use to execute configured commands.

r10k (Apache License) provides a general purpose toolset for deploying Puppet environments and modules. Maps a branch in a Git repository to a Puppet environment.

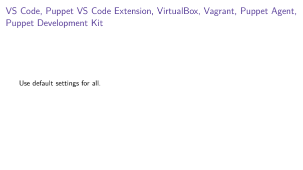
Combining Git branches, Gitea webhooks, adnanh's Webhook, and r10k allows easy management of multiple Puppet environments for developing and testing services.

So we need something to react to Gitea's webhook request, and that's another single-Go-binary program cleverly called `webhook`.

It lets you run arbitrary commands when a request is made to the webhook, and can host several hooks on one system.

What do we want to run from the webhook? It'll be the `r10k` program that lets us map branches in Git to isolated environments in Puppet.

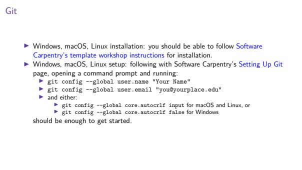
That way, we can assign pilot systems to an alternative environment and test/break things there without risking production at every step.



VS Code, Puppet VS Code Extension, VirtualBox, Vagrant, Puppet Agent, Puppet Development Kit

Use default settings for all.

So for installation of the needed tools on the host system, you can just take the default values for anything in VS Code, VirtualBox, Vagrant, and Puppet.



Git

- Windows, macOS, Linux installation: you should be able to follow Software Carpentry's template workshop instructions for installation.
- Windows, macOS, Linux setup: following with Software Carpentry's Setting Up Git page, opening a command prompt and running:
  - `git config --global user.name "Your Name"`
  - `git config --global user.email "you@yourplace.edu"`
  - and either:
    - `git config --global core.autocrlf input` for macOS and Linux, or
    - `git config --global core.autocrlf false` for Windows

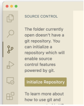
  should be enough to get started.

For Git, I've linked to Software Carpentry's instructions for installation, since there's a few places we'd like to deviate from the defaults.

And we'd like to configure names, email addresses, and end-of-line handling between Windows and non-Windows systems.

Git in Visual Studio Code

- File / Open Folder
- Find existing folder, or create empty one.
- View / Source Control
- Initialize Repository button



SOURCE CONTROL

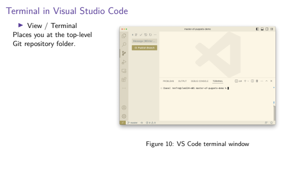
The folder currently open doesn't have a git repository. You can initialize a repository which will enable source control features powered by git.

To learn more about how to use git and

Figure 9: VS Code Initialize Repo button

The first main thing we want to get going in VS Code is version control with Git.

If you use the File / Open Folder menu and either open an existing or new folder, then in the View / Source Control menu (or the branch-looking icon on the left), there'll be an Initialize Repository button you can use to turn that folder into a local Git repository.



Terminal in Visual Studio Code

- View / Terminal
  Places you at the top-level Git repository folder.

```
(base) renfro@clen224-m01 master-of-puppets-demo %
```

Figure 10: VS Code terminal window

The other main thing we want to set up in VS Code is the terminal.

You can get to it with the View / Terminal menu, or by pressing Control and backquote (and even on a Mac, it's the Control key and not the Command key).

How To Make the First Vagrant VM?

We want a VM:

- running a Red Hat-family OS, version 8
- supporting a shared folder `/vagrant` mapped from the host OS

You can find lots of starting points at https://app.vagrantup.com/.

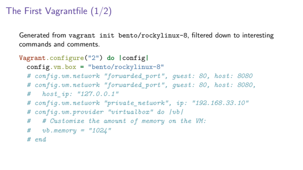
Start the definition of a new Vagrant VM in the repository folder with `vagrant init bento/rockylinux-8`, and look at the `Vagrantfile` that was just created.

(The `bento` Vagrant boxes are built by the Chef Bento project.)

Ok, so if we want to make a VM running something in the Red Hat 8 family, we can search app.vagrantup.com for different options.

I'm going with the Rocky Linux one since it's what I use in our high performance computing environment, and it's an ideological sucessor to what CentOS was in version 7.

So do a `vagrant init bento/rockylinux-8` to get an updated and minimized Rocky VM from the Chef Bento project.



The First Vagrantfile (1/2)

Generated from `vagrant init bento/rockylinux-8`, filtered down to interesting commands and comments.

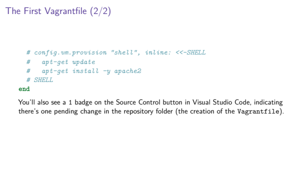
```
Vagrant.configure("2") do |config|
  config.vm.box = "bento/rockylinux-8"
  # config.vm.network "forwarded_port", guest: 80, host: 8080
  # config.vm.network "forwarded_port", guest: 80, host: 8080,
  #   host_ip: "127.0.0.1"
  # config.vm.network "private_network", ip: "192.168.33.10"
  # config.vm.provider "virtualbox" do |vb|
  #   # Customize the amount of memory on the VM:
  #   vb.memory = "1024"
  # end
```

That `vagrant init` will happen pretty quickly because all it does is make a Ruby-syntax `Vagrantfile` with the VM definition.

So it doesn't download or install anything.

If we look at the `Vagrantfile`, it's got a lot of suggested settings that are commented out.

I'm showing here the ones that matter the most to us: here we've got networking and resource requirements



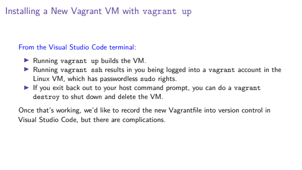
The First Vagrantfile (2/2)

```
    # config.vm.provision "shell", inline: <<-SHELL
    #   apt-get update
    #   apt-get install -y apache2
    # SHELL
  end
```

You'll also see a 1 badge on the Source Control button in Visual Studio Code, indicating there's one pending change in the repository folder (the creation of the `Vagrantfile`).

And here we've got provisioning of the VM after the main download has finished.

Here you can see it's just a few lines that don't actually work with a Red Hat family VM, as `vagrant init` isn't doing any kind of validation of the file when it's created.



Installing a New Vagrant VM with `vagrant up`

From the Visual Studio Code terminal:

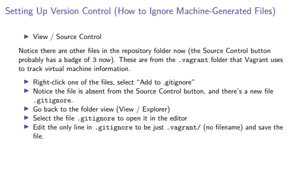
- Running `vagrant up` builds the VM.
- Running `vagrant ssh` results in you being logged into a `vagrant` account in the Linux VM, which has passwordless `sudo` rights.
- If you exit back out to your host command prompt, you can do a `vagrant destroy` to shut down and delete the VM.

Once that's working, we'd like to record the new Vagrantfile into version control in Visual Studio Code, but there are complications.

Clicking on these links referring to a terminal usually means I've got a link to a screencast showing the commands and output for that step.

So here, from the folder containing the `Vagrantfile`, we can do a `vagrant up` and see the base VM image get downloaded, the port forwarding for secure shell (if you look closely, you can see it works around other VMs that already have port forwards in place), and mounts the shared folder from the host filesystem.

Once the VM is up, we can ssh in with `vagrant ssh`. There's no password required since Vagrant sets up public key authentication for us, and that account has passwordless `sudo` on the VM, which makes it easy to set up applications and other services.



Setting Up Version Control (How to Ignore Machine-Generated Files)

- View / Source Control

Notice there are other files in the repository folder now (the Source Control button probably has a badge of 3 now). These are from the `.vagrant` folder that Vagrant uses to track virtual machine information.

- Right-click one of the files, select “Add to .gitignore”
- Notice the file is absent from the Source Control button, and there’s a new file `.gitignore`.
- Go back to the folder view (View / Explorer)
- Select the file `.gitignore` to open it in the editor
- Edit the only line in `.gitignore` to be just `.vagrant/` (no filename) and save the file.

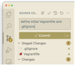
So if you’ve got a Vagrant VM created or running, there’ll be extra files showing up in a `.vagrant` folder in the repository.
We don’t want them cluttering up the repository, as they’re all machine-generated and can always be recreated.

So we’ll right-click one, add it to a `.gitignore` file, and if we edit the `gitignore` to just have the folder name and not a specific file, we can ignore everything from that folder on down.

Setting Up Version Control (Adding and Committing Useful Files)

- Notice the Source Control badge now reads 2 (one for `Vagrantfile`, one for `.gitignore`).
- Select both `Vagrantfile` and `.gitignore`, select "Stage Changes".
- In the "Message" text entry, enter `Define initial Vagrantfile and .gitignore` and select the "Commit" button.

In theory, Git commits should be "atomic", *i.e.*, a single, complete unit of work that can be described in a single sentence. In practice, we're often not that disciplined about it. Git commit messages should be short and imperative, completing the sentence, "when applied, this commit will . . . ".



define initial Vagrantfile and .gitignore

Figure 11: Readying first Git commit

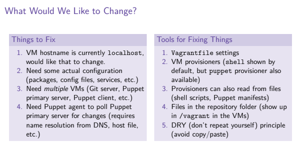
Once you've got the .gitignore saved, the Source Control badge number will drop to a 2, one for the `Vagrantfile`, and one for the `.gitignore`.

You can right-click both of them to add them to the staged changes, and then you can add a commit message and hit the Commit button.

Wherever posible, you want your commits to be atomic, where the changes represent one functional unit of work, as big or as small as you need.

And your commit messages should be short and imperative, completing a sentence "when applied, this commit will. . . "

But realistically, we're not super-disciplined about that in practice.



What Would We Like to Change?

Things to Fix

1. VM hostname is currently `localhost`, would like that to change.
2. Need some actual configuration (packages, config files, services, etc.)
3. Need *multiple* VMs (Git server, Puppet primary server, Puppet client, etc.)
4. Need Puppet agent to poll Puppet primary server for changes (requires name resolution from DNS, host file, etc.)

Tools for Fixing Things

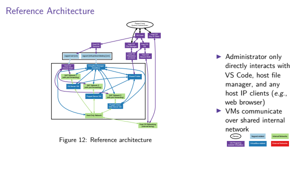
1. `Vagrantfile` settings
2. VM provisioners (`shell` shown by default, but `puppet` provisioner also available)
3. Provisioners can also read from files (shell scripts, Puppet manifests)
4. Files in the repository folder (show up in `/vagrant` in the VMs)
5. DRY (don't repeat yourself) principle (avoid copy/paste)

So we've got the equivalent of a Hello, World in Vagrant, and there's a lot we'll need to change to make a functional environment.

We want to change hostnames; add packages, configuration files, and services; we want to build multiple VMs from the same Vagrantfile, and we'd like to have the VMs poll the Puppet primary server for changes.

Some things we can change directly in the Vagrantfile, others with the available provisioners, some by copying files from the shared folder we showed earlier.

Ideally, we want to factor out everything into separate files and functions, so that we don't have a lot of copy/paste going on.



Reference Architecture

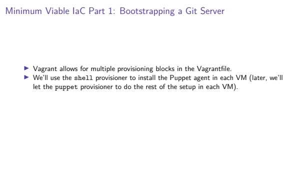
Puppet-Using Systems Administrator
VS Code
Host OS Any IP Client
Host OS Terminal
Host OS File Manager
Host OS Git
vagrant ssh [vm]
vagrant [init|up|down|destroy] [vm]
Local Git Repository (or clone)
Host OS Secure Shell
Host OS Secure Shell
VirtualBox Manager (headless)
NAT Network 1 (with port forwarding)
Shared Folder
Git Server VM
NAT Network 2 (with port forwarding)
Puppet Server VM
NAT Network 3 (with port forwarding)
Puppet Client, Web Server VM
Host-Only Network
Host OS Networking (internal-facing)
Person
Vagrant-related
Internal Networks
OS Programs outside VirtualBox
VirtualBox-related
External Networks

Figure 12: Reference architecture

- Administrator only directly interacts with VS Code, host file manager, and any host IP clients (*e.g.*, web browser)
- VMs communicate over shared internal network

So the overall architecture we're heading for will have three VMs managed by Vagrant: a Git server, a Puppet primary server, and a barebones web server that will get its configuration from the Puppet primary server.
They'll communicate amongst themselves over a network they share with the host OS.
That will also let us communicate with the Git server from Visual Studio code or from a host web browser.

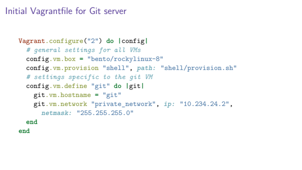
Everything inside the rectangular area is abstracted away by Vagrant, so we reduce a lot of the scaling and chances for manual error.



Minimum Viable IaC Part 1: Bootstrapping a Git Server

- Vagrant allows for multiple provisioning blocks in the Vagrantfile.
- We'll use the `shell` provisioner to install the Puppet agent in each VM (later, we'll let the `puppet` provisioner to do the rest of the setup in each VM).

So Vagrant will let us use multiple provisioners.
The shell provisioner will work out of the box on any Unix system,
and if we use that to install the Puppet agent, we can do the rest of the setup inside Puppet.



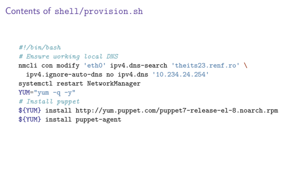
Initial Vagrantfile for Git server

```ruby
Vagrant.configure("2") do |config|
  # general settings for all VMs
  config.vm.box = "bento/rockylinux-8"
  config.vm.provision "shell", path: "shell/provision.sh"
  # settings specific to the git VM
  config.vm.define "git" do |git|
    git.vm.hostname = "git"
    git.vm.network "private_network", ip: "10.234.24.2",
      netmask: "255.255.255.0"
  end
end
```

So here’s a more minimized Vagrantfile for the Git server.

We can factor out some common settings with the `config` object, like having a common operating system image or a common shell script.

Other things can be customized for each VM, like its hostname and IP.

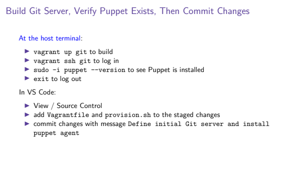


Contents of `shell/provision.sh`

```bash
#!/bin/bash
# Ensure working local DNS
nmcli con modify 'eth0' ipv4.dns-search 'theits23.renf.ro' \
  ipv4.ignore-auto-dns no ipv4.dns '10.234.24.254'
systemctl restart NetworkManager
YUM="yum -q -y"
# Install puppet
${YUM} install http://yum.puppet.com/puppet7-release-el-8.noarch.rpm
${YUM} install puppet-agent
```

There's two main things we want to have this provisioning script do:

1. Use a local DNS server so that the Git server can contact the webhook on the Puppet server by name, and the Puppet server can pull changes from the Git server by name.
2. Install the Puppet agent for all other configuration.



Build Git Server, Verify Puppet Exists, Then Commit Changes

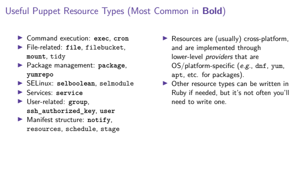
At the host terminal:

- `vagrant up git` to build
- `vagrant ssh git` to log in
- `sudo -i puppet --version` to see Puppet is installed
- `exit` to log out

In VS Code:

- View / Source Control
- add `Vagrantfile` and `provision.sh` to the staged changes
- commit changes with message `Define initial Git server and install puppet agent`

So in this screencast, we can see Vagrant build the VM and install Puppet.
If we log into it with vagrant ssh, we can verify the Puppet version,
and afterwards, we can go back into VS Code to add and commit the changes we made to the Vagrantfile and the provision.sh.



Useful Puppet Resource Types (Most Common in **Bold**)

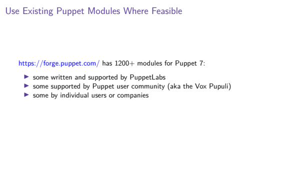
- Command execution: **`exec`**, **`cron`**
- File-related: **`file`**, `filebucket`, **`mount`**, `tidy`
- Package management: **`package`**, **`yumrepo`**
- SELinux: **`selboolean`**, `selmodule`
- Services: **`service`**
- User-related: **`group`**, **`ssh_authorized_key`**, **`user`**
- Manifest structure: **`notify`**, `resources`, `schedule`, `stage`

- Resources are (usually) cross-platform, and are implemented through lower-level *providers* that are OS/platform-specific (*e.g.*, `dnf`, `yum`, `apt`, etc. for packages).
- Other resource types can be written in Ruby if needed, but it's not often you'll need to write one.

So the Puppet provisioner runs the `puppet apply` command, and has full access to all the Puppet features.
The fundamental resources that we can manage in Puppet include:

- running arbitrary commands with exec and cron
- file content and permissions
- package installation and removal
- SELinux settings
- services
- local users, groups, and SSH keys, both for hosts and for users

Those resources don't generally change syntax if you change platforms: a package resource installs with apt on Debian-family systems and with yum or dnf on Red Hat-family systems.



Use Existing Puppet Modules Where Feasible

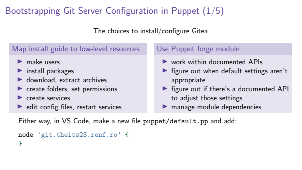
https://forge.puppet.com/ has 1200+ modules for Puppet 7:

- some written and supported by PuppetLabs
- some supported by Puppet user community (aka the Vox Pupuli)
- some by individual users or companies

On the other end of the abstraction spectrum, we've got over 1200 modules on Puppet Forge compatible with Puppet 7.

These can vary in age and quality, but the most common software stacks are pretty well covered by either PuppetLabs themselves, or by various people in the Puppet user community

It's also possible to keep your own modules in any Git repository, whether private or public.



Bootstrapping Git Server Configuration in Puppet (1/5)

The choices to install/configure Gitea

Map install guide to low-level resources

- make users
- install packages
- download, extract archives
- create folders, set permissions
- create services
- edit config files, restart services

Use Puppet forge module

- work within documented APIs
- figure out when default settings aren't appropriate
- figure out if there's a documented API to adjust those settings
- manage module dependencies

Either way, in VS Code, make a new file `puppet/default.pp` and add:

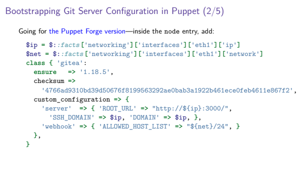
```
node 'git.theits23.renf.ro' {
}
```

So whether it's Gitea or any other service, we can map an installation guide to lower-level resources,

or we can use Puppet modules to abstract away a lot of the implementation details.

And we can mix the two if we need to.

To get started, we'll make a new node entry in the default.pp in the puppet folder with the FQDN of the Git server.



Bootstrapping Git Server Configuration in Puppet (2/5)

Going for the Puppet Forge version—inside the node entry, add:

```
$ip = $::facts['networking']['interfaces']['eth1']['ip']
$net = $::facts['networking']['interfaces']['eth1']['network']
class { 'gitea':
  ensure   => '1.18.5',
  checksum =>
    '4766ad9310bd39d50676f8199563292ae0bab3a1922b461ece0feb4611e867f2',
  custom_configuration => {
    'server'  => { 'ROOT_URL' => "http://${ip}:3000/",
      'SSH_DOMAIN' => $ip, 'DOMAIN' => $ip, },
    'webhook' => { 'ALLOWED_HOST_LIST' => "${net}/24", }
  },
}
```

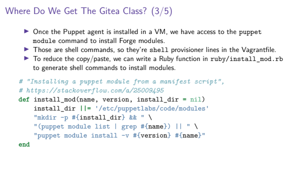
So I decided to go for the Puppet Forge version of setting up Gitea, and though I've had to compact the formatting a bit to fit it on one slide, this is all that it takes.

We've got a couple examples of pulling IP and network addresses from the Gitea host to avoid hard-coding things.

In the interest of time, I did have to hard-code the number of netmask bits, but there are modules that can do that conversion.

And this is about an 80% reduction in lines of code from the lower-level version.

└─Where Do We Get The Gitea Class? (3/5)



Where Do We Get The Gitea Class? (3/5)

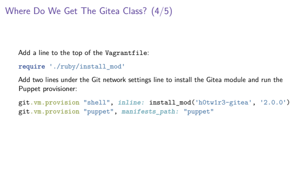
- Once the Puppet agent is installed in a VM, we have access to the `puppet module` command to install Forge modules.
- Those are shell commands, so they're `shell` provisioner lines in the Vagrantfile.
- To reduce the copy/paste, we can write a Ruby function in `ruby/install_mod.rb` to generate shell commands to install modules.

```ruby
# "Installing a puppet module from a manifest script",
# https://stackoverflow.com/a/25009495
def install_mod(name, version, install_dir = nil)
    install_dir ||= '/etc/puppetlabs/code/modules'
    "mkdir -p #{install_dir} && " \
    "(puppet module list | grep #{name}) || " \
    "puppet module install -v #{version} #{name}"
end
```

So how do we get the Gitea module installed from Puppet Forge?

There's a `puppet module install` command that can install modules, but that could get tedious to type repeatedly.

So we can make a Ruby function that returns the shell command to run to install an arbitrary module and version, and we'll pass those strings inline to the shell provisioner.



Where Do We Get The Gitea Class? (4/5)

Add a line to the top of the `Vagrantfile`:

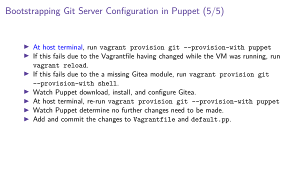
```
require './ruby/install_mod'
```

Add two lines under the Git network settings line to install the Gitea module and run the Puppet provisioner:

```
git.vm.provision "shell", inline: install_mod('h0tw1r3-gitea', '2.0.0')
git.vm.provision "puppet", manifests_path: "puppet"
```

We'll make sure we include the Ruby file with the function in it,
and then we can add a shell provision line for each module we want to install.

As all of those modules have to be installed before the Puppet provisioner runs, we put them in the right order in the Vagrantfile.

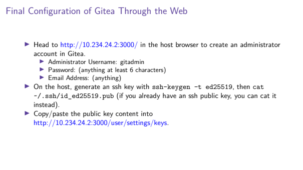


Bootstrapping Git Server Configuration in Puppet (5/5)

- At host terminal, run `vagrant provision git --provision-with puppet`
- If this fails due to the Vagrantfile having changed while the VM was running, run `vagrant reload`.
- If this fails due to the a missing Gitea module, run `vagrant provision git --provision-with shell`.
- Watch Puppet download, install, and configure Gitea.
- At host terminal, re-run `vagrant provision git --provision-with puppet`
- Watch Puppet determine no further changes need to be made.
- Add and commit the changes to `Vagrantfile` and `default.pp`.

So in theory, we'd run the Puppet provisioner to finish setting things up.
But if the VM was still running when we changed the Vagrantfile, we'll have to reload the VM with `vagrant reload`.
If we're missing a Puppet module, we'll need to rerun the shell provisioners to install them, and then we can re-run the Puppet provisioner to use the newly-installed modules.

If we rerun the Puppet provisioner again, it sees that everything is configured as it's supposed to be, and it exits out quickly.



Final Configuration of Gitea Through the Web

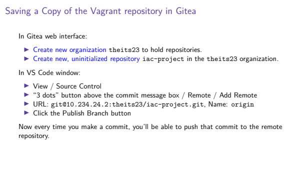
- Head to http://10.234.24.2:3000/ in the host browser to create an administrator account in Gitea.
  - Administrator Username: gitadmin
  - Password: (anything at least 6 characters)
  - Email Address: (anything)
- On the host, generate an ssh key with `ssh-keygen -t ed25519`, then `cat ~/.ssh/id_ed25519.pub` (if you already have an ssh public key, you can cat it instead).
- Copy/paste the public key content into http://10.234.24.2:3000/user/settings/keys.

Once the Puppet provisioner is finished, we’ve got a running Gitea server at the address in the first bullet point.

It’s had some initial setup done, and is waiting on an administrator account to get created.

Once we’ve made the account, we can add an ssh key to it so we can push code to it.

└─Saving a Copy of the Vagrant repository in Gitea



Saving a Copy of the Vagrant repository in Gitea

In Gitea web interface:

- Create new organization `theits23` to hold repositories.
- Create new, uninitialized repository `iac-project` in the `theits23` organization.

In VS Code window:

- View / Source Control
- “3 dots” button above the commit message box / Remote / Add Remote
- URL: `git@10.234.24.2:theits23/iac-project.git`, Name: `origin`
- Click the Publish Branch button

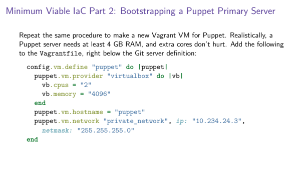
Now every time you make a commit, you’ll be able to push that commit to the remote repository.

Gitea supports separation of accounts and organizations, and you’ll normally want to make an organization, add members to it, and let the organization “own” the repository.

So we’ll make a theits organization and a repository in the organization to hold the Vagrant content and other files we need to stand up this environment.

Once it’s created, we can add a new remote hosted on the Gitea server, and can push to it over ssh.

└─Minimum Viable IaC Part 2: Bootstrapping a Puppet Primary Server

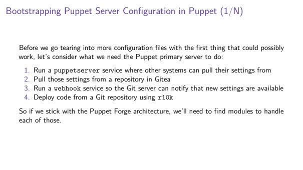


Minimum Viable IaC Part 2: Bootstrapping a Puppet Primary Server

Repeat the same procedure to make a new Vagrant VM for Puppet. Realistically, a Puppet server needs at least 4 GB RAM, and extra cores don't hurt. Add the following to the `Vagrantfile`, right below the Git server definition:

```
config.vm.define "puppet" do |puppet|
  puppet.vm.provider "virtualbox" do |vb|
    vb.cpus = "2"
    vb.memory = "4096"
  end
  puppet.vm.hostname = "puppet"
  puppet.vm.network "private_network", ip: "10.234.24.3",
    netmask: "255.255.255.0"
end
```

Ok, so we can now set up a second server in Vagrant, and it'll be the Puppet primary server. It'll need at least 4 GB of RAM, and extra CPU cores are useful. We'll assign it an IP, too.



Bootstrapping Puppet Server Configuration in Puppet (1/N)

Before we go tearing into more configuration files with the first thing that could possibly work, let's consider what we need the Puppet primary server to do:

1. Run a `puppetserver` service where other systems can pull their settings from
2. Pull those settings from a repository in Gitea
3. Run a `webhook` service so the Git server can notify that new settings are available
4. Deploy code from a Git repository using `r10k`

So if we stick with the Puppet Forge architecture, we'll need to find modules to handle each of those.

Now I didn't show any of the custom provisioning yet, because the Puppet server we want will need to do more than just one thing like the Gitea server does.

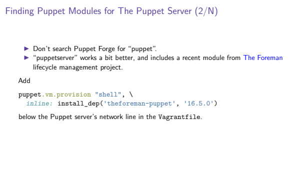
We need it to listen for Puppet agent connections from remote systems.

We need it to pull new settings for those systems from the Git server.

We need that pull from Git to be triggered by a webhook on the Git server when it gets a push of new code.

And we want to run some deploy scripts against the code that we pulled.

So if we're going to use Forge modules, we'll need to piece several together to make this work, and see what's left over that we have to set up more manually.



Finding Puppet Modules for The Puppet Server (2/N)

- Don't search Puppet Forge for "puppet".
- "puppetserver" works a bit better, and includes a recent module from The Foreman lifecycle management project.

Add

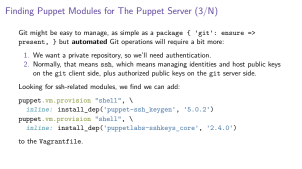
```
puppet.vm.provision "shell", \
  inline: install_dep('theforeman-puppet', '16.5.0')
```

below the Puppet server's network line in the `Vagrantfile`.

For managing the Puppet primary server's puppet service called `puppetserver`, we can find a recent module from the Foreman project.

So we'll drop that into the provisioning lines.

└─Finding Puppet Modules for The Puppet Server (3/N)



Finding Puppet Modules for The Puppet Server (3/N)

Git might be easy to manage, as simple as a `package { 'git': ensure => present, }` but **automated** Git operations will require a bit more:

1. We want a private repository, so we'll need authentication.
2. Normally, that means `ssh`, which means managing identities and host public keys on the `git` client side, plus authorized public keys on the `git` server side.

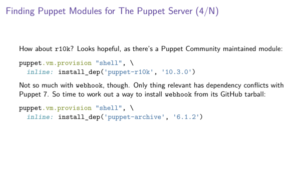
Looking for ssh-related modules, we find we can add:

```
puppet.vm.provision "shell", \
  inline: install_dep('puppet-ssh_keygen', '5.0.2')
puppet.vm.provision "shell", \
  inline: install_dep('puppetlabs-sshkeys_core', '2.4.0')
```

to the `Vagrantfile`.

So installing the Git binary is pretty simple,
but for automated Git usage, we need to manage two sets of ssh keys:
one user-based public key that can be authorized to repositories on the Git server,
and one host public key for the Git server, so that the Git client on Puppet knows its talking to the correct system.

So we'll add one module to automate user key generation, and another module to manage the known hosts file for ssh connections.



Finding Puppet Modules for The Puppet Server (4/N)

How about `r10k`? Looks hopeful, as there's a Puppet Community maintained module:

```
puppet.vm.provision "shell", \
  inline: install_dep('puppet-r10k', '10.3.0')
```

Not so much with `webhook`, though. Only thing relevant has dependency conflicts with Puppet 7. So time to work out a way to install `webhook` from its GitHub tarball:

```
puppet.vm.provision "shell", \
  inline: install_dep('puppet-archive', '6.1.2')
```

There's a module to handle setting up `r10k`, as that's pretty popular package for Puppet people. But surprisingly, there's not a current module for Adnan's webhook, so we'll need to start by downloading and uncompressing a tarball.

We've got the `archive` module for that.

└─Bootstrapping Puppet Server Configuration in Puppet (5/N)

In the interest of time, I'm not going to go through all the Puppet code required to set up the Puppet primary server.

IT'd require about 4 slides or so, but I can review it later if anyone wants.

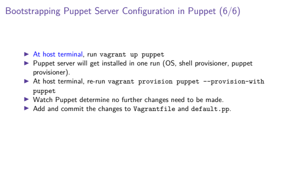
Bootstrapping Puppet Server Configuration in Puppet (5/N)

Even with the Forge modules, still wound up with several slides of code. So go see it here.

Spoiler alert, we ended up needing to resolve a permissions issue between the Foreman's Puppet module and running `r10k` as an unprivileged `puppet` user. So we ended up needing

```
puppet.vm.provision "shell", \
  inline: install_dep('npwalker-recursive_file_permissions', '0.6.2')
```

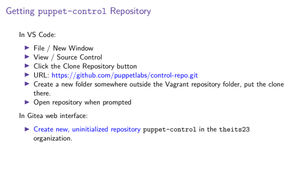
in the `Vagrantfile` as well.



Bootstrapping Puppet Server Configuration in Puppet (6/6)

- At host terminal, run `vagrant up puppet`
- Puppet server will get installed in one run (OS, shell provisioner, puppet provisioner).
- At host terminal, re-run `vagrant provision puppet --provision-with puppet`
- Watch Puppet determine no further changes need to be made.
- Add and commit the changes to `Vagrantfile` and `default.pp`.

Like before, we can run a `vagrant up puppet` to build the BM, and since we've got all the provisioning steps worked out in advance, then barring any bugs, we end up with a fully functional Puppet server ready for webhook and r10k.



Getting `puppet-control` Repository

In VS Code:

- File / New Window
- View / Source Control
- Click the Clone Repository button
- URL: https://github.com/puppetlabs/control-repo.git
- Create a new folder somewhere outside the Vagrant repository folder, put the clone there.
- Open repository when prompted

In Gitea web interface:

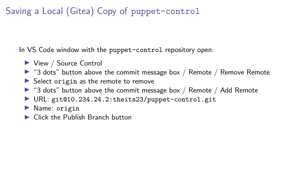
- Create new, uninitialized repository `puppet-control` in the `theits23` organization.

So we need a starting point and some structure for the Puppet code we'll deploy to the webserver we're about to build and manage in Puppet.

If we start a new window in VS Code, we can clone the canonical Puppet control repository from PuppetLabs.

It's got a good bit more sophistication than what we've done with the single default.pp file so far, and it gives a starting point that can scale to most any system you want to manage.

Once we've got the repository cloned to our local system, we can make a new, unititlized repository called `puppet-control` in the local Gitea server.



Saving a Local (Gitea) Copy of `puppet-control`

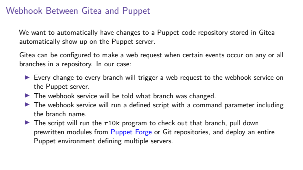
In VS Code window with the `puppet-control` repository open:

- View / Source Control
- "3 dots" button above the commit message box / Remote / Remove Remote
- Select `origin` as the remote to remove
- "3 dots" button above the commit message box / Remote / Add Remote
- URL: `git@10.234.24.2:theits23/puppet-control.git`
- Name: `origin`
- Click the Publish Branch button

We can use VS Code's Source Control tools to remove the GitHub remote we got the control repository from,

add a new remote for Gitea,

and publish the repository to Gitea.



Webhook Between Gitea and Puppet

We want to automatically have changes to a Puppet code repository stored in Gitea automatically show up on the Puppet server.

Gitea can be configured to make a web request when certain events occur on any or all branches in a repository. In our case:

- Every change to every branch will trigger a web request to the webhook service on the Puppet server.
- The webhook service will be told what branch was changed.
- The webhook service will run a defined script with a command parameter including the branch name.
- The script will run the `r10k` program to check out that branch, pull down prewritten modules from Puppet Forge or Git repositories, and deploy an entire Puppet environment defining multiple servers.

Now that we've got the code stored in Gitea, we need to set up a webhook whenever we make a change to the repository.

That way, any time we make a change to the code, Gitea can send a web request to the webhook service on the Puppet server,

include information about what got changed (mostly the branch),

the Puppet server can pull changes from that branch into a separate folder that will make up an isolated environment for the systems we're developing,

and r10k can pull down any Forge modules that we need to make that environment function.

General Data Flows

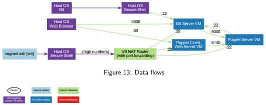
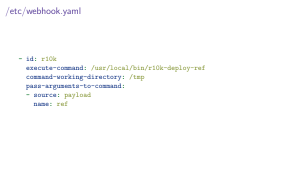


Figure 13: Data flows

This slide shows some more of the internal details and port numbers from the previous slide.
Git pushes changes over ssh to the Git server's port 22,
where Gitea notices the change and makes a web request to Puppet's port 9000,
which runs a deployment script to pull new code from the Git server's port 22 and deploy it.
Later, the web server or any other system running the Puppet agent can pick up on the modified code.



/etc/webhook.yaml

```yaml
- id: r10k
  execute-command: /usr/local/bin/r10k-deploy-ref
  command-working-directory: /tmp
  pass-arguments-to-command:
  - source: payload
    name: ref
```

The webhook YAML shows a hook name, the script to run, and the arguments to pass to the script.

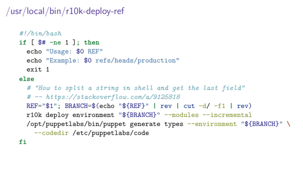
2023-04-23

Master of Puppets

└─Typical Infrastructure as Code Workflows

  └─Webhook Between Gitea and Puppet

    └─/usr/local/bin/r10k-deploy-ref

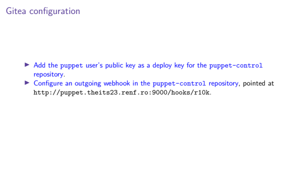


/usr/local/bin/r10k-deploy-ref

```
#!/bin/bash
if [ $# -ne 1 ]; then
  echo "Usage: $0 REF"
  echo "Example: $0 refs/heads/production"
  exit 1
else
  # "How to split a string in shell and get the last field"
  # -- https://stackoverflow.com/a/9125818
  REF="$1"; BRANCH=$(echo "${REF}" | rev | cut -d/ -f1 | rev)
  r10k deploy environment "${BRANCH}" --modules --incremental
  /opt/puppetlabs/bin/puppet generate types --environment "${BRANCH}" \
    --codedir /etc/puppetlabs/code
fi
```

The deployment script can run the `r10k` program that's been configured to know where the control repository is,

pulls code from the updated branch and deploys it.

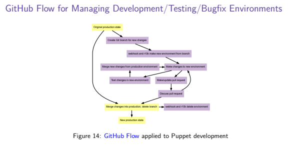


Gitea configuration

- Add the `puppet` user's public key as a deploy key for the `puppet-control` repository.
- Configure an outgoing webhook in the `puppet-control` repository, pointed at `http://puppet.theits23.renf.ro:9000/hooks/r10k`.

For this to work, we'd need to `cat` the `puppet` account's ssh public key and import it as a deploy key for the repository in Gitea.

And we'd need to configure an outgoing webhook in the repository to contact the Puppet server at the URL shown here.



GitHub Flow for Managing Development/Testing/Bugfix Environments

Original production state
Create Git branch for new changes
webhook and r10k make new environment from branch
Merge new changes from production environment
Make changes to new environment
Test changes in new environment
Make/update pull request
Discuss pull request
Merge changes into production, delete branch
webhook and r10k delete environment
New production state

Figure 14: GitHub Flow applied to Puppet development

Another thing that factors into service development and bug-fixing is how you manage multiple branches of work all happening at once.

GitHub Flow is one of the most common methods of managing this.

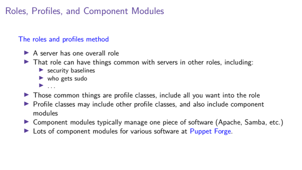
Each new line of work starts by branching off the current state of a primary branch (named `production` in the `puppet-control` repository).

Dvery time we push a change up, the webhook will update the Puppet server.

We iterate over making changes, testing those changes, and most importantly, merging any new changes from the production branch on a regular basis.

Once we're satisfied our code solves the problem it needs to, we make and discuss a pull request to merge the branch back into production, make any more changes required, and eventually merge it back in and delete the just-merged branch.

This sets up guard rails to protect production systems from any breaking changes from the other branches, and allows enormous latitude in the changes that can be made in the isolated environments.



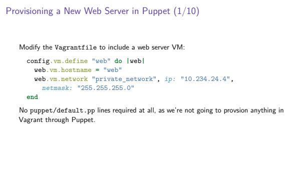
Roles, Profiles, and Component Modules

The roles and profiles method

- A server has one overall role
- That role can have things common with servers in other roles, including:
  - security baselines
  - who gets sudo
  - . . .
- Those common things are profile classes, include all you want into the role
- Profile classes may include other profile classes, and also include component modules
- Component modules typically manage one piece of software (Apache, Samba, etc.)
- Lots of component modules for various software at Puppet Forge.

Another best practice in Puppet service development is the “roles and profiles” method.
In this, we consider a server to have one and only one role (e.g., “a Banner database server”)
That server has several things in common with other servers: it runs an Oracle database, it allows certain groups sudo access, it has particular firewall rules that have to be enabled, etc.
These smaller sets of settings that are common with other servers are called profiles.
We factor out the profiles into their own classes, and include them as part of the server’s role.
That way, we can manage common baseline settings across all applicable servers and reduce duplication.

The last sort of module is a component module, like the Gitea one we used earlier. Component modules usually manage a single piece of software, and are included from profile classes.



Provisioning a New Web Server in Puppet (1/10)

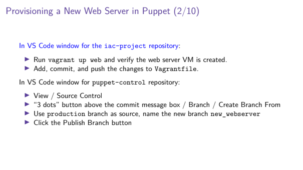
Modify the `Vagrantfile` to include a web server VM:

```
config.vm.define "web" do |web|
  web.vm.hostname = "web"
  web.vm.network "private_network", ip: "10.234.24.4",
    netmask: "255.255.255.0"
end
```

No `puppet/default.pp` lines required at all, as we're not going to provsion anything in Vagrant through Puppet.

So for the web server, we can make a bare-bones Vagrant entry with just the common shell provisioner enabled.

We don't need the `puppet apply` command to do any setup at all in Vagrant, since we've now got a Puppet server that can define the web server's configuration through the `puppet agent` command.



Provisioning a New Web Server in Puppet (2/10)

In VS Code window for the `iac-project` repository:

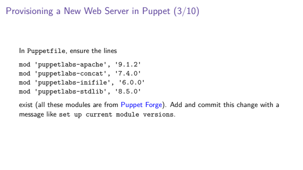
- Run `vagrant up web` and verify the web server VM is created.
- Add, commit, and push the changes to `Vagrantfile`.

In VS Code window for `puppet-control` repository:

- View / Source Control
- "3 dots" button above the commit message box / Branch / Create Branch From
- Use `production` branch as source, name the new branch `new_webserver`
- Click the Publish Branch button

We can do a `vagrant up web` to build the VM and run the shell provisioner. We can add, commit, and push the changes to the Vagrantfile.

And in the `puppet-control` repository, we can make a new branch from production called "new_webserver" and publish that branch to Gitea.



Provisioning a New Web Server in Puppet (3/10)

In `Puppetfile`, ensure the lines

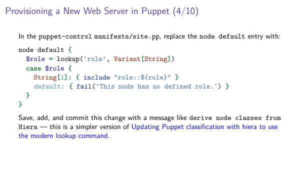
```
mod 'puppetlabs-apache', '9.1.2'
mod 'puppetlabs-concat', '7.4.0'
mod 'puppetlabs-inifile', '6.0.0'
mod 'puppetlabs-stdlib', '8.5.0'
```

exist (all these modules are from Puppet Forge). Add and commit this change with a message like `set up current module versions`.

In the new branch, let's make sure we have the right modules to manage Apache on the web server.

Those are the Apache module and its dependencies that are listed in the file called `Puppetfile`.

Then we can add, commit, and push the Puppetfile.



Provisioning a New Web Server in Puppet (4/10)

In the puppet-control manifests/site.pp, replace the node default entry with:

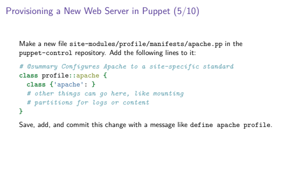
```
node default {
  $role = lookup('role', Variant[String])
  case $role {
    String[1]: { include "role::${role}" }
    default: { fail('This node has no defined role.') }
  }
}
```

Save, add, and commit this change with a message like derive node classes from Hiera — this is a simpler version of Updating Puppet classification with hiera to use the modern lookup command.

We want to modernize the site-wide manifest in Puppet while serves the same role as the default.pp we used in Vagrant.

Instead of making different entries in the file for each node we manage, we use a catch-all default node that looks up a value called role and includes a role class corresponding to that role name.

We'll get back around to where the data comes from shortly, for now, let's define the rest of the web server's classes.



Provisioning a New Web Server in Puppet (5/10)

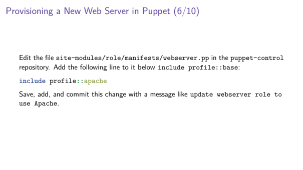
Make a new file `site-modules/profile/manifests/apache.pp` in the `puppet-control` repository. Add the following lines to it:

```
# @summary Configures Apache to a site-specific standard
class profile::apache {
  class {'apache': }
  # other things can go here, like mounting
  # partitions for logs or content
}
```

Save, add, and commit this change with a message like `define apache profile`.

Let’s make a new profile class for managing Apache.

On this slide, it just declares the Apache component module, but it could easily apply other settings like setting up partitions, firewall rules, or other settings.



Provisioning a New Web Server in Puppet (6/10)

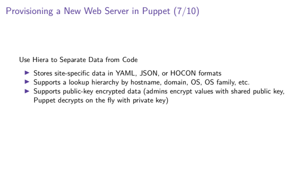
Edit the file `site-modules/role/manifests/webserver.pp` in the `puppet-control` repository. Add the following line to it below `include profile::base`:

```
include profile::apache
```

Save, add, and commit this change with a message like `update webserver role to use Apache`.

There's already a `webserver` role in the default control repository.

So we'll edit that to include our Apache profile class and push up that changes.



Provisioning a New Web Server in Puppet (7/10)

Use Hiera to Separate Data from Code

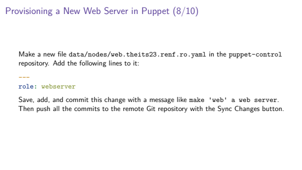
- Stores site-specific data in YAML, JSON, or HOCON formats
- Supports a lookup hierarchy by hostname, domain, OS, OS family, etc.
- Supports public-key encrypted data (admins encrypt values with shared public key, Puppet decrypts on the fly with private key)

So then we get to the data.

There's a library called Hiera linked into Puppet that can look up data relevant to a server.

The data can be factored out by hostname, operating system, any other fact provided by the client system, or common data to the whole environment.

It can store data encrypted with a shared public key, too.



Provisioning a New Web Server in Puppet (8/10)

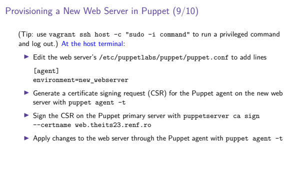
Make a new file `data/nodes/web.theits23.renf.ro.yaml` in the `puppet-control` repository. Add the following lines to it:

```
---
role: webserver
```

Save, add, and commit this change with a message like `make 'web' a web server`. Then push all the commits to the remote Git repository with the Sync Changes button.

By default, the `puppet-control` repository will look in a file stored in the `data/nodes` directory, named for the FQDN of the server.

For now, we'll add one entry for the web server's role, and commit and push that change to the Git server.



Provisioning a New Web Server in Puppet (9/10)

(Tip: use `vagrant ssh host -c "sudo -i command"` to run a privileged command and log out.) At the host terminal:

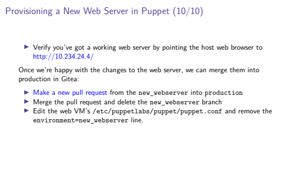
- Edit the web server's `/etc/puppetlabs/puppet/puppet.conf` to add lines

```
[agent]
environment=new_webserver
```

- Generate a certificate signing request (CSR) for the Puppet agent on the new web server with `puppet agent -t`
- Sign the CSR on the Puppet primary server with `puppetserver ca sign --certname web.theits23.renf.ro`
- Apply changes to the web server through the Puppet agent with `puppet agent -t`

Now it's finally time to build the web server.
Since we need to bounce between the web server and the Puppet server at first,
we can run commands with the `-c` flag to `vagrant ssh`.
So we need to do three things:

1. We need to make sure the web server goes into the new environment we made, so we'll edit its puppet.conf accordingly.
2. We need to enroll the system in Puppet management by sending a certificate signing request to the Puppet server and sign it.
3. And we need to apply the webserver's settings from the Puppet server by running the Puppet agent another time.



Provisioning a New Web Server in Puppet (10/10)

- Verify you've got a working web server by pointing the host web browser to http://10.234.24.4/

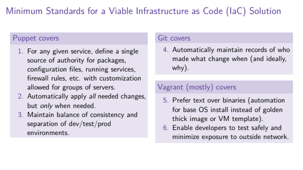
Once we're happy with the changes to the web server, we can merge them into production in Gitea:

- Make a new pull request from the `new_webserver` into `production`
- Merge the pull request and delete the `new_webserver` branch
- Edit the web VM's `/etc/puppetlabs/puppet/puppet.conf` and remove the `environment=new_webserver` line.

And if those changes applied without errors, you've now got a working, but totally minimal, Apache server available at the URL shown here.

If we needed to make any more changes, we'd push more changes to the Git server and apply them with `puppet agent -t`.

Once we're happy with how the web server is configured, we can make a pull request to merge the changes into the production environment



Minimum Standards for a Viable Infrastructure as Code (IaC) Solution

Puppet covers

1. For any given service, define a single source of authority for packages, configuration files, running services, firewall rules, etc. with customization allowed for groups of servers.
2. Automatically apply *all* needed changes, but *only* when needed.
3. Maintain balance of consistency and separation of dev/test/prod environments.

Git covers

4. Automatically maintain records of who made what change when (and ideally, why).

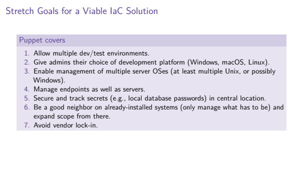
Vagrant (mostly) covers

5. Prefer text over binaries (automation for base OS install instead of golden thick image or VM template).
6. Enable developers to test safely and minimize exposure to outside network.

So did we meet the goals I set out at the beginning?

Pretty much. Puppet covers being the source of authority, making the right changes, and letting us handle differences between environments.

Git handles the recordkeeping, and Vagrant reduces the dependence on binary artifacts and lets us set up airgapped development environments.

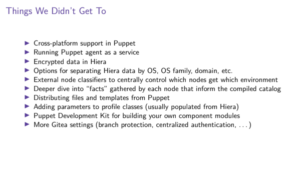


Stretch Goals for a Viable IaC Solution

Puppet covers

1. Allow multiple dev/test environments.
2. Give admins their choice of development platform (Windows, macOS, Linux).
3. Enable management of multiple server OSes (at least multiple Unix, or possibly Windows).
4. Manage endpoints as well as servers.
5. Secure and track secrets (e.g., local database passwords) in central location.
6. Be a good neighbor on already-installed systems (only manage what has to be) and expand scope from there.
7. Avoid vendor lock-in.

The stretch goals are all handled by the open source variant of Puppet, even though we didn't get to all of these features.

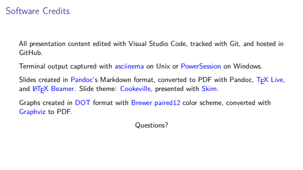


Things We Didn’t Get To

- Cross-platform support in Puppet
- Running Puppet agent as a service
- Encrypted data in Hiera
- Options for separating Hiera data by OS, OS family, domain, etc.
- External node classifiers to centrally control which nodes get which environment
- Deeper dive into “facts” gathered by each node that inform the compiled catalog
- Distributing files and templates from Puppet
- Adding parameters to profile classes (usually populated from Hiera)
- Puppet Development Kit for building your own component modules
- More Gitea settings (branch protection, centralized authentication, . . .)

So there’s a lot of things we didn’t get to, mostly in Puppet, but a bit in Gitea, too.

I can dig into any of those people are interested in later.



Software Credits

All presentation content edited with Visual Studio Code, tracked with Git, and hosted in GitHub.

Terminal output captured with asciinema on Unix or PowerSession on Windows.

Slides created in Pandoc's Markdown format, converted to PDF with Pandoc, TeX Live, and LaTeX Beamer. Slide theme: Cookeville, presented with Skim.

Graphs created in DOT format with Brewer paired12 color scheme, converted with Graphviz to PDF.

Questions?

In case anybody's interested, here's the list of software used to make this presentation. It's over-engineered and over-analyzed, but I like them.

Questions?