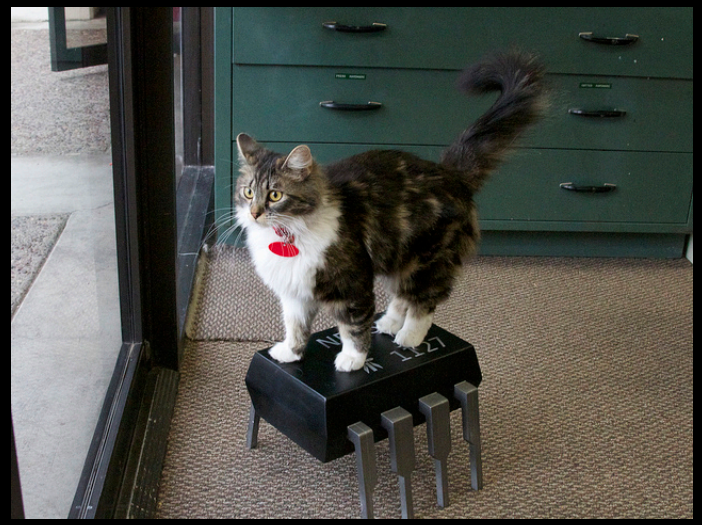


Automate Everything

Hardware as Code

Hi I'm Tim

- "Docker Captain" – Community Leader
- PHP JHB Organiser
- 17 Years Development and DevOps

- tim@haak.co

Quick Survey

- Primary laptop/desktop OS?
- Primary server OS?
- Who has set up a server?
- Who finds setting up servers fun?

First, what are the problems?

How long…

- Provision a new server?
- Rebuild all the pieces for a service?
- Work out what is on a server?
- Rollout updates to everything?
- Roll back a service or server?

Problem 1: Speed

- Setting up servers by hand can take quite a while.
- Deploying new services
  - Slow by hand
  - Error prone
- Takes to long to set up a clean server to quickly test something

Problem 2: What's really on the server?

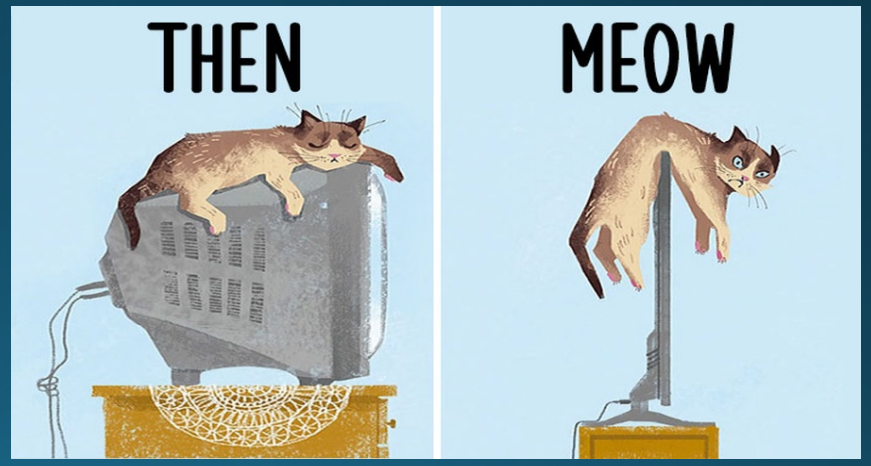
- Human memory problems
  - What was on this server again?
- Is live, staging, and testing the same?
- But it worked on the developers/my PC!

Problem 2: What's really on the server?

- Bob's sick, how did he set this up?
  - How do I bring new people on? (Documentation)
- We are error prone
  - PC's only ever do what we tell them.

Problem 3: What Changed?

- What was changed?
- When was it changed?
- Can I prove either of these?

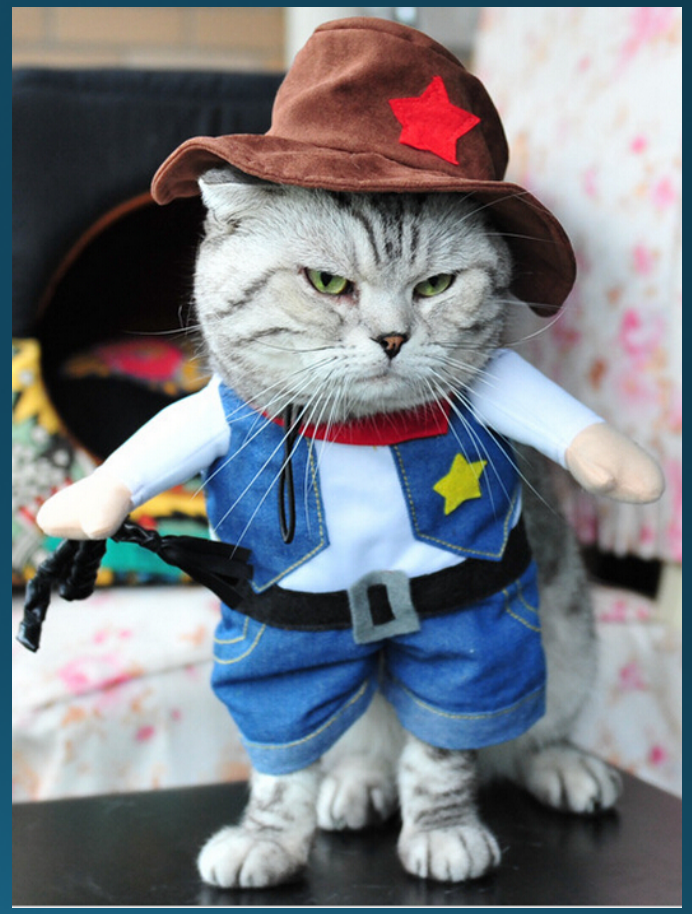


Problem 4: Repeatability

- I'm not always sure when I roll out that it will work
  - So let's not do it often

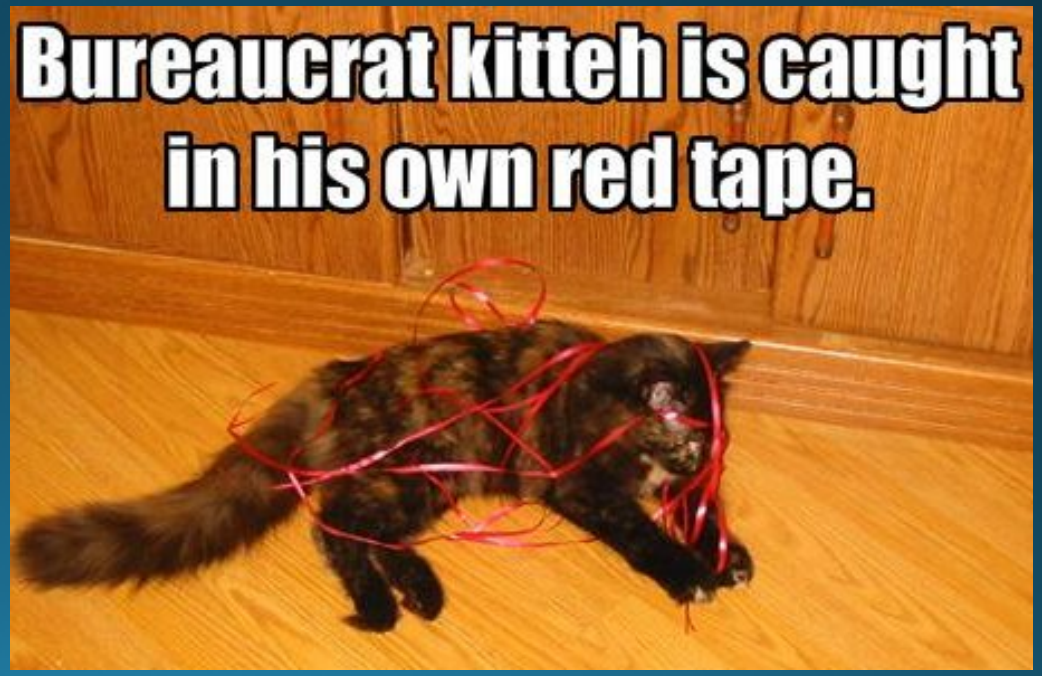
Bad patterns

- Cowboy/Cowgirl style
  - Just do it! We'll worry when it breaks…
  - I don't test, but when I do I do it on live!



Bad patterns

- Bureaucrat style
  - If it's breaking, it means we don't have enough documentation/processes.
  - We can't roll out the critical patch to fix the broken thing as change control only meets next week. (True story)



Better patterns to solve these

- Automation
  - Automate all the things!!!
  - Have tools that everyone in the team can use
    - But you know are safe and tested.
    - Have an option to easily roll back.
    - Blue/Green deployments.
  - Add web based frontends to these
    - DevOps don't just tell people to run a script.
    - Dev you have to help write the frontend to run the script.

Better patterns to solve these

- Immutability where we can
  - An **immutable object** is an object whose state cannot be modified after it is created

But how do we do this?

But how do we do this?

- On the server, hardware and operating system side
  - Hardware/Infrastructure as code
    - Describe layout in text files that are committed to version control systems
    - Continuous Integration (CI)
    - Automated testing
    - Automated building

But how do we do this?

- On the code side
  - Code is text that you commit to version control systems
  - Continuous Integration (CI)
  - Automated testing
  - Automated building

These look strangely similar

- On the server, hardware and operating system side
  - Hardware/Infrastructure as code
    - Describe layout in text files that are committed to version control systems
    - Continuous Integration (CI)
    - Automated testing
    - Automated building
- On the code side
  - Code is text that you commit to version controll systems
  - Continuous Integration (CI)
  - Automated testing
  - Automated building

How do we do this practically?

Step 1:Definition Files

Use Definition Files

- All configuration is defined in executable configuration definition files, such as shell scripts, Ansible, SaltStack, Chef, or Puppet.
- Personal preference is Ansible as it's simpler to get started.
- Commit scripts to your version control.

Cattle not pets

- Servers should be treated as cattle not pets!

Stop doing things by hand

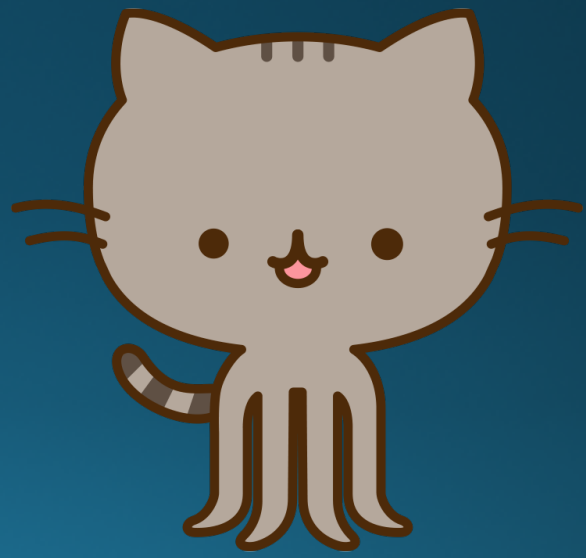
- At no time should anyone log into a server and make on-the-fly adjustments.
- Any time you do this you risk turning your server into a pet.

Bonus outcome: Documentation

- You've just documented what is on your server.
- If you need it in a more human format it can be generated from the code.
- Unlike a document a human would follow and run by hand, you can guarantee that it was done exactly as set in the code.

Version All The Things!!!

- Keep this code in version control.
- Changes can now be audited.
- Something breaking on live?
  - Just use the script to build a test server.



Continuous Deployment and Testing

- Now that you have it as code you can use all the normal code tooling to run and test.

Run It Often

- The smaller the change the lower the risk.
  - Fewer things to keep in your head.
- Ensure that your servers are always in a know state (Immutable).
- The more often you do something…The more predictable it gets.
- Doing things more often actually decreases risk.

Step 2: Build The Whole Server

Use Definition Files for Everything!

- Stop doing the initial operating system install by hand.
- Here I use Packer.
  - Handles most cloud providers and virtual servers.
- Produces images to your specification.
- Doesn't do bare metal ☹
  - Look at
    - https://github.com/google/netboot
    - https://github.com/jpetazzo/pxe
    - http://cobbler.github.io/
    - https://theforeman.org/

Packer

- Very simple config file.
- Can use most provisioning systems.
  - Ansible, Puppet, Windows Shell, PowerShell, etc…
- Use the definition files you've already created.
- Gives you the true guarantee that no human has touched the image and everything done to the server is documented.

Step 3: **Automate System Deployment**

- You know have all the pieces for your server versioned and building.
- The next step is to automate their deployment and setting up things like networking.
- Most virtualisation has API's to do this.
- My preferred tool for this is Terraform (Done by the same company as packer)

Terraform

- Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently.
- Terraform can manage existing and popular service providers as well as custom in-house solutions.

Terraform

- Description of infrastructure done via Text files.
- There's a pattern there ☺
- When you make a change it will work out the difference between what was done previously and current change.
- Can get it to return planned changes.

Terraform

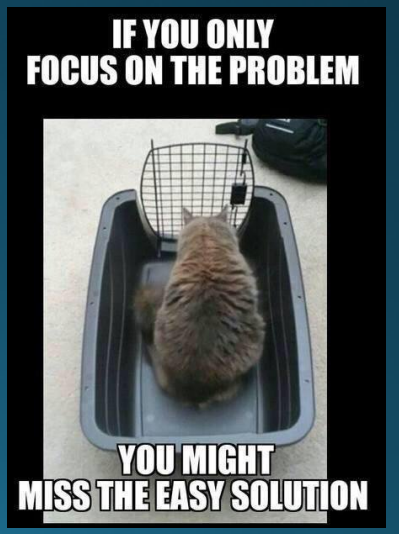
- Can set up pretty much your entire infrastructure.
  - Networking
  - DNS
  - Servers
  - etc...

Terraform

- Outputs from one section can be used in a second.
  - i.e. IP's from servers it's setup can be used to create DNS entries later.
- This was one of the simplest new technologies I've learnt recently.
  - Had something up and working in about 2 hours. (Starting from no knowledge).

Step 4: **Containerize Your Applications**

- This area is still a work in progress.
- Docker seems to be the winner.
- More so in the Linux world than Windows.
  - Though you now have Windows native containers as well.
  - Microsoft working with Docker quiet closely on this.



Container Advantages

- Immutability .
  - Images versioned.
  - Removes the “but it worked on my pc”.
  - No more difference between dev, testing, and prod.
    - Within reason i.e. DB
- Can be shared easily.
- Built off of text files (Dockerfiles).

Container Advantages

- Lower cost than virtualization.
- Base systems need fewer things installed which increases it's security.
- Allows splitting pieces up making it easier to update single piece.
- Simpler to test.
- Simpler roll back.
  - Within reason i.e. DB

What things aren't dealt with?

- Things that mutate:
  - Assets
  - Database
  - etc…
- Problems that already exist on the stack.

Data Not With Server

- Try to not keep the data with the server.
- Store images on NAS, GridFS, Ceph, etc…
  - When running in a cloud, use the provider's blob storage and similar storage.
- Relational DB's are a special case as you want them as close to the metal as possible.

My Current Pattern

My Current Pattern

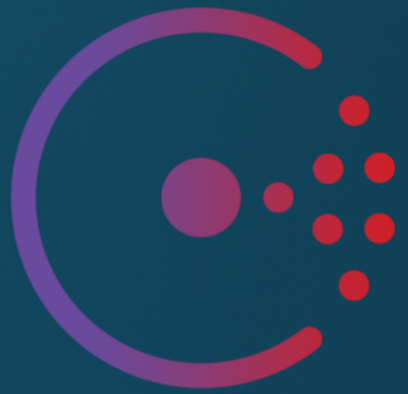
- Image build with packer.
  - Docker installed and setup.
- Terraform for setup.
  - This depends on client and complexity.
- Docker Machine for key generation and Docker setup.
  - Allows controlling Docker remotely with shared SSL keys.
- Docker for all services running on servers.
  - Allows me to replicate entire stack locally when developing.

Some Interesting Things To Look At

Chaos Monkey

- Developed by Netflix.
- Will randomly turn off services and servers.
- How sure are you about your failover and high availability.
- If you don't test you don't know.

Consul, etcd, etc.



- Distributed fault tolerant DB.
- Aimed at storing config.
- Not built to be a database but as a way to store config and system state.
- Tooling to generate configs:
  - https://github.com/hashicorp/consul-template
  - https://www.hashicorp.com/blog/introducing-consul-template/

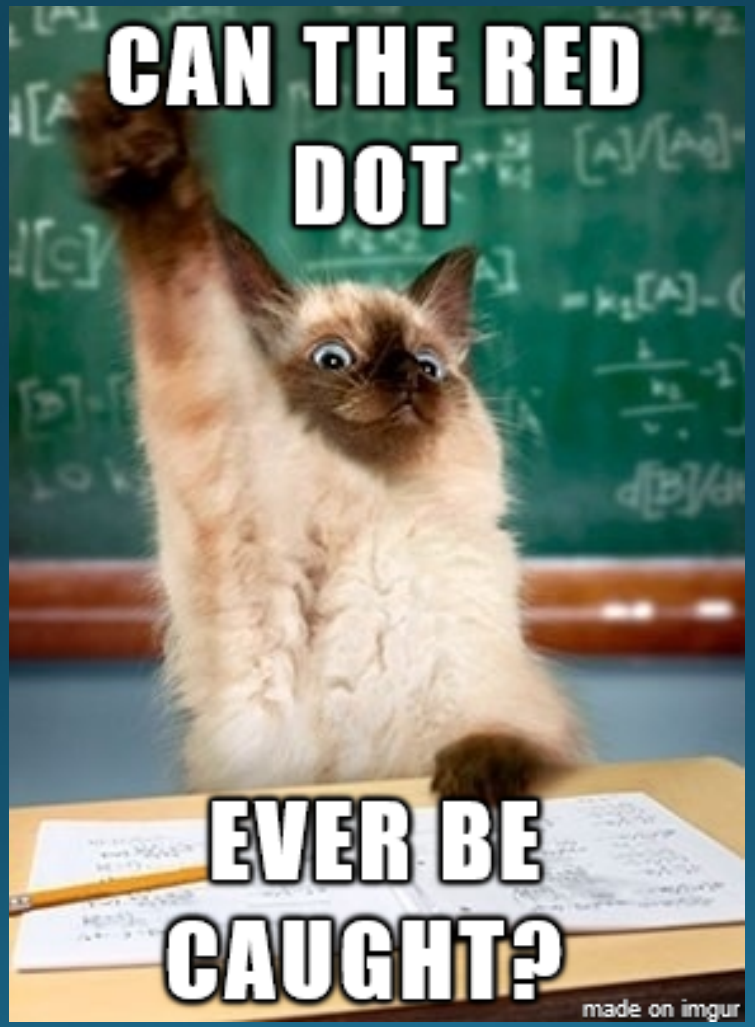
Vault

- Hashicorp, again…
- Centrally secure, store, and tightly control access to secrets across distributed infrastructure and applications.

Rundeck

- Webased workflow and job scheduler.
- Can run things like Ansible etc…
- Centralized.
- Simple way for DevOps to provide GUI to Developers.
- http://rundeck.org/

Questions?



Thanks