The (Mutable) Config Management Showdown

...TL;DR version

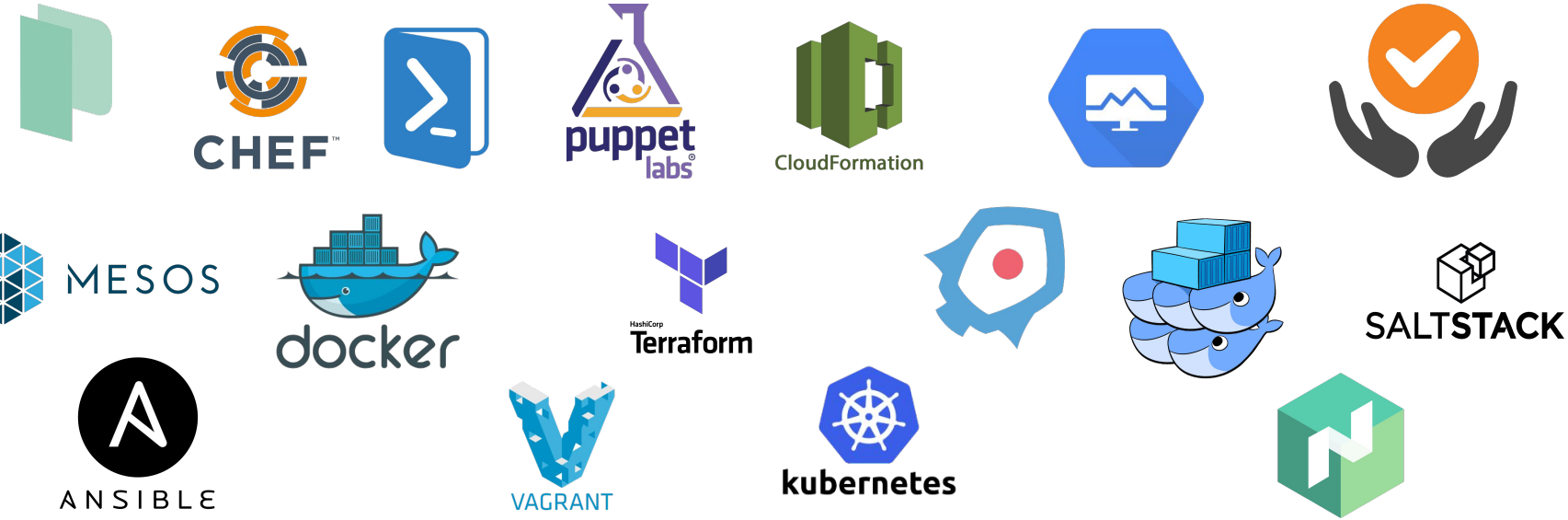
Who am I?

Bob Killen / rkillen@umich.edu / @mrbobbytables

Senior Research Cloud Administrator @ ARC-TS

http://arc-ts.umich.edu

Configuration Management Landscape



Paring things down: Mutable vs Immutable Infrastructure

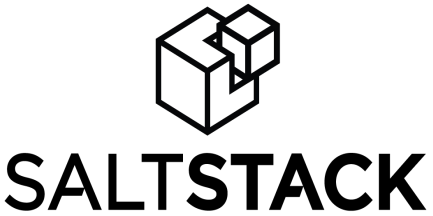
Mutable

- Object is Managed in Place.
- Object State can drift.

Immutable

- Object is built before deployment.
- Mutable systems can be used to build an immutable object.

Who are the (mutable) players?



Language:
Python

DSL:
yaml

Architecture:
Agentless

Execution:
Imperative

Language:
Ruby

DSL:
-

Architecture:
Agent/Server

Execution:
Imperative

Language:
Ruby

DSL:
PuppetDSL

Architecture:
Agent/Server

Execution:
Declarative

Language:
Python

DSL:
yaml*

Architecture:
Agent/Server*

Execution:
Declarative or
Imperative

Ansible: Overview

- Created in 2012 by Michael DeHaan
- Designed to be ‘radically simple’ (KISS)
- Uses INI based file format for host inventory and group management.
- Executes ‘**playbooks**’ consisting of **tasks**, **roles**, or other **playbooks**.
- Secrets managed through ‘**ansible-vault**’.
- Central Server/API available through Ansible Tower / AWX
- Large Users:
  - Atlassian
  - Capital One
  - Verizon

Ansible: Task Examples

```
- name: Create Mitchell
  user:
    name: mitchell
    uid: 1000
    shell: /bin/bash
    home: /home/mitchell
```

```
- name: Start httpd
  service:
    name: httpd
    state: started
    enabled: true
    when: ansible_distribution == 'Ubuntu'
```

Ansible: Anatomy of a Run

1. **Host Inventories**, **Playbooks** and **Roles** are stored on server.
2. A **playbook** is called referencing members of the **host inventory.**
3. Ansible server connects to hosts via ssh and copies the needed python scripts (**modules**) to complete the **playbook**.
4. The **playbook** is executed on the host with the results of each task being returned to the server before the next task is executed.

Ansible: Pros & Cons

Pros:

- SSH-Based
- Low Barrier of entry
- Supports a variety of devices
- Easy to extend with Modules
- Good aws support

Cons:

- Master must be able to reach target
- DSL is extremely limiting
- Slow with a large number of hosts
- Very little windows support

Chef: Overview

- Created in 2009 by former AWS engineers
- No DSL, fully Ruby based with an erlang powered API server.
- Full rest/json api
- Compliance automation checks built-in
- Executes '**runlists**' containing **roles**, and **recipes**
- Secrets are managed via encrypted '**data-bags**'
- Large Customers:
    - Facebook
    - Riot Games
    - Microsoft

Chef: Resource Example

```
user 'mitchell' do
  uid '1000'
  gid '1000'
  home '/home/mitchell'
  shell '/bin/bash'
end
```

```
if node[:platform] == ‘ubuntu’
  service 'httpd' do
    action [ :enable, :start ]
  end
end
```

Chef: Anatomy of a Run

1. Client rebuilds node information via **ohai**
2. Client contacts server and pull down **node object** containing **run list**.
3. Client pulls down **cookbooks** contained in **run-list.**
4. **Node object** is rebuilt and the run is compiled
5. **Run-list** is executed and node is **converged**.
6. Updated **node object** with results is returned to server.

Chef: Pros & Cons

Pros:

- Mature
- Very easy to learn if coming from a programming background.
- Massive amount of community contributions and tools.
- Tightly integrated with AWS Opswork
- Strong Windows Support

Cons:

- Production level deployment is a challenge if not using managed solution.
- Steep learning curve if not familiar with Ruby.
- Secret Management is problematic without 3rd party tools

Puppet: Overview

- Created in 2005 by Luke Kanies.
- json like DSL describing resources and their dependencies.
- Client/Server communicate over TLS
- Full rest/json API
- Generates a DAG to determine what clients execute
- Large Customers:
  - CERN
  - Walmart
  - Wikimedia Foundation

Puppet: Resource Example

```
user { 'mitchell':
       ensure    => present,
       uid       => '1000',
       gid       => '1000',
       shell     => '/bin/bash',
       home      => '/home/mitchell'
}
```

```
if $::osfamily == 'Ubuntu' {
  service { 'httpd':
    ensure => running,
    enable => true
   }
}
```

Puppet: Anatomy of a Run

1. Client rebuilds **fact index** by running **facter**
2. Client checks in with server.
3. Desired agent state **(catalog)** is calculated on server and produced a DAG
4. **Catalog** is sent to client and is evaluated against actual state
5. Result is returned to server

Puppet: Pros & Cons

Pros:

- Mature
- Very Simple DSL
- Strong Windows Support
- Very good UI
- Strong Reporting Functionality

Cons:

- Difficult to troubleshoot
- Customizing it requires Ruby knowledge
- Can be complicated to scale
- Can be slow

SALTSTACK

Salt: Overview

- Created in 2011
- ‘Event-Driven’ orchestration, designed to execute tasks quickly and in parallel
- ‘**Grain**’ or fact driven inventory system
- Mainly uses yaml and jinja, but supports many other templating engines and data sources.
- Full json/rest API
- Large User:
  - Cloudflare
  - LinkedIn
  - Lyft

SALTSTACK

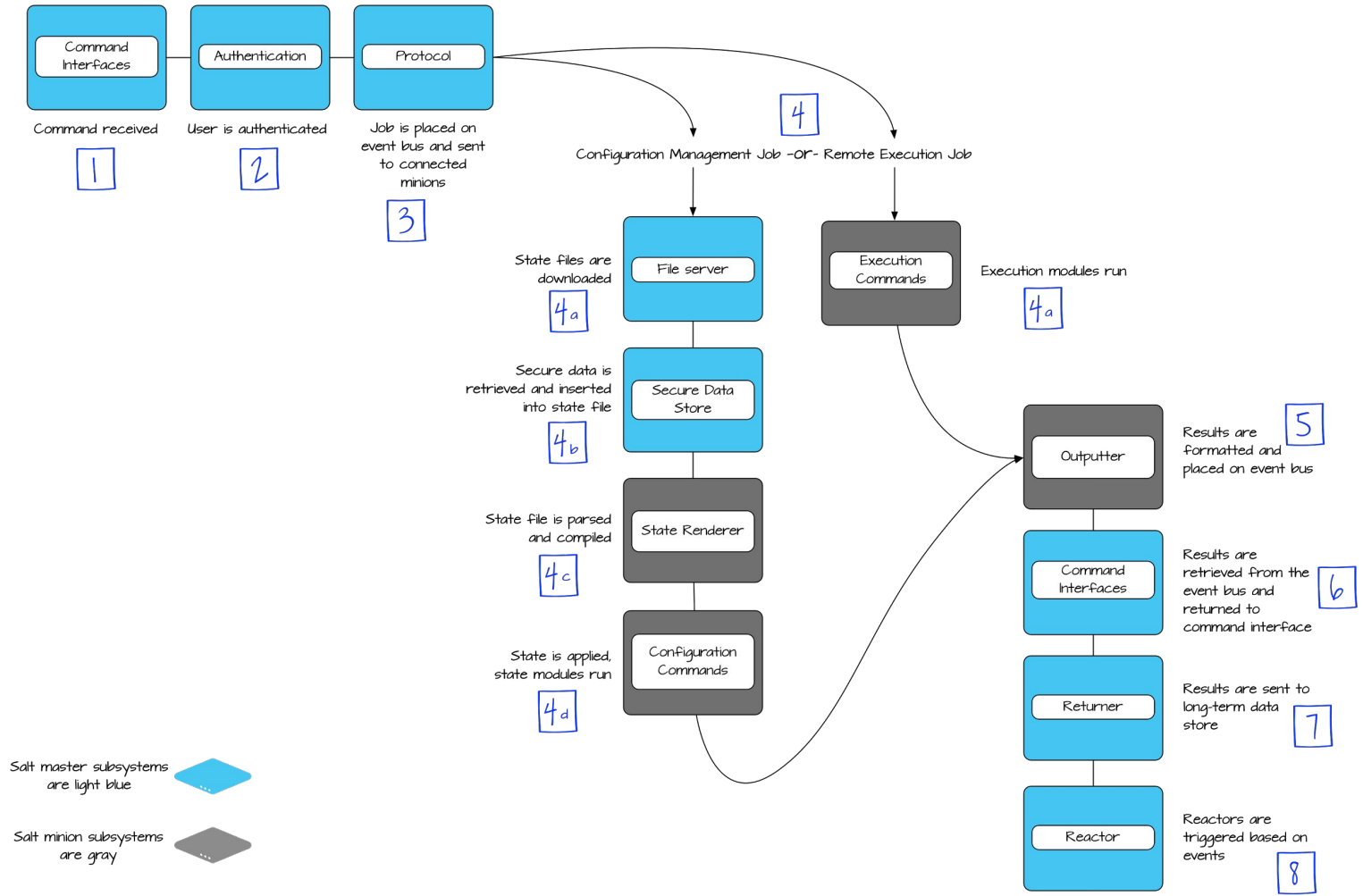
Salt: State Examples

```
Create Mitchell:
  user.present:
    - name: mitchell
    - uid: 1000
    - gid: 1000
    - shell: /bin/bash
    - home: /home/mitchell
```

```
{% if grains['os'] == 'Ubuntu' %}
Start httpd:
  service.running:
    - name: httpd
    - enabled: true
{% endif %}
```

Salt: Anatomy of a Run

...it’s complicated


Command Interfaces
Authentication
Protocol
Command received
1
User is authenticated
2
Job is placed on event bus and sent to connected minions
3
4
Configuration Management Job -or- Remote Execution Job
State files are downloaded
4a
File server
Secure data is retrieved and inserted into state file
4b
Secure Data Store
State file is parsed and compiled
4c
State Renderer
State is applied, state modules run
4d
Configuration Commands
Execution Commands
Execution modules run
4a
Outputter
Results are formatted and placed on event bus
5
Command Interfaces
Results are retrieved from the event bus and returned to command interface
6
Returner
Results are sent to long-term data store
7
Reactor
Reactors are triggered based on events
8
Salt master subsystems are light blue
Salt minion subsystems are gray

SALTSTACK

Salt: Pros & Cons

Pros:

- Extremely flexible
- Powerful inventory system
- Built in event-engine
- Highly scalable architecture
- Python 3.x support

Cons:

- Exceedingly high learning curve
- Difficult to debug
- Event-bus nature of the system can make it hard to secure in a multi-tenant fashion

Wrapping up

- Use Ansible when…
  - Need a CM system up and going yesterday
  - Your inventory and applications are fairly static and don't require much in the way of conditional cases
- Use Chef if…
  - You're coming from a programming background
  - Tight cloud integration is a boon
- Use Puppet when…
  - Have little to no programming experience
  - UI and reporting are important
- Use Salt if…
  - You have complex orchestration tasks
  - Event driven infrastructure makes sense for your organization