Building a Test-Driven Network Infrastructure

Tyler Christiansen

@supertylerc
Introductions
Disclaimer

This presentation does not reflect the views or opinions of my employer or anyone else. They're mine. They're probably wrong.
Who am I?

- Network Architect
  - I make *some* decisions
    - Hardware
    - Logical and physical designs
- Aspiring Pythonista
- Lover of Regular Expressions
Where am I?

- Twitter: @supertylerc
- GitHub: @supertylerc
- GitLab: @supertylerc
Who are you?

- Network engineers/administrators/technicians?
- Linux engineers/administrators/technicians?
- Software engineers/developers?
What Isn't This?

- How to Install <software>
- How to Configure <protocol>
- How to Design <system>
- How to […]
What Is This?

• An Exploration of Problems and Potential Solutions
• An Introduction to CI/CD Practices in Networking
Misalignment
Business vs. Network

• Networks are:
  – Frequently Complex
  – Generally Slow to Adapt
  – Often 100% Production

• "Everybody has a testing environment. Some people are lucky enough to have a totally separate environment to run production in."
Business vs. Network

• Businesses Need:
  – Transport of Services
  – Rapid Response and Agility
  – Reliability and Stability
Networking is a Little Behind

• Minimal Virtualization of Networks
  – RAM Gluttons: 8-16GB RAM for one VM
  – Limited Data Plane: ASICs

• Limited Automation Tooling
  – Ansible
  – SaltStack
Networking is a Little Behind

• Less Familiarity with Software Engineering
  – Python is Gaining Ground
  – CI/CD are Nearly Foreign

• View of Networks is Skewed
  – Protocols: General view of network professionals
  – Services: This is what we really enable
Aligning Networking
Networks Transport Services

• View Configuration as Services
  – Not per device
  – Full configuration to support a service over the base of the underlying network
Software Engineering Principles are Critical

- The basics of variables and flow control are necessities
- Modularity is your friend
- Pipelines are the foundation of this entire talk
Definitions

• **Job**: a series of instructions
  – Sequential
• **Stage**: a collection of jobs
  – Nonsequential
• **Pipeline**: a collection of stages
  – (Usually) Sequential
Example Pipeline

Source: https://docs.gitlab.com/ee/ci/pipelines.html
Network Pipelines

• Same as Software Pipelines
  – Use a combination of tools to orchestrate the pipeline
    • GitLab
    • Vagrant
    • Python
    • SaltStack
      – or Ansible or any other "config management" system)
Example Network Pipeline

- Lint: lint_yaml
- Unit: unit
- Integration: iosxr, junos
- Review: review_junos
---
stages:
- lint
- unit
- integration
- review

job: &job
  stage: integration
  tags: ['kitchen']
  before_script:
    - python --version
  script: './test/scripts/integration.sh'

junos:
  <<: *job
  variables:
    VENDOR: junos

iosxr:
  <<: *job
  variables:
    VENDOR: iosxr

unit:
  <<: *job
  tags: ['docker']
  stage: unit
  image: supertylerc/salt-masterless-test
  script: './test/scripts/unit.sh'

lint_yaml:
  <<: *job
  stage: lint
  tags: ['docker']
  image:
    name: boiyaa/yamllint
    # This entrypoint can be blank in 10.x
    # entrypoint: ['/bin/sh', '-c']
  script: './test/scripts/lint.sh'
GitLab CI Configuration Example

```
 junos_review:
  <<: *job
  stage: review
  environment:
    name: review/$CI_COMMIT_REF_NAME
    on_stop: stop_junos_review
  only:
    - branches
  except:
    - master
  variables:
    VENDOR: junos
    REVIEW: 'true'

 stop_junos_review:
  <<: *job
  stage: review
  environment:
    name: review/$CI_COMMIT_REF_NAME
    action: stop
  variables:
    GIT_STRATEGY: none
  script:
    - cd test; vagrant destroy -f salt junos
  when: manual
```
• Validate Syntax and Models
  – Syntax: yamllint, xmllint, jsonlint, etc.
    • Don't go further if something breaks the rules!
  – Models/schemas: yamale, xsd, kwalify, jsonschema, etc.
    • Stop if incorrect data is entered!
      – example: customer VLAN ranges are over 3000, but someone entered a VLAN id of 1003
Unit Tests

- Test Discrete Features
  - Use mock or fake data
    - Expected configuration output vs. actual configuration output
  - Tests should be fast and have a high confidence of success
    - Don't bring up a virtual router during this stage
    - Ensure your tests are relevant to the changes being made
Unit Tests

- Tests written in Python using pytest and testinfra
  - Take advantage of testinfra's salt capabilities
  - Since it's a container or Linux VM, fake the host's OS to get the correct configuration for a network device
import json
import re

import pytest

@pytest.mark.parametrize("router", [
    'junos',
    'nxos',
    'iosxr'
])
def test_lint_ntp_state(host, router):
    host.salt('grains.set', ['os', router], local=True)
    assert host.salt('state.show_sls', ['ntp.test_netconfig'], local=True)

@ pytest.mark.parametrize("router", [
    'junos',
    'nxos',
    'iosxr'
])
def test_ntp_state_test(host, router):
    host.salt('grains.set', ['os', router], local=True)
    assert host.salt('state.apply', ['ntp.test_netconfig', 'test=true'], local=True)
Unit Tests

```python
@pytest.mark.parametrize("router", [  
    'junos',  
    'nxos',  
    'iosxr'
])

def test_state(host, router):
    host.salt('grains.set', ['os', router], local=True)
    host.salt('saltutil.refresh_pillar', local=True)
    host.salt('state.apply', ['ntp.test_netconfig', 'exclude=[{"id": "file.remove"}]'], local=True)
    expected = [x.rstrip() for x in host.file('/tmp/mock/%s_unit_ntp.mock % router).content_string.strip().split('\n')]  
    actual = [x.rstrip() for x in host.file('/tmp/__salt__ntp_salt.example.com.txt').content_string.strip().split('\n')]  
    assert expected == actual
    host.salt(  
        'state.apply',  
        ['ntp.test_netconfig', 'exclude=[{"id": "file.read"}, {"id": "oc_ntp_netconfig_test"}]'],  
        local=True
    )
```
Integration Tests

• Apply changes to virtual routers
  – Be prepared for longer test times!
  – Test as close to your production software version(s) as possible
  – You can have many of these running in parallel as long as you have the resources

• Validate changes to virtual routers
Integration Tests

• Vagrant controls the VMs
• pytest and testinfra provide the testing framework
• Mocks of configurations and states ensure live network device VM data matches expectations
• A bash script ties them together
import json
import re

import pytest

@pytest.mark.parametrize('router', [
    pytest.param('junos', marks=pytest.mark.junos),
    pytest.param('iosxr', marks=pytest.mark.iosxr)
])
def test_ntp_state(host, router):
    output = None
    with host.sudo():
        output = host.check_output('salt %s state.apply % router' % router)
    assert re.search(r'\s+Failed:.*', output)
    assert re.search(r'\s+Succeeded:.*\(changed=1\)\n', output)
    assert re.search(r'\s+Configuration changed:.*\n', output)
    # can probably be replaced with getattr
    _test_ntp_map(router)(output)

def _test_ntp_map(router):
    return {
        'junos': _test_junos_ntp,
        'iosxr': _test_iosxr_ntp
    }[router]

def _test_junos_ntp(output):
    assert re.search(r'\s+peer 172.17.19.2\n', output)
    assert re.search(r'\s+server 172.17.19.1 prefer\n', output)
def _test_iosxr_ntp(output):
    assert re.search(r'\s+peer 172.17.19.2\n', output)
    assert re.search(r'\s+server 172.17.19.1 prefer\n', output)

@pytest.mark.parametrize("router", [
    pytest.param('junos', marks=pytest.mark.junos),
    pytest.param('iosxr', marks=pytest.mark_iosxr)
])

def test_ntp_is_applied(host, router):
    with host.sudo():
        output = host.check_output('salt %s state.apply' % router)
    assert re.search(r'Succeeded:\s+1\n', output)
    assert re.search(r'Failed:\s+0\n', output)
    assert re.search(r'Comment:\s+Already configured.\n', output)

def _router_ntp_config_command(router):
    return {
        'junos': 'sh conf system ntp',
        'iosxr': 'sh run ntp'
    }[router]
@pytest.mark.parametrize("router", [
    pytest.param('junos', marks=pytest.mark.junos),
    pytest.param('iosxr', marks=pytest.mark.iosxr)
])
def test_state(host, router):
    cmd = _router_ntp_config_command(router)
    output = None
    with host.sudo():
        output = json.loads(host.check_output('salt --output=json %s net.cli "%s" % (router, cmd)))
    output = output[router]['out'][cmd].strip()
    assert output == host.file('/vagrant/mock/%s_ntp.mock' % router).content_string.strip()
Integration Tests

```python
@pytest.mark.parametrize("router", [
    pytest.param('junos', marks=pytest.mark.junos),
    pytest.param('iosxr', marks=pytest.mark.iosxr)
])
def test_state(host, router):
    cmd = _router_ntp_config_command(router)
    output = None
    with host.sudo():
        output = json.loads(host.check_output('salt --output=json %s net.cli "%s" % (router, cmd)))
    output = output[router]['out'][cmd].strip()
    assert output == host.file('/vagrant/mock/%s_ntp.mock' % router).content_string.strip()
```
Integration Tests - Vagrant

- **vagrant-triggers** plugin is useful for network devices that can't use provision during the normal Vagrant cycle.
- An extra network is used on all VMs to put the Salt Proxy Minions on the same management network as the network VMs.
- Additional shell scripts set up base network connectivity from the Salt infrastructure to the network devices.
Review Environment

• Bring up the exact same environment as was used for the Integration Tests
• As a final gate before something is released, allow an engineer to log into the virtual environment and inspect its behavior for any additional oddities
  – Ideally anything not caught by integration tests is noted during this stage and added to integration tests
More Complex Topologies

- Vagrant isn't the only way to control VMs
- GNS3 and other simulation/emulation tools have APIs to bring complex and resource-intensive topologies to life
Gating
Branching Strategy

- Have a branch for development, staging, and production
- Only allow changes to flow from development to staging to production
- Do not allow direct changes to the staging or production branches
Development Branch

• This is really just a feature branch
• Short-lived and concerned only with the changes being made in a specific pull request
Staging Branch

- This is the branch into which the development branches get merged.
- This longer-lived branch is concerned with combining multiple development features into a single point-in-time state.
- Tag it before you want to release it to production.
Staging Environment

• The staging environment should consist of a tagged version of the staging branches of all features (repositories)

• A suite of automated tests should exist that are designed to validate the staging environment end-to-end across many features
Staging Environment

- The staging environment can be physical or virtual
  - If virtual, ensure you're taking advantage of APIs of your systems to speed up provisioning and decommissioning of the environment
- Always start from scratch!
Production Branch

- Once the staging environment has been thoroughly tested by automated systems and human review, merge the branch from staging to the production branch.
- From this point, an automated system could deploy the changes from the production branch directly to your network devices.
Production Deploy

- SaltStack has schedules
- Run high states on a schedule to always deploy the production branch
  - Temporarily disable schedules when implementing workarounds or emergency fixes until they make their way back into configuration management
- Run post-production deploy automated tests to validate your production network
• Click to edit text
  – Second level
    • Third level
      – Fourth level
        » Fifth level
Summary
Tools

- Python
- SaltStack
- Jinja2
- YAML
- GitLab
- Vagrant
- GNS3
- EVE-NG
- VIRL
- OpenStack
Pipelines

- Series of tests
- Start with short, high value tests
- Progress to increasingly complex and longer tests in new stages
- Fail early, fail fast!
- Spin up review environments!
Branching

- Protect production
  - Don't allow changes not originating from your config management
    - Automatically revert them to the correct state according to your config management system
  - Don't allow changes directly to the production branch of your config management system
    - Gate them through development and staging first!
Questions?