Network Automation at the University of Michigan
Software Architecture
Eric Boyd, Kris Steinhoff

INFORMATION AND TECHNOLOGY SERVICES
UNIVERSITY OF MICHIGAN
Overview

- U-M Network Background Information
- Network Automation Strategy
- Network Automation Execution
Organic Growth of U-M Network Tools:
- A broad range of tools
- Commercial, open source, dead-end forks of open source, in-house developed
- Each tool fulfills a particular niche

Tool fragmentation resulted in technical debt:
- Legacy and in-house tools support lapsed
- Open source tool releases lag
- Overlaps in tool versions and functionality
- “Source of authority” fragmented across tools
Background

University of Michigan (U-M) Network

200+ Buildings ◆ 4 Data Centers

Thousands devices
- 4 core, 3 border, 227 distribution layer routers
- 3,083 access-layer switches
- ~ 14,000 wifi APs

Major device vendors: Cisco and Juniper
- Dozens of device models, dozens of OS versions: 125+ combinations
- Traditional Cisco three-tier architecture
- Elements of enterprise and service-provider networks: spanned VLANs + VRF
Significant overlaps in functionality, tool interdependencies, and staff turnover have made our current tool chain difficult to maintain.
Strategy

Network Transformation Strategy Goals

- Transition to FEWER, MORE INTELLIGENT TOOLS to:
  - Proactively alert, even react to problems in the network
  - Harden the network
  - Reduce staff time needed to detect and respond to incidents
  - Improve the quality and consistency of network management and configuration

- Deploy AUTOMATION FRAMEWORK built out to become, for most basic tasks, as trustworthy as the switch itself
Implementation Strategy

Networks are currently managed with the existing collection of tools and processes.
Our approach is to build a second software stack in parallel, implement the minimum synchronization between the two stacks, build new networks with the new stack, and then migrate existing networks to it.

Existing system is left in place and the **new system is built in parallel**:

- ETL process loads data from the old system into the new
- New system is validated and tested
Our approach is to build a second software stack in parallel, implement the minimum synchronization between the two stacks, build new networks with the new stack, and then migrate existing networks to it.

Existing system is left in place and the new system is built in parallel:

- ETL process loads data from the old system into the new
- New system is validated and tested
- Subsets of networks are gradually moved to the new system
Our approach is to build a second software stack in parallel, implement the minimum synchronization between the two stacks, build new networks with the new stack, and then migrate existing networks to it. Existing system is left in place and the **new system is built in parallel**:

- ETL process loads data from the old system into the new
- New system is validated and tested
- Subsets of networks are gradually moved to the new system
- Existing system runs in parallel while confidence is built in the new system
- Continue to add functionality and networks to the new system.
Implementation Strategy

End State: All networks migrated to the new system

- New system
  - All functionality replicated from the old
  - Sufficient trust built in the new service
- Old system shut down
Strategy

Network Transformation Strategy Principles

● **Establish central source of configuration authority** for all devices
  ○ Make network changes in the SoA rather than the device(s).

● **Software reads configuration data and implements changes programmatically.**
  ○ Network industry seems to be settling on Python, so we naturally gravitated towards Salt and/or Ansible.

● **Utilize open-source tools conventionally**
  ○ Refactor existing processes rather than implement U-M-specific versions.
  ○ If changes are necessary, contribute them upstream so U-M does not run special software.
Execution
Execution

Loosely Coupled Components

Source of Authority

Translate data to config templates

Push template to network device

netbox

Jinja

Junos

Cisco
Execution

New Data Center

Goal: provision a small leaf-spine data center fabric entirely using automation

- **Step 1**: Datacenter inventory, IPAM, and topology defined in Netbox:
  - Network device roles, physical connections between devices. IP addressing/interface assignments...

- **Step 2**: Templates written to map data from Netbox into device-specific configuration:
  - Use some data directly, e.g. interface IP addresses, VLAN IDs...
  - Other configuration implicit: e.g. for a device with the “spine” role:
    - for each interface connected to a device with the “leaf” role configure IS-IS on that interface (underlay),
    - configure a route-reflector client for the IP address on the remote end of that connected interface (overlay).

- **Step 3**: Salt renders complete device configuration, pushed to device via NAPALM load-replace
  - Load-replace ensures that the automation process generates and applies all device configurations.
Use several existing sources of operations data, including:
  ○ NetInfo (home-grown tool)
  ○ Custom tables in a Remedy database

Developed a suite of python scripts to extract data from sources (NetInfo and Remedy), and load them into NetBox.

Automatically transferring thousands of records and relationships.

Work to do: validate accuracy, and source data quality.
With this flow, the older systems are the source of authority.

But NetBox will be where automation tools (Salt / Ansible) will pull data from.
We have many existing scripts that pull data from sources like NetInfo and Remedy into other tools like BlueCat.
**Option: Rewrite import script**
Create a new import script that uses the NetBox APIs to pull data and import into systems like BlueCat.
Option: Reverse the flow
Implement an export script to move data from NetBox to the existing databases.
We will **strongly** prefer to rewrite the import script, but will consider reversing the flow.

This will depend partly on the complexity of the systems involved and how long we expect them to continue to exist.
Execution
Salt Contribution

● Created a Salt “pillar” (data source middleware) to pull data from the NetBox API.

● Our Pull Request was approved and merged in March. And has since had additional changes contributed by the community.

● This pillar will be included with the next release of Salt (fluorine).
Launching a project to replace an existing tool: “DCT”.

DCT was built to allow departmental IT staff to make (some) configuration changes to switch ports.

Going to replace this with an open-source tool with a modular architecture to support different backends and devices.

Discussing use cases and architecture with the University of Wisconsin.

Looking for collaborators!
Execution

GitLab

- Collaborating across our Infrastructure group to stand up GitLab server.

- We are using this extensively to manage:
  - code for software projects
  - some device configurations
  - server deployment configurations (Ansible)

- Initial target has been central IT infrastructure group, but we are expanding to provide this as a service to campus, including research, and teaching and learning uses.
GitLab

- Code review process as part of merge requests.

  **“Maintainer”**
  - Review Merge Request
  - Approve?
  - yes → Merge branch
  - no

  **“Contributor”**
  - Create Branch
  - Implement
  - Merge Request
  - Approve?
  - yes → Merge branch
  - no

- CI/CD
  - Code linting / Syntax validation
  - Automated Testing
Execution

Skills

- Conducting internal workshops to help colleagues build skills and experience with the technologies and approaches we need to adopt.
  - Git
  - Ansible / Salt
  - NetBox
  - Python
  - Software development practices
Questions?

Eric Boyd <ericboyd@umich.edu>

Kris Steinhoff <steinhof@umich.edu>