Introduction to Network Automation using Ansible

Ganesh B. Nalawade

Principal Software Engineer

Ansible Engineering

Github/IRC: ganeshrn

Twitter: @ganesh634

Agenda

- Ansible overview
- Ansible module execution Linux v/s Network host
- Ansible Collections overview
- Ansible Network Collections
- Fundamental modules
- Resource module
- Operational state management modules
  - Parsing operational state to structured data
  - Validating structured data against a criteria

Ansible overview

SIMPLE

Human readable automation

No special coding skills needed

Tasks executed in order

Get productive quickly

POWERFUL

Gather Information and Audit

Configuration management

Workflow orchestration

Manage ALL IT infrastructure

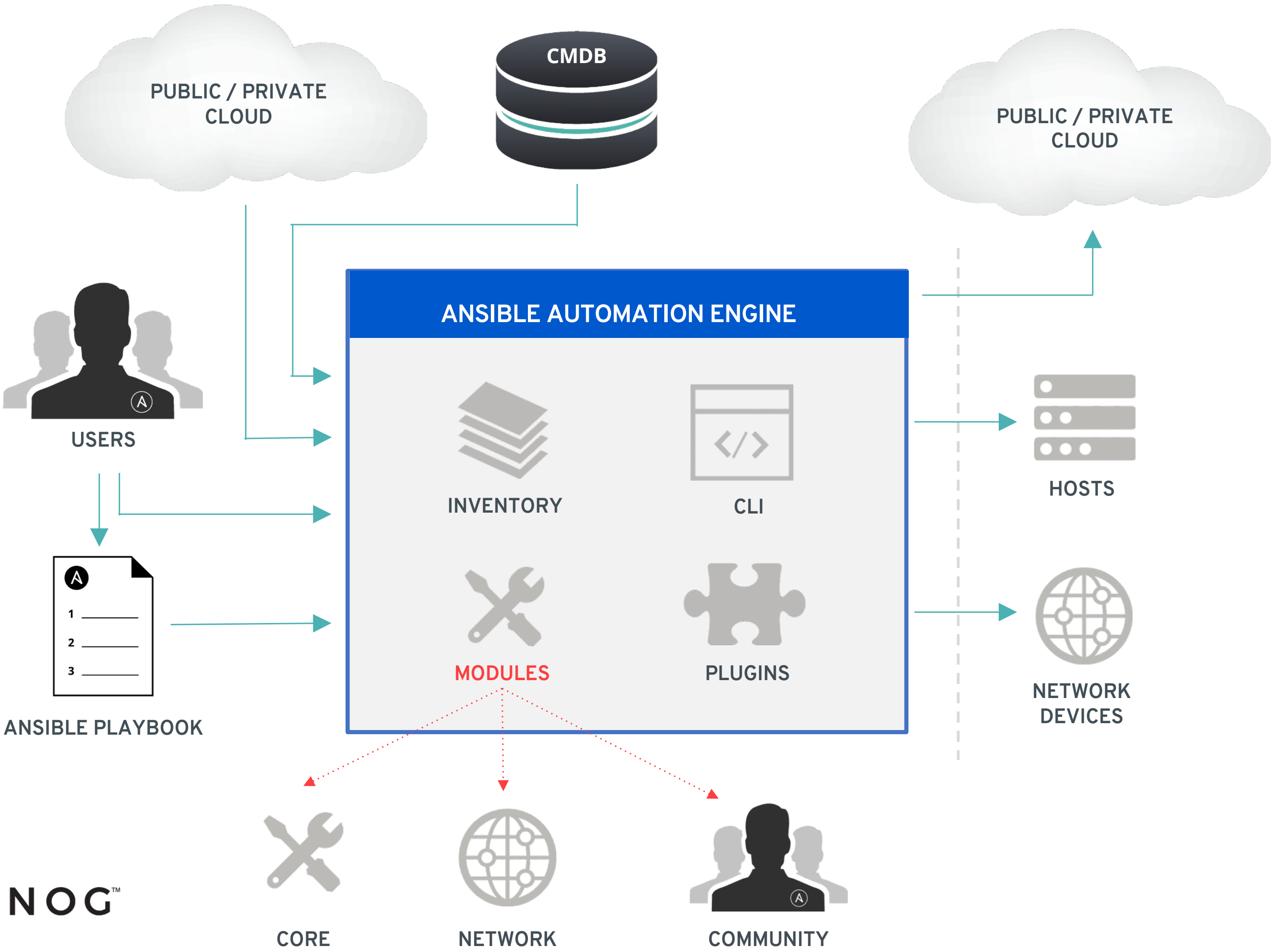
AGENTLESS

Agentless architecture

Uses OpenSSH and paramiko

No agents to exploit or update

More efficient & more secure



PUBLIC / PRIVATE CLOUD

CMDB

PUBLIC / PRIVATE CLOUD

USERS

ANSIBLE PLAYBOOK

ANSIBLE AUTOMATION ENGINE

INVENTORY

CLI

MODULES

PLUGINS

HOSTS

NETWORK DEVICES

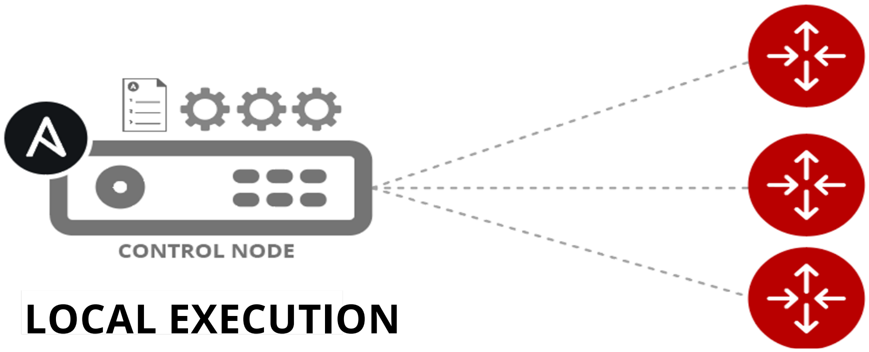
CORE

NETWORK

COMMUNITY

Ansible module execution

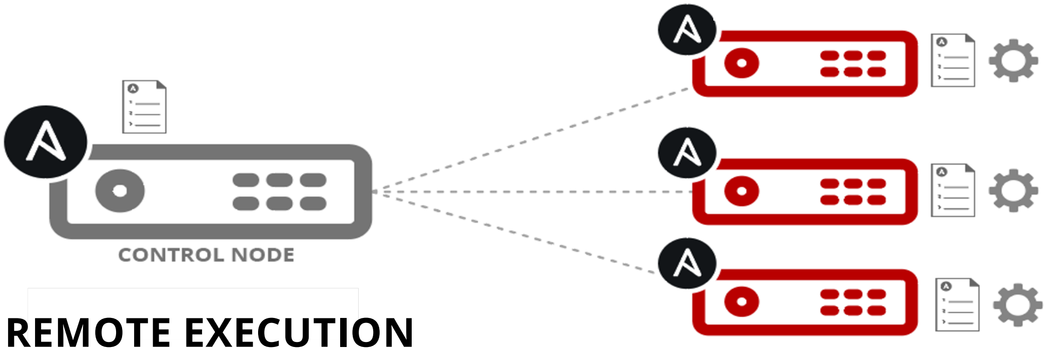
Module code is executed locally on the control node



LOCAL EXECUTION

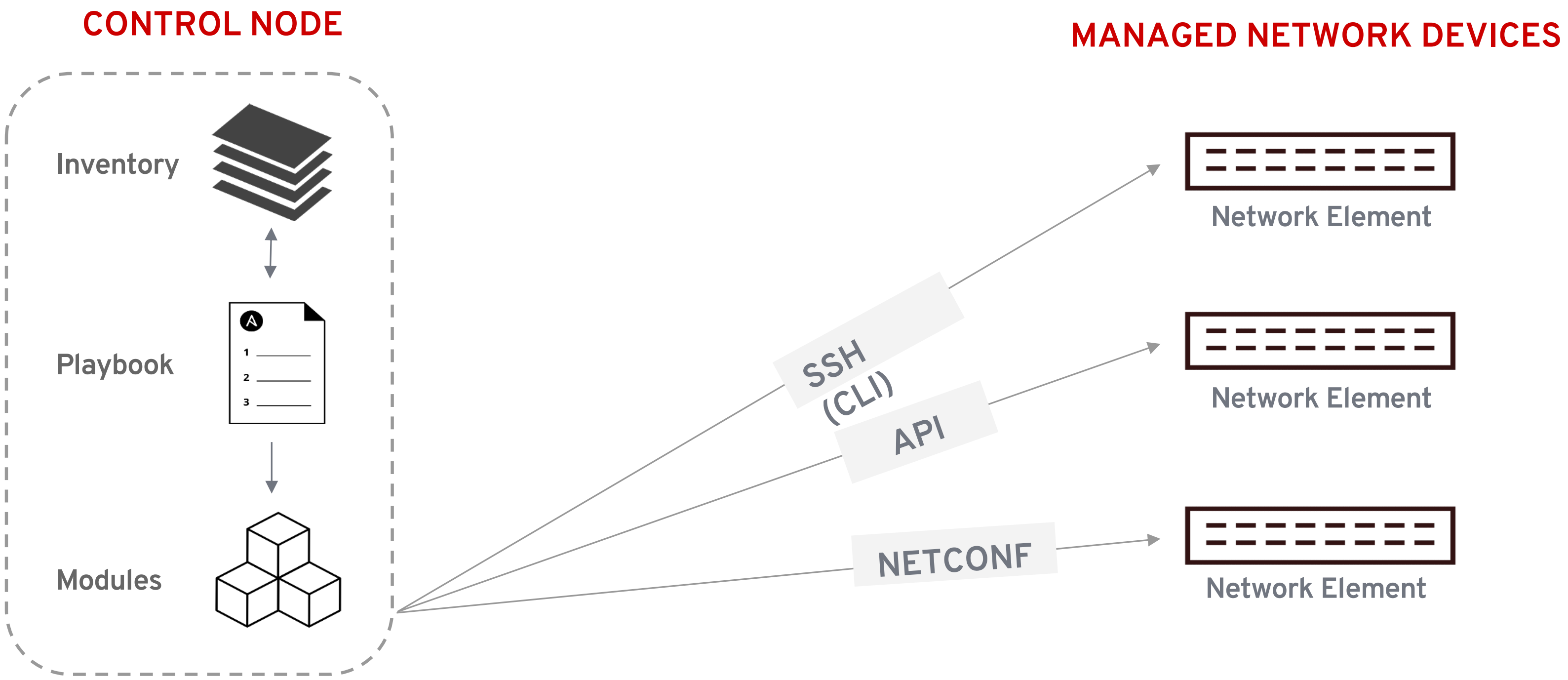
NETWORKING DEVICES

Module code is copied to the managed node, executed, then removed



REMOTE EXECUTION

LINUX/WINDOWS HOSTS



Managed Nodes (Inventory):
A collection of endpoints being
managed via SSH or API.

Control Node:
Any client system (server, laptop,
VM) running Linux or Mac OSX

Modules:
Handles execution of remote
system commands

Ansible collections overview

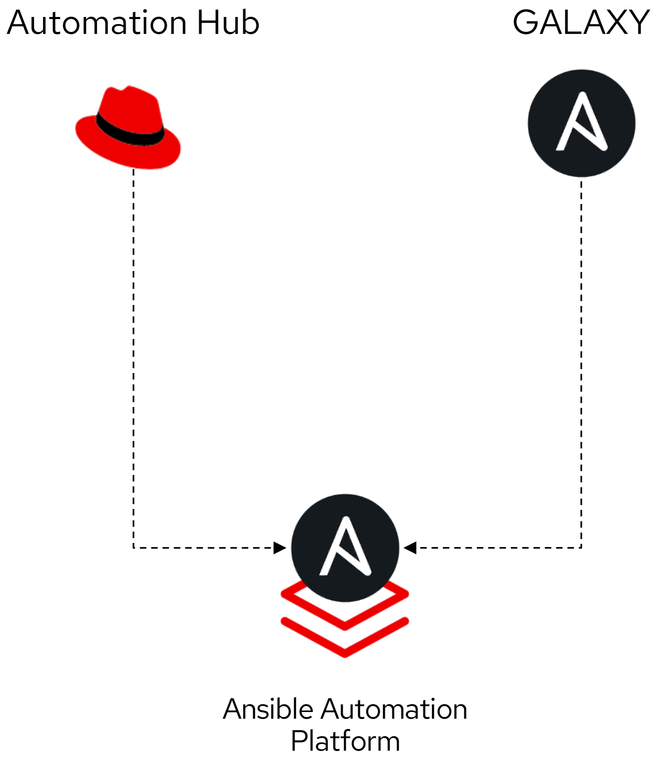
NANOG™

Ansible Collection

Simplified and consistent content schema

- A standardized way to organize and package Ansible content (roles, modules, module utilities, plugins, documentation)
- Semantic versioning
- Portable and flexible delivery

Content = Collections

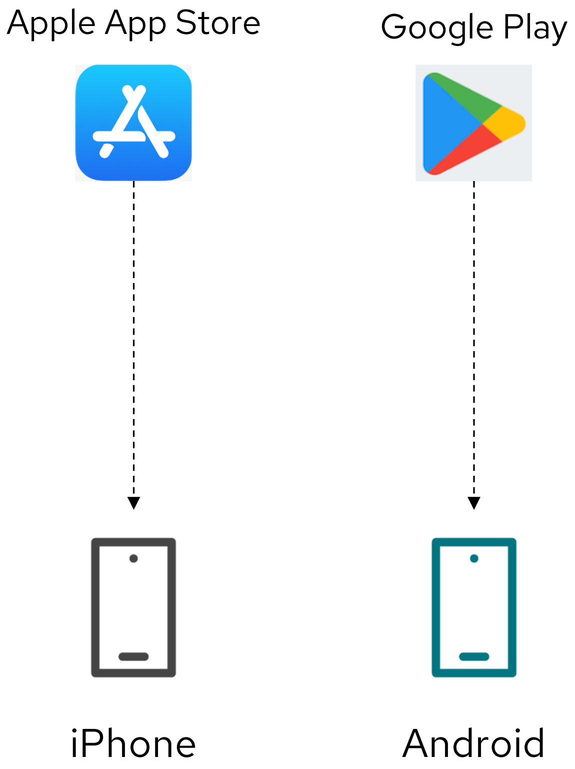


Automation Hub

GALAXY

Ansible Automation Platform

Content = Apps



Apple App Store

Google Play

iPhone

Android

Collection Directory Structure

- **galaxy.yml**: source data for the MANIFEST.json that will be part of the collection package
- **README.md**: “Front page” for documentation
- **docs/**: local documentation for the collection
- **meta/**: metadata files including runtime.yml (for redirection rules, compatibility, deprecation)
- **playbooks/**: playbook snippets
  - **tasks/**: holds 'task list files' for include_tasks/import_tasks usage
- **plugins/**: all Ansible plugins, each in its own subdir
  - **modules/**: module plugins (aka “modules”)
  - **lookups/**: lookup plugins
  - **filters/**: Jinja2 filter plugins
  - **connection/**: connection plugins required if not using default
- **roles/**: Ansible roles
- **tests/**: sanity, unit, integration tests

requires_ansible provides Ansible version compatibility

Reference: https://docs.ansible.com/ansible/latest/user_guide/collections_using.html

Ansible plugin types

- **modules** - Ansible modules (a.ka. task plugins) are discrete unit of code that can either run on managed host or control node and collects the return values. Example. cli_command, cli_config)

- **connection -** Connection plugins allow Ansible to connect to the target hosts so it can execute tasks on them. Ansible ships with many connection plugins, but only one can be used per host at a time. Example. network_cli, ssh

- **lookup** - Lookup plugins are an Ansible-specific extension to the Jinja2 templating language and can be used to access data from outside sources (files, databases, key/value stores, APIs, and other services) within your playbooks. Example. file

Reference: https://docs.ansible.com/ansible/latest/plugins/plugins.html

Ansible plugin types (contd.)

- **filter** - Filters plugin are used mainly transform data from within playbook like transform JSON data into YAML data, split a URL to extract the hostname so on. Example to_json, from_json, to_yaml, from_yaml

- **test** - Used for data validation in playbook and is Jinja way of evaluating template expressions and returning True or False

- **inventory** - Inventory plugins allow users to point at data sources to compile the inventory of hosts that Ansible uses to target tasks Example: amazon.aws.aws_ec2

- **callback** - Callback plugins enable adding new behaviors to Ansible when responding to events.
  - callback plugins control most of the output you see when running the playbook
  - can also be used to add additional output
  - integrate with other tools and marshall the events to a storage backend.

Reference: https://docs.ansible.com/ansible/latest/plugins/plugins.html

Ansible network collections

- **ansible.netcommon:**
  - Platform independent plugins
    - Connection (network_cli, netconf, httpapi)
    - Filter (network, ipaddr)
    - Modules (cli_config, cli_command, netconf_get, netconf_config etc.)

- **ansible.utils:**
  - Plugins to aid in the management, manipulation and visibility of data for the Ansible playbook developer.
    - Modules (cli_parse, validate, fact_diff etc.)
    - Filter (to_path, to_xml, index_of etc.)
    - Test (network test plugins)

- **arista.eos:**
  - Fundamental modules (eos_config, eos_command, eos_facts)
  - Resource modules (eos_inerfaces, eos_bgp etc.)

Collection can be downloaded from: https://galaxy.ansible.com/

- **cisco.ios:**
    - Fundamental modules (ios_config, ios_command, ios_facts)
    - Resource modules (ios_inerfaces, ios_bgp etc.)
- **cisco.iosxr:**
    - Fundamental modules (iosxr_config, iosxr_command, iosxr_facts)
    - Resource modules (iosxr_inerfaces, iosxr_bgp etc.)
- **cisco.nxos:**
    - Fundamental modules (nxos_config, nxos_command, nxos_facts)
    - Resource modules (nxos_inerfaces, nxos_bgp etc.)
- **junipernetworks.junos:**
    - Fundamental modules (junos_config, junos_command, junos_facts)
    - Resource modules (junos_inerfaces, junos_bgp etc.)
- **vyos.vyos:**
    - Fundamental modules (vyos_config, vyos_command, vyos_facts)
    - Resource modules (vyos_inerfaces, vyos_bgp etc.)

Collection can be downloaded from: https://galaxy.ansible.com/

Ansible Network Ecosystem

SWITCHES

ROUTERS

ENTERPRISE FIREWALLS

LOAD BALANCERS

CONTROLLERS

IP ADDRESS MGMT

A10

ARISTA

aruba NETWORKS

Check Point SOFTWARE TECHNOLOGIES LTD

CISCO

DELL EMC

Infoblox NEXT LEVEL NETWORKING

f5

Open Switch

JUNIPER NETWORKS

VyOS

100+

certified content collections

Infrastructure	Cloud	Network	Security
Red Hat	aws	ARISTA	Check Point SOFTWARE TECHNOLOGIES LTD
NetApp	Google	CISCO	CYBERARK
IBM	Microsoft	f5	FORTINET

Fundamental modules

NANOG™

Network modules to fetch state

- **ansible.netcommon.cli_command**:
  - Send the command to a network device and returns the result read from device.
  - The returned result can be in human readable plain text format or in structured format (e.g. JSON) based on device capability.
  - Provide executing command requiring inputs for multiple cli prompts

- **<network_os>_command**:
  - For example ***arista.eos.eos_command***, ***cisco.ios.ios_command*** and so on
  - Sends arbitrary commands to an node and returns the results read from the device.
  - Has an argument that will cause the module to wait for a specific condition before returning or timing out if the condition is not met.

- **<network_os>_facts:**
  - For example ***arista.eos.eos_facts***, ***cisco.ios.ios_facts*** and so on
  - Runs a predefined set show commands to gather operational and configurational data.

Network modules to fetch state (contd.)

- **ansible.netcommon.netconf_get**:
  - This module allows the user to fetch configuration and state data from NETCONF enabled network devices.
  - Work with *ansible.netcommon.netconf* connection

- **ansible.netcommon.netconf_rpc**:
  - This module allows the user to execute NETCONF RPC requests as defined by IETF RFC standards as well as proprietary requests.
  - Returns XML/JSON response data
  - Work with *ansible.netcommon.netconf* connection

Network modules to configuration management

- **ansible.netcommon.cli_config**:
  - This module provides platform agnostic way of pushing text based configuration to network devices over network_cli connection plugin.

- **<network_os>_config**:
  - This module provides an implementation for working with network configuration sections in a deterministic way.
  - Provides additional features like backup running config to a file on local system
  - Works with *ansible.netcommon.network_cli* connection

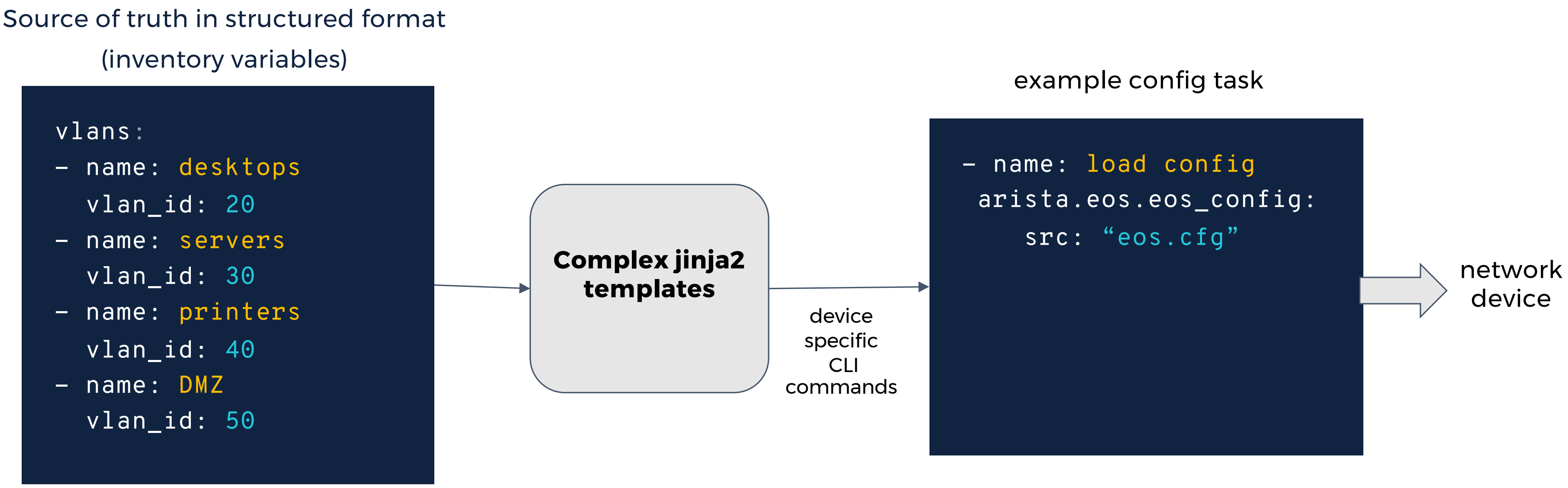
- **ansible.netcommon.netconf_config**:
  - This module allows the user to send a configuration in XML/JSON format to a netconf enabled device, and detects if there was a configuration change.
  - Work with *ansible.netcommon.netconf* connection

Demo

(See modules discussed so far in action)

Resource modules

Cons of <network_os>_config module



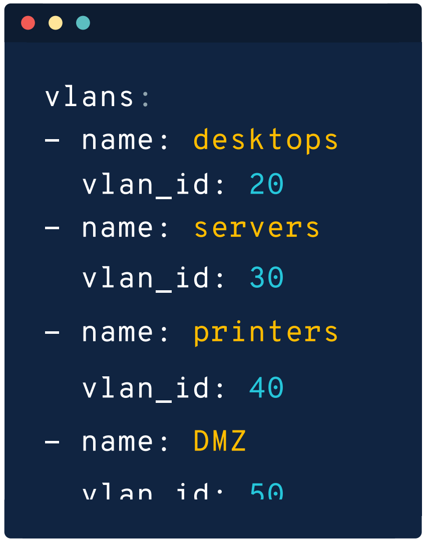
Source of truth in structured format
(inventory variables)

```
vlans:
- name: desktops
  vlan_id: 20
- name: servers
  vlan_id: 30
- name: printers
  vlan_id: 40
- name: DMZ
  vlan_id: 50
```

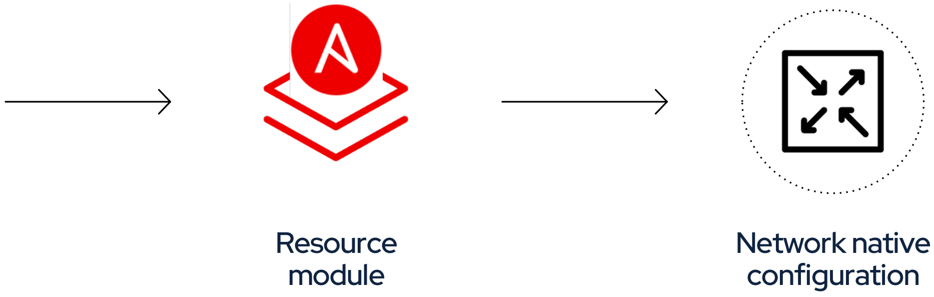
example config task

```
- name: load config
  arista.eos.eos_config:
     src: “eos.cfg”
```

Manage specific network resources



```
vlans:
– name: desktops
  vlan_id: 20
– name: servers
  vlan_id: 30
– name: printers
  vlan_id: 40
– name: DMZ
  vlan_id: 50
```



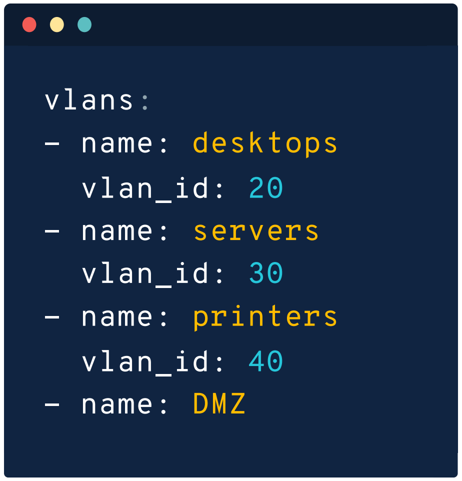
Resource module

Network native configuration

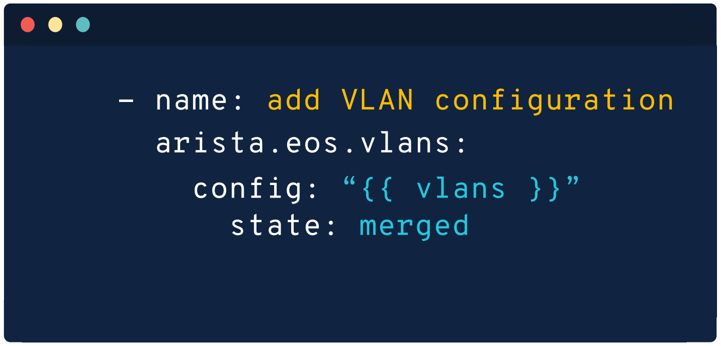
NANOG™

Managing device state

Practical examples of using network resource modules



```
vlans:
- name: desktops
  vlan_id: 20
- name: servers
  vlan_id: 30
- name: printers
  vlan_id: 40
- name: DMZ
```



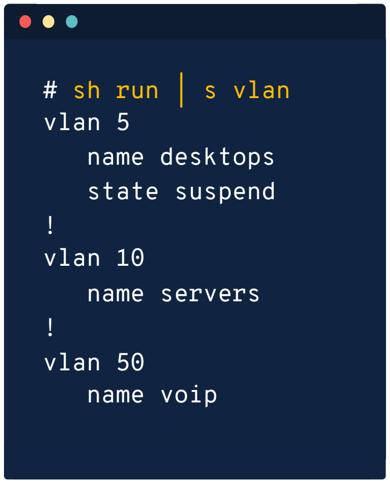
```
- name: add VLAN configuration
  arista.eos.vlans:
    config: “{{ vlans }}”
      state: merged
```

State:	`Merged`	- add/increment
	`Replaced`	- template/diff
	`Overridden`	- force/policy
	`Deleted`	- destroy/remediate

Understanding state parameters

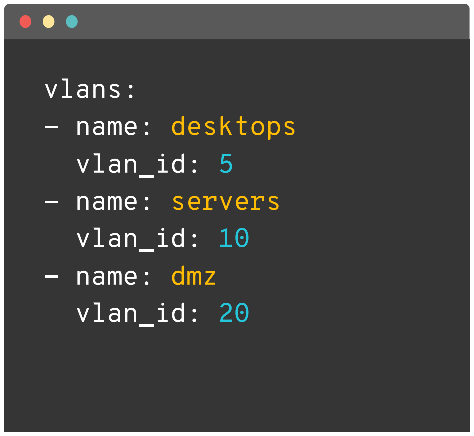
`state: merged`

Existing config



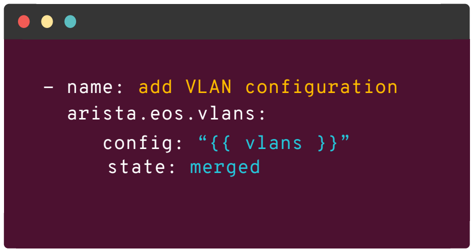
```
# sh run | s vlan
vlan 5
   name desktops
   state suspend
!
vlan 10
   name servers
!
vlan 50
   name voip
```

YAML Source of Truth



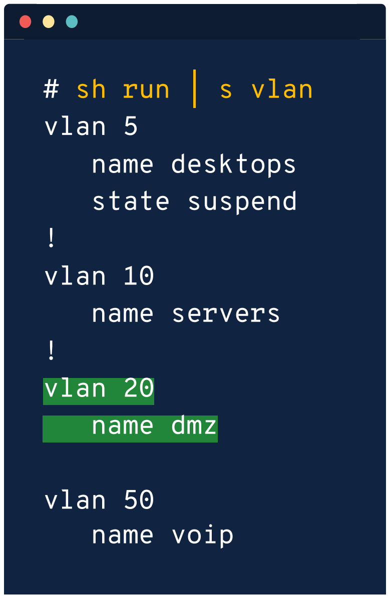
```
vlans:
- name: desktops
  vlan_id: 5
- name: servers
  vlan_id: 10
- name: dmz
  vlan_id: 20
```

Ansible task



```
- name: add VLAN configuration
  arista.eos.vlans:
     config: "{{ vlans }}"
     state: merged
```

New Config



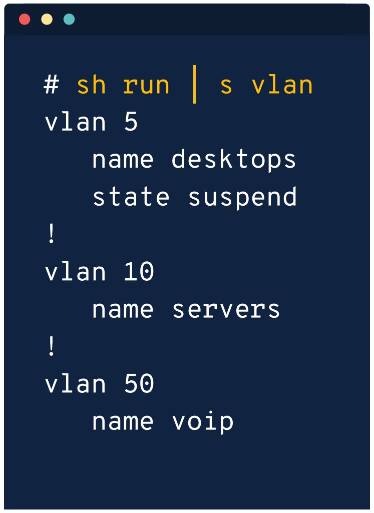
```
# sh run | s vlan
vlan 5
   name desktops
   state suspend
!
vlan 10
   name servers
!
vlan 20
   name dmz

vlan 50
   name voip
```

NANOG™

Understanding state parameters

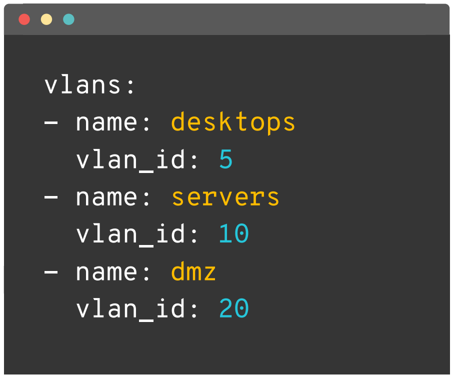
`state: replaced`

Existing config



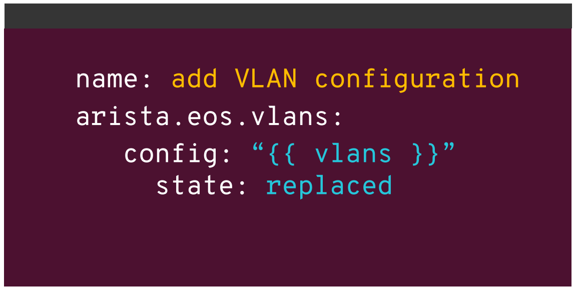
```
# sh run | s vlan
vlan 5
   name desktops
   state suspend
!
vlan 10
   name servers
!
vlan 50
   name voip
```

YAML Source of Truth



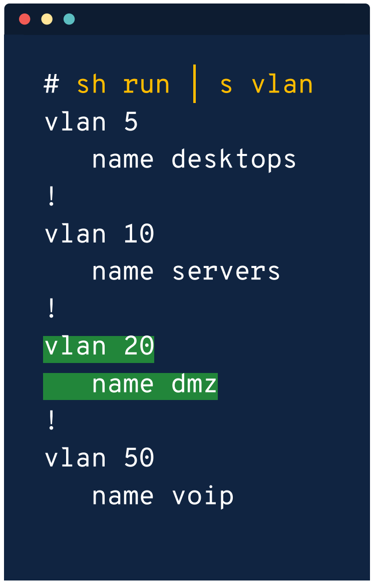
```
vlans:
- name: desktops
  vlan_id: 5
- name: servers
  vlan_id: 10
- name: dmz
  vlan_id: 20
```

Ansible task



```
name: add VLAN configuration
arista.eos.vlans:
   config: “{{ vlans }}”
     state: replaced
```

New Config

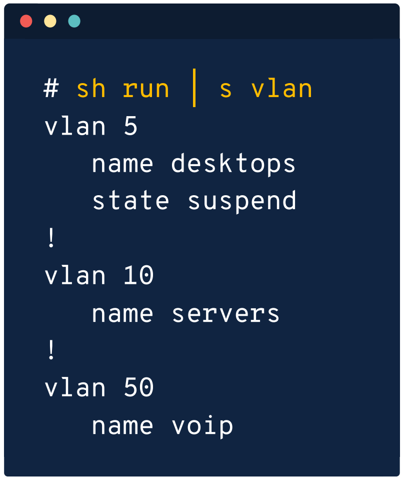


```
# sh run | s vlan
vlan 5
   name desktops
!
vlan 10
   name servers
!
vlan 20
   name dmz
!
vlan 50
   name voip
```

Understanding state parameters

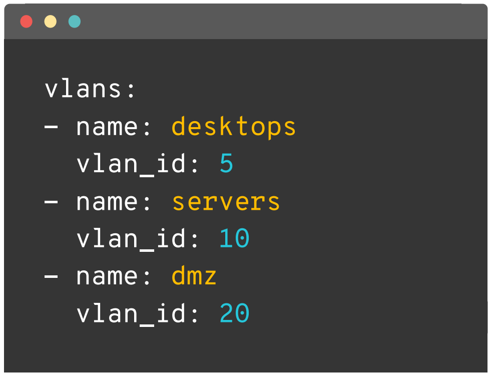
`state: overridden`

Existing config



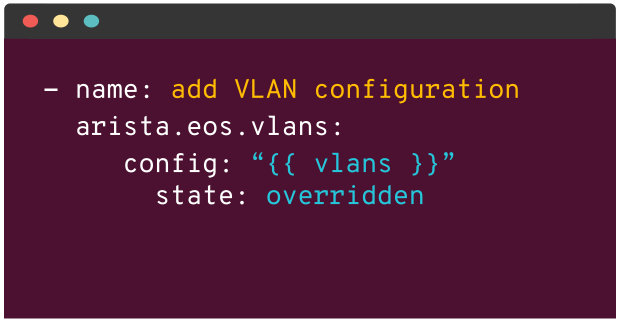
```
# sh run | s vlan
vlan 5
   name desktops
   state suspend
!
vlan 10
   name servers
!
vlan 50
   name voip
```

YAML Source of Truth



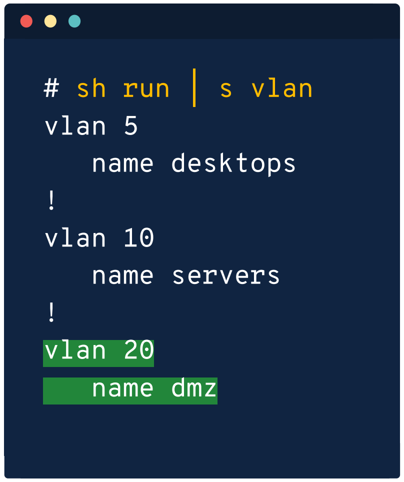
```
vlans:
- name: desktops
  vlan_id: 5
- name: servers
  vlan_id: 10
- name: dmz
  vlan_id: 20
```

Ansible task



```
- name: add VLAN configuration
  arista.eos.vlans:
     config: “{{ vlans }}”
       state: overridden
```

New Config

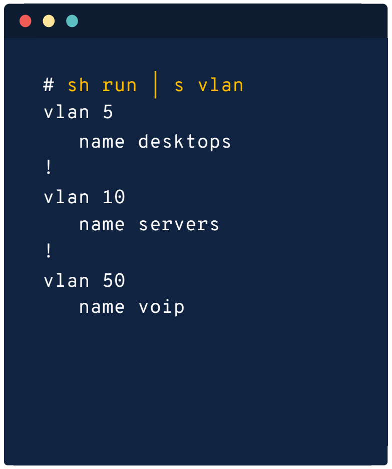


```
# sh run | s vlan
vlan 5
   name desktops
!
vlan 10
   name servers
!
vlan 20
   name dmz
```

Understanding state parameters

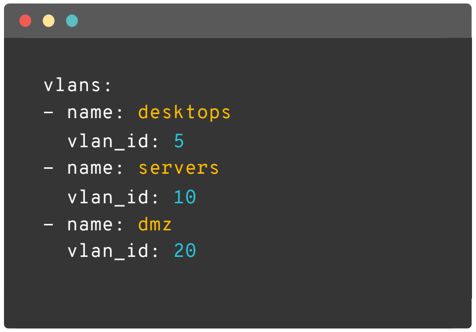
`state: deleted`

Existing config



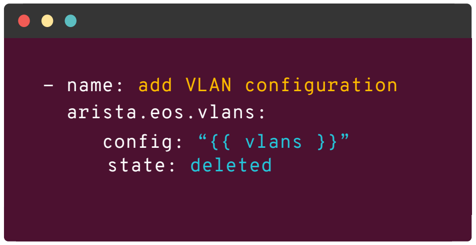
```
# sh run | s vlan
vlan 5
   name desktops
!
vlan 10
   name servers
!
vlan 50
   name voip
```

YAML Source of Truth



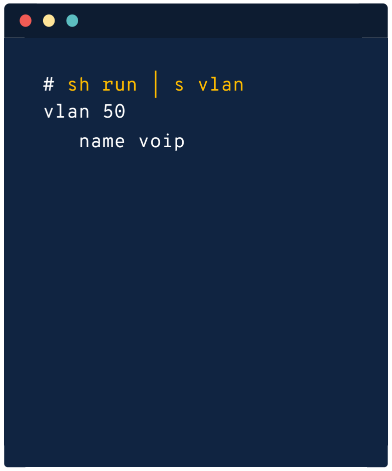
```
vlans:
- name: desktops
  vlan_id: 5
- name: servers
  vlan_id: 10
- name: dmz
  vlan_id: 20
```

Ansible task



```
- name: add VLAN configuration
  arista.eos.vlans:
     config: “{{ vlans }}”
     state: deleted
```

New Config



```
# sh run | s vlan
vlan 50
   name voip
```

Network Resource Modules - Return values

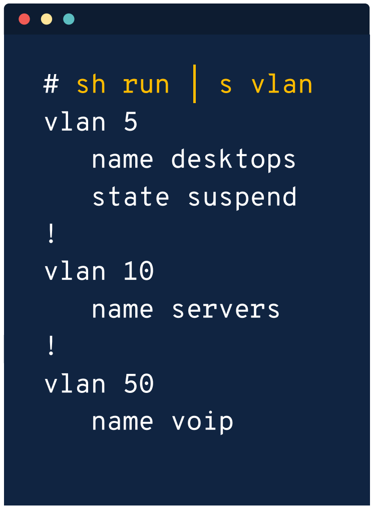
Practical examples of using network resource modules

- **before**
  The configuration prior to module execution is always returned.
- **commands**
  delta command set for the device
- **after**
  the configuration post module execution

Understanding return values

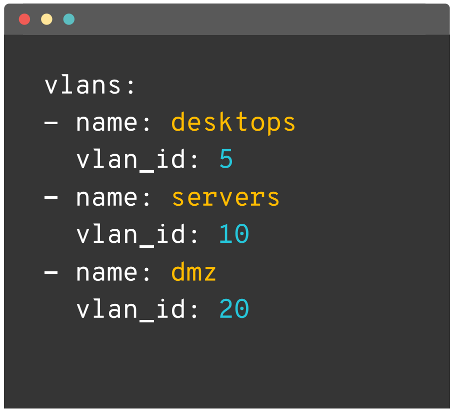
`state: merged`

Existing config



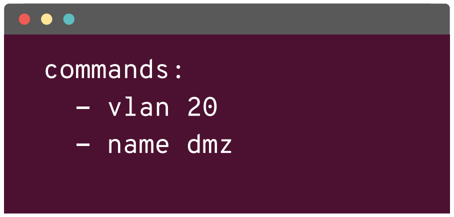
```
# sh run | s vlan
vlan 5
   name desktops
   state suspend
!
vlan 10
   name servers
!
vlan 50
   name voip
```

YAML Source of Truth



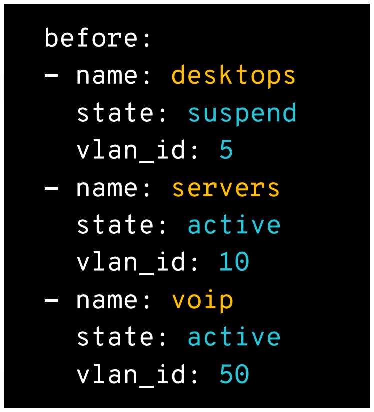
```
vlans:
- name: desktops
  vlan_id: 5
- name: servers
  vlan_id: 10
- name: dmz
  vlan_id: 20
```

Commands



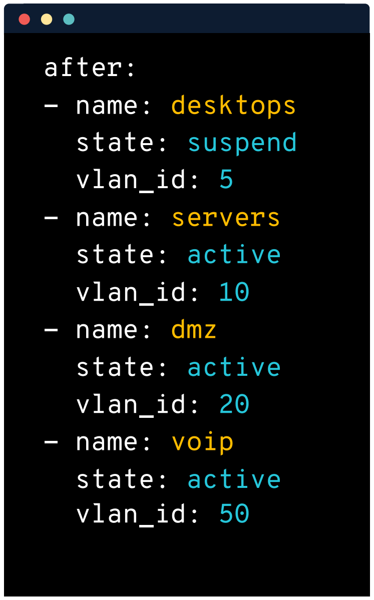
```
commands:
  - vlan 20
  - name dmz
```

Before



```
before:
- name: desktops
  state: suspend
  vlan_id: 5
- name: servers
  state: active
  vlan_id: 10
- name: voip
  state: active
  vlan_id: 50
```

After



```
after:
- name: desktops
  state: suspend
  vlan_id: 5
- name: servers
  state: active
  vlan_id: 10
- name: dmz
  state: active
  vlan_id: 20
- name: voip
  state: active
  vlan_id: 50
```

Demo

(See resource modules discussed so far in action)

Operational state management modules

NANOG™

Use cases for operational state data assessment

Common state assessment workflow

- **Retrieve (source of truth):**
  - Collect the current operational state from the remote host
  - Convert it into normalised structure data.
  - Store is as inventory variables

- **Validate:**
  - Define the desired state criteria in a standard based format
  - Retrieve operational state at runtime
  - Validate the current state data against the pre-defined criteria to identify if there is any deviation.

- **Remediate:**
  - Required configuration changes to remove the drift
  - Reporting

Use cases for operational state data assessment

- Conditional task and roles within Ansible playbooks (pre-config)
  - Only make configuration changes if all the BGP neighbours are healthy
- Fleet health assessment and inventory
  - Ensure all configured NTP servers are in sync
- Post change validation
  - LLDP, OSPF neighbours and reachability has not changed
- Custom reports using templates
  - Interface operating state vs. configured state

Retrieving operational state in structured format

- **ansible.utils.cli_parse**:
  - Module available now in ansible.utils collection
  - Parse CLI output or text using a variety of parsers
  - Works with all platforms (network/linux/windows)
  - Work with many parsing engines and is extensible
  - Single task to run a command, parse & set facts
  - Returns structured data from show command output

https://galaxy.ansible.com/ansible/utils

NANOG™

Retrieving operational state in structured format

```
tasks:
- name: Run a command and parse results
  ansible.utils.cli_parse:
    command: show interfaces
    parser:
      name: ansible.utils.xxxx
      set_fact: interfaces
```

- Runs the command on the device
- Parse using the ‘xxxx’ engine
- Uses default template folder
- Parsed data set as fact
- Command output returned as stdout

Available ansible.utils.cli_parse parsing engines

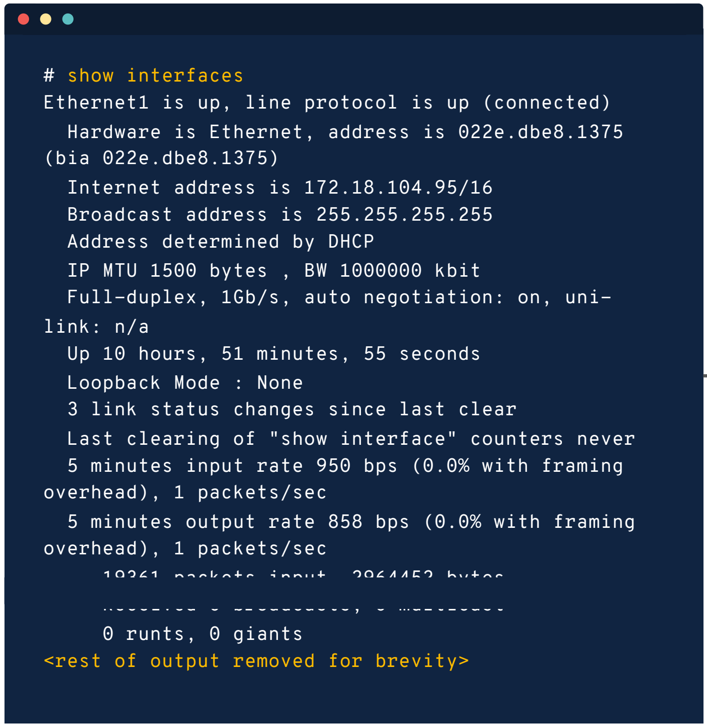
- **ansible.utils.textfsm**: Python module for parsing semi-formatted text
- **ansible.utils.ttp**: Template based parsing, low regex use, jinja like DSL
- **ansible.netcommon.native**: Internal jinja, regex, yaml. No additional 3rd party libraries required
- **ansible.netcommon.ntc_templates**: Predefined textfsm templates packaged as python library
- **ansible.netcommon.pyats**: Cisco Test Automation & Validation Solution (11 OSs/2500 parsers)
- **ansible.utils.xml**: convert XML to json using xmltodict

Thank you library developers & contributors

Parsing Example (1/3)

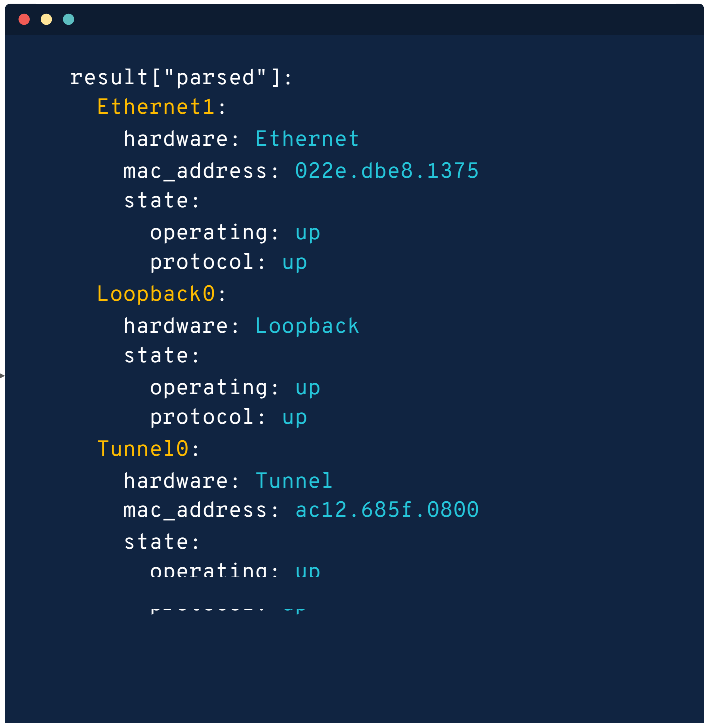
parsing using native Ansible parsing library

Network Device output



```
# show interfaces
Ethernet1 is up, line protocol is up (connected)
  Hardware is Ethernet, address is 022e.dbe8.1375
(bia 022e.dbe8.1375)
  Internet address is 172.18.104.95/16
  Broadcast address is 255.255.255.255
  Address determined by DHCP
  IP MTU 1500 bytes , BW 1000000 kbit
  Full-duplex, 1Gb/s, auto negotiation: on, uni-
link: n/a
  Up 10 hours, 51 minutes, 55 seconds
  Loopback Mode : None
  3 link status changes since last clear
  Last clearing of "show interface" counters never
  5 minutes input rate 950 bps (0.0% with framing
overhead), 1 packets/sec
  5 minutes output rate 858 bps (0.0% with framing
overhead), 1 packets/sec
     19361 packets input, 2964452 bytes
     Received 0 broadcasts, 0 multicast
     0 runts, 0 giants
<rest of output removed for brevity>
```

Parsed Data

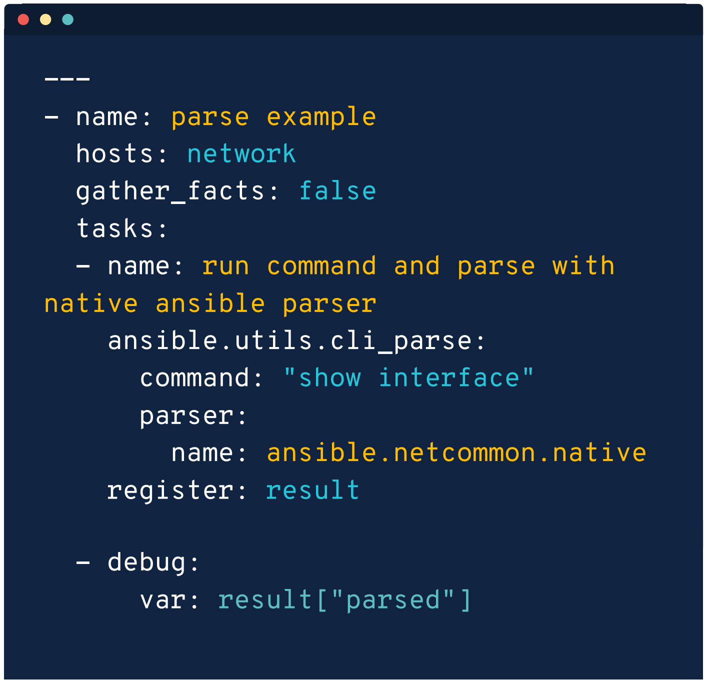


```
result["parsed"]:
  Ethernet1:
    hardware: Ethernet
    mac_address: 022e.dbe8.1375
    state:
      operating: up
      protocol: up
  Loopback0:
    hardware: Loopback
    state:
      operating: up
      protocol: up
  Tunnel0:
    hardware: Tunnel
    mac_address: ac12.685f.0800
    state:
      operating: up
      protocol: up
```

Parsing Example (2/3)

How does it work?

Ansible Playbook



```
---
- name: parse example
  hosts: network
  gather_facts: false
  tasks:
  - name: run command and parse with
native ansible parser
    ansible.utils.cli_parse:
      command: "show interface"
      parser:
        name: ansible.netcommon.native
    register: result

  - debug:
      var: result["parsed"]
```

Example with Arista EOS

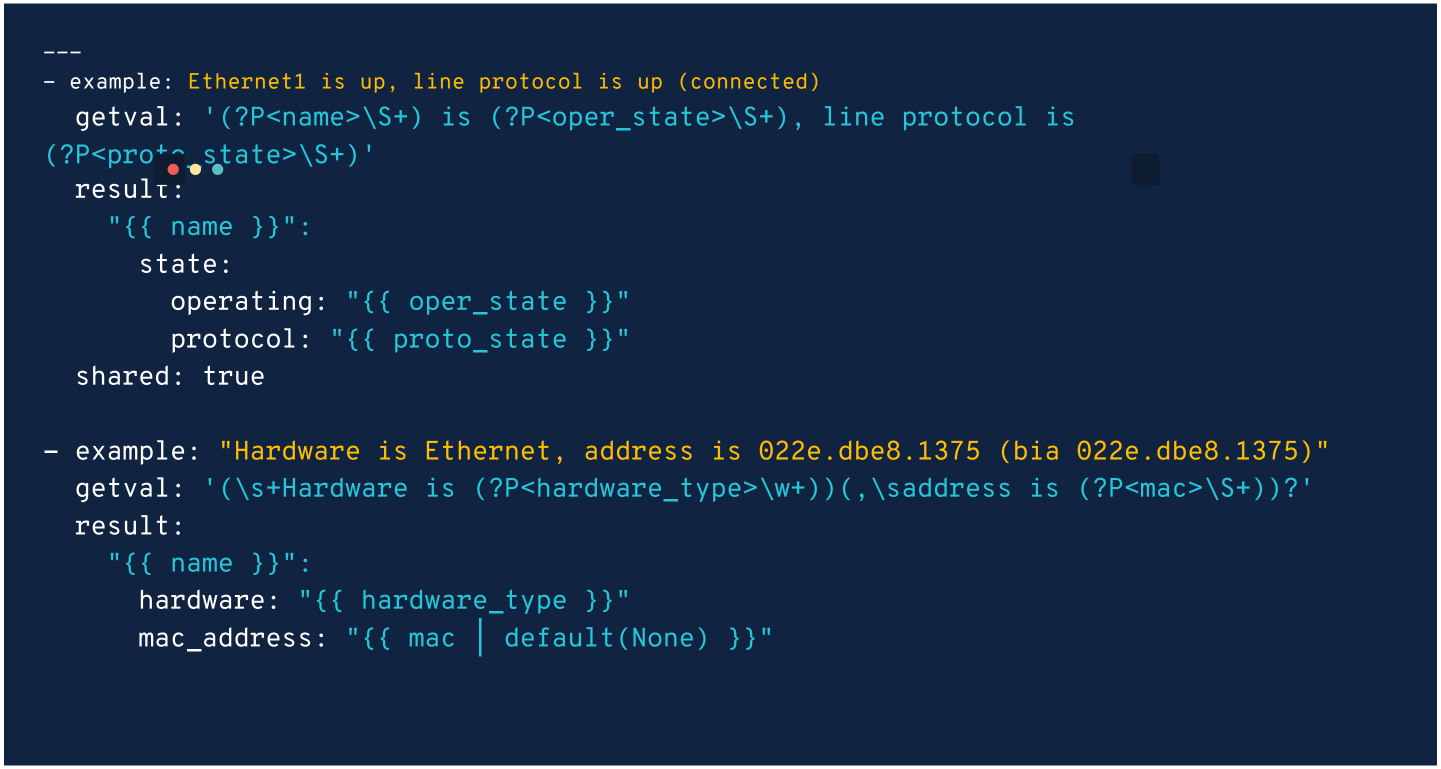
ansible_newtork_os: arista.eos.eos
command: “show interface”

Looks for:
templates/eos_show_interface.yaml

Parsing Example (3/3)

Easy to share templates with others

Parsing Template Example



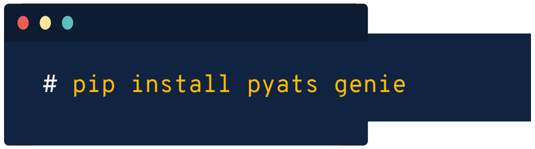
```
---
- example: Ethernet1 is up, line protocol is up (connected)
  getval: '(?P<name>\S+) is (?P<oper_state>\S+), line protocol is
(?P<proto_state>\S+)'
  result:
    "{{ name }}":
      state:
        operating: "{{ oper_state }}"
        protocol: "{{ proto_state }}"
  shared: true

- example: "Hardware is Ethernet, address is 022e.dbe8.1375 (bia 022e.dbe8.1375)"
  getval: '(\s+Hardware is (?P<hardware_type>\w+))(,\saddress is (?P<mac>\S+))?'
  result:
    "{{ name }}":
      hardware: "{{ hardware_type }}"
      mac_address: "{{ mac | default(None) }}"
```

templates/eos_show_interface.yaml

Integrating with pyATS & Genie

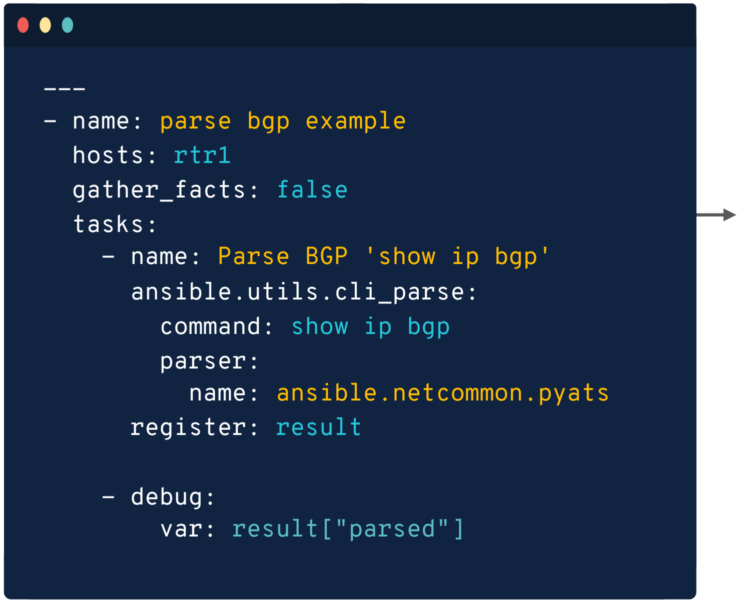
Quick setup for Cisco Network Devices

Install Python Packages



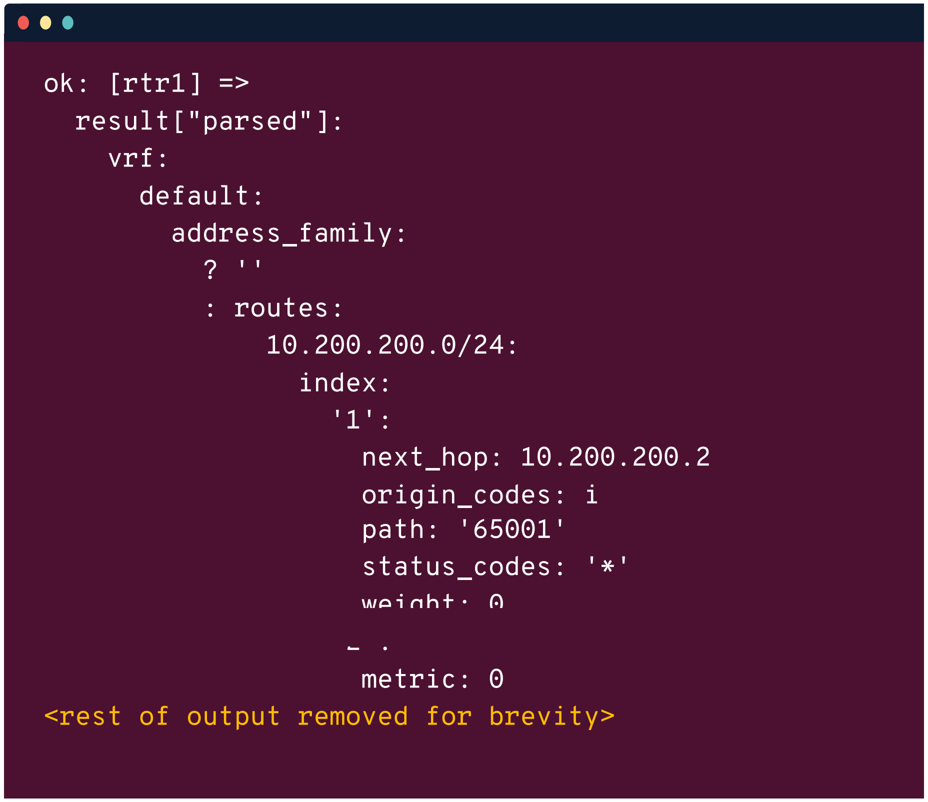
```
# pip install pyats genie
```

Run Ansible Playbook



```
---
- name: parse bgp example
  hosts: rtr1
  gather_facts: false
  tasks:
    - name: Parse BGP 'show ip bgp'
      ansible.utils.cli_parse:
        command: show ip bgp
        parser:
          name: ansible.netcommon.pyats
      register: result

    - debug:
        var: result["parsed"]
```

Output to terminal window



```
ok: [rtr1] =>
  result["parsed"]:
    vrf:
      default:
        address_family:
          ? ''
          : routes:
            10.200.200.0/24:
              index:
                '1':
                  next_hop: 10.200.200.2
                  origin_codes: i
                  path: '65001'
                  status_codes: '*'
                  weight: 0

                 - .
                  metric: 0
<rest of output removed for brevity>
```

Validating structured data using Ansible

- **ansible.utils.validate**:
  - New module available now in ansible.utils collection
  - Works with all platforms
  - Has extensible validation engine support, currently works with jsonschema validation engine
  - Single task to read the structured data and validate it with data model schemas
  - Returns either list of errors or success (in case data is valid as per schema)

Validating structured data using Ansible

```
tasks:
- name: "Validate structured data"
  ansible.utils.validate:
    data: "{{ input_data }}"
    criteria:
     - "{{ lookup('file',  './criteria.json') | from_json }}"
    engine: ansible.utils.xxxx
```

- Reads the input JSON data and the criteria for data (schema mode)
- Validate using the ‘xxxx’ engine
- Returns list of error if data does not conform to the schema criteria

Available ansible.utils.validate validation engines

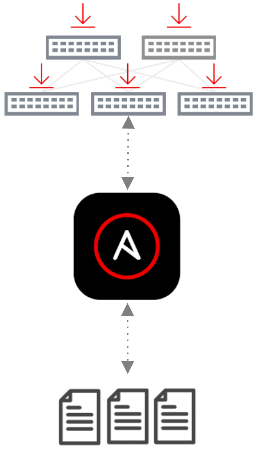
- **ansible.utils.jsonschema**: Python module to validate json data against a schema

More validation engines in pipeline

Take away

Start Small

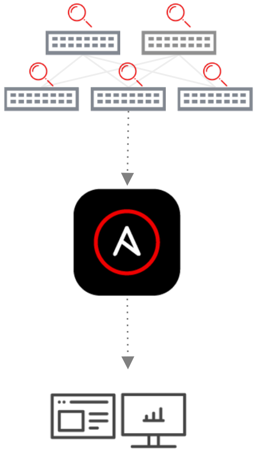
Quick automation victories for network engineers



Config Backup and Restore

Ubiquitous first touch use case

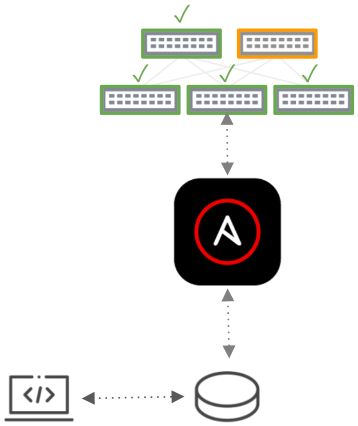
- Gain confidence in automation quickly
- First steps towards network as code
- Quickly recover network steady state



Dynamic Documentation

Use Ansible facts to gain information

- Read-only, no production config change
- Dynamic Documentation and reporting
- Understand your network



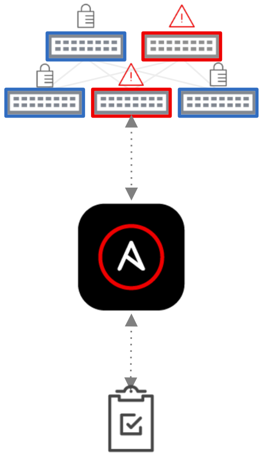
Scoped Config Management

Focus on high yield victories

- Automate VLANs, ACLs and SNMP config
- Introduce source of truth concepts
- Enforce Configuration policy

Think Big

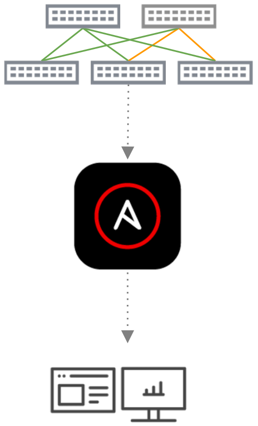
Institutionalizing automation into your organization



Network Compliance

Respond quickly and consistently

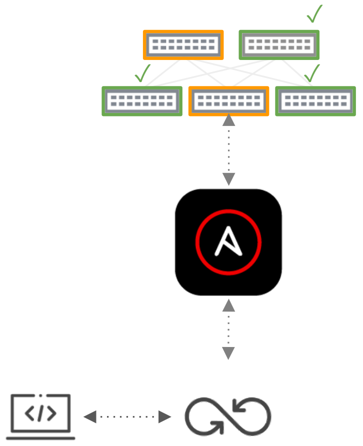
- Security and config compliance for network
- Remove human error from security responses
- Enforce Configuration policies and hardening



Operational State Validation

Going beyond config management

- Parsing operational state to structured values
- Schema validation and verification
- Enhance operational workflows



Automated NetOps

Infrastructure as code

- Data centric automation
- Deploy configuration pipelines
- GitOps for Network Automation

NANOG™

References

- https://docs.ansible.com/ansible/latest/network/user_guide/platform_index.html
- https://docs.ansible.com/ansible/2.9/dev_guide/overview_architecture.html
- https://docs.ansible.com/ansible/latest/network/dev_guide/developing_plugins_network.html
- https://docs.ansible.com/ansible/latest/network/user_guide/network_resource_modules.html
- https://docs.ansible.com/ansible/latest/network/user_guide/cli_parsing.html
- https://docs.ansible.com/ansible/devel/network/user_guide/validate.html

Thank you

NANOG™