



# Chapter 29: Introduction to Automation Tools

Instructor Materials

CCNP Enterprise: Core Networking



# Chapter 29 Content

## This chapter covers the following content:

- **Embedded Event Manager (EEM)** - This section illustrates common use cases and operations of the on-box EEM automation tool as well as the Tcl scripting engine.
- **Agent-Based Automation Tools** - This section examines the benefits and operations of the various agent-based automation tools.
- **Agentless Automation Tools** - This section examines the benefits and operations of the various agentless automation tools.

This chapter is intended to provide a high-level overview of some of the most common configuration management and automation tools that are available.

# Embedded Event Manager

- Embedded Event Manager (EEM) is a very flexible and powerful Cisco IOS tool. EEM allows engineers to build software applets that can automate many tasks.
- EEM enables you to build custom scripts using Tcl. Scripts can automatically execute based on the output of an action or an event on a device.
- EEM is all contained within the local device. There is no need to rely on an external scripting engine or monitoring device in most cases.
- This section will cover EEM applets, EEM and Tcl scripts, and EEM summary.

# Embedded Event Manager

## EEM Event Detectors

Figure 29-1 illustrates some EEM event detectors and how they interact with the IOS subsystem.

- EEM applets are composed of multiple building blocks. This chapter focuses on two of the primary building blocks that make up EEM applets: events and actions.
- EEM applets use a similar logic to the if-then statements used in some of the common programming languages (for instance, if an event happens, then an action is taken).

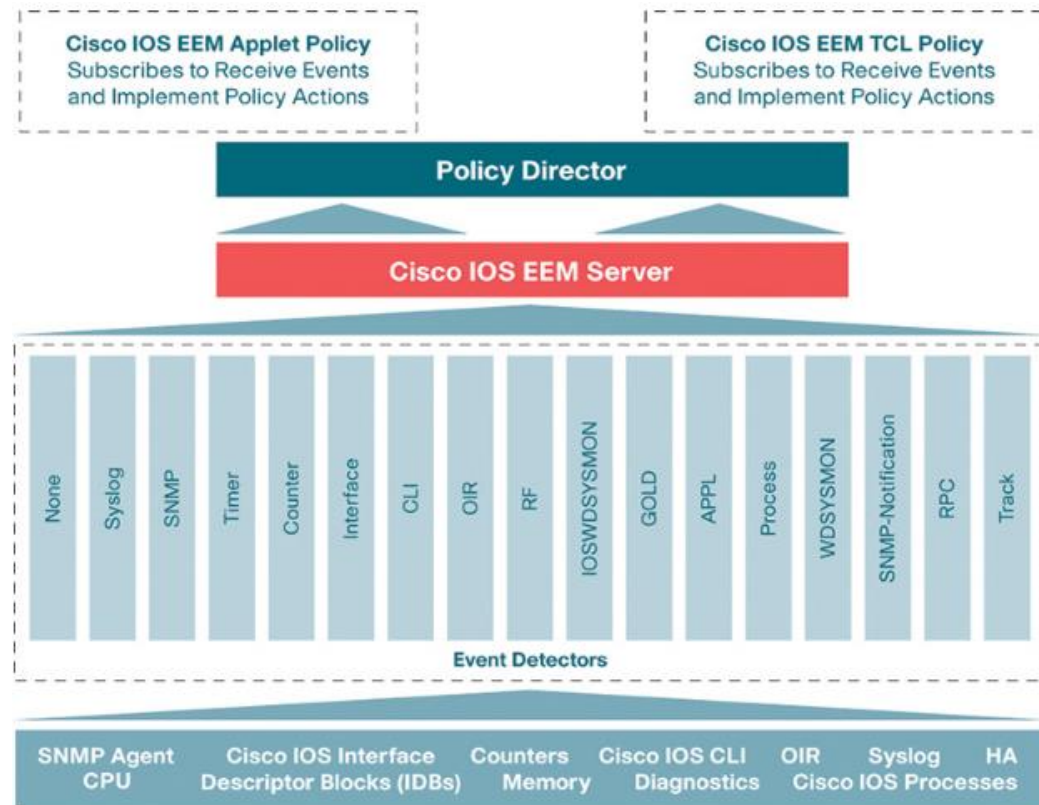


Figure 29-1  
EEM Event Detectors

# Embedded Event Manager EEM Applets

Example 29-1 shows an applet that is looking for a specific syslog message, stating that the Loopback0 interface went down. If this specific syslog pattern is matched (an event) at least once, then the following actions will be taken:

1. The Loopback0 interface will be shut down and brought back up.
2. The router will generate a syslog message that says, "I've fallen, and I can't get up!"
3. An email message that includes the output of the **show interface loopback0** command will be sent to the network administrator.

**Example 29-1** *Syslog Applet Example*

```
event manager applet LOOP0
  event syslog pattern "Interface Loopback0.* down" period 1
  action 1.0 cli command "enable"
  action 2.0 cli command "config terminal"
  action 3.0 cli command "interface loopback0"
  action 4.0 cli command "shutdown"
  action 5.0 cli command "no shutdown"
  action 5.5 cli command "show interface loopback0"
  action 6.0 syslog msg "I've fallen, and I can't get up!"
  action 7.0 mail server 10.0.0.25 to neteng@yourcompany.com
    from no-reply@yourcompany.com subject "Loopback0 Issues!"
    body "The Loopback0 interface was bounced. Please monitor
    accordingly. "$_cli_result"
```

Remember to include the **enable** and **configure terminal** commands at the beginning of actions within an applet.

# Embedded Event Manager Debugging Output

Based on the output from the **debug event manager action cli** command, you can see the actions taking place when the applet is running. Example 29-2 shows the applet being engaged when a user issues the **shutdown** command on the Loopback0 interface.

**Example 29-2** *Debugging Output of an Event Manager Action*

```
Switch#
Switch# configure terminal
Enter configuration commands, one per line.  End with CNTL/Z.
Switch(config)# interface loopback0
Switch(config-if)# shutdown
Switch(config-if)#

17:21:59.214: %LINK-5-CHANGED: Interface Loopback0, changed state to administratively
down
17:21:59.217: %HA_EM-6-LOG: LOOP0 : DEBUG(cli_lib) : : CTL : cli_open called.
17:21:59.221: %HA_EM-6-LOG: LOOP0 : DEBUG(cli_lib) : : OUT : Switch>
17:21:59.221: %HA_EM-6-LOG: LOOP0 : DEBUG(cli_lib) : : IN  : Switch>enable
17:21:59.231: %HA_EM-6-LOG: LOOP0 : DEBUG(cli_lib) : : OUT : Switch#
17:21:59.231: %HA_EM-6-LOG: LOOP0 : DEBUG(cli_lib) : : IN  : Switch#show
interface loopback0
17:21:59.252: %HA_EM-6-LOG: LOOP0 : DEBUG(cli_lib) : : OUT : Loopback0 is
administratively down, line protocol is down
```

(Entire output not shown.)

# Embedded Event Manager

## CLI Patterns

When certain commands are entered using the CLI, they trigger an EEM event within an applet. The configured actions then take place as a result of the CLI pattern being matched.

Example 29-3 uses another common EEM applet to match the CLI pattern “write mem.” When the applet is triggered, the following actions are invoked:

1. The router generates a syslog message that says “Configuration File Changed! TFTP backup successful.”
2. The startup-config file is copied to a TFTP server.

### Example 29-3 WR MEM Applet

```
event manager environment filename Router.cfg
event manager environment tftpserver tftp://10.1.200.29/
event manager applet BACKUP-CONFIG
  event cli pattern "write mem.*" sync yes
  action 1.0 cli command "enable"
  action 2.0 cli command "configure terminal"
  action 3.0 cli command "file prompt quiet"
  action 4.0 cli command "end"
  action 5.0 cli command "copy start $tftpserver$filename"
  action 6.0 cli command "configure terminal"
  action 7.0 cli command "no file prompt quiet"
  action 8.0 syslog priority informational msg "Configuration File Changed!
    TFTP backup successful."
```

## Embedded Event Manager

# EEM Environment Values

There are multiple ways to call out specific EEM environment values. Although it is possible to create custom names and values that are arbitrary and can be set to anything, it is good practice to use common and descriptive variables. Table 29-2 lists some of the email variables most commonly used in EEM.

**Table 29-2** Common EEM Email Variables

EEM Variable	Description	Example
_email_server	SMTP server IP address or DNS name	10.0.0.25 or MAILSVR01
_email_to	Email address to send email to	neteng@yourcompany.com
_email_from	Email address of sending party	No-reply@yourcompany.com
_email_cc	Email address of additional email receivers	helpdesk@yourcompany.com



# Embedded Event Manager Tcl Scripts

Example 29-4 shows how to manually execute an EEM applet that executes a Tcl script. It shows an EEM script configured with the **event none** command, which means there is no automatic event that the applet is monitoring, and this applet runs only when it is triggered manually. To manually run an EEM applet, the **event manager run applet-name** command must be used.

## Example 29-4 Manually Execute EEM Applet

```
event manager applet Ping
  event none
  action 1.0 cli command "enable"
  action 1.1 cli command "tclsh flash:/ping.tcl"
Router# event manager run Ping
Router#
19:32:16.564: %HA_EM-6-LOG: Ping : DEBUG(cli_lib) : : CTL : cli_open called.
19:32:16.564: %HA_EM-6-LOG: Ping : DEBUG(cli_lib) : : OUT : Router>
19:32:16.568: %HA_EM-6-LOG: Ping : DEBUG(cli_lib) : : IN  : Router>enable
19:32:16.578: %HA_EM-6-LOG: Ping : DEBUG(cli_lib) : : OUT : Router#
19:32:16.578: %HA_EM-6-LOG: Ping : DEBUG(cli_lib) : : IN  : Router#tclsh
flash:/ping.tcl
19:32:16.711: %HA_EM-6-LOG: Ping : DEBUG(cli_lib) : : OUT : Type escape sequence
to abort.
19:32:16.711: %HA_EM-6-LOG: Ping : DEBUG(cli_lib) : : OUT : Sending 5, 100-byte
ICMP Echos to 192.168.0.2, timeout is 2 seconds:
19:32:16.711: %HA_EM-6-LOG: Ping : DEBUG(cli_lib) : : OUT : !!!!!
19:32:16.711: %HA_EM-6-LOG: Ping : DEBUG(cli_lib) : : OUT : Success rate is
100 percent (5/5), round-trip min/avg/max = 1/1/4 ms
```

(Entire output not shown.)

# Embedded Event Manager

## Script Contents

Example 29-5 displays a snippet for the exact content of the ping.tcl script used in the manually triggered EEM applet in Example 29-4.

To see the contents of a Tcl script that resides in flash memory, issue the **more** command followed by the file location and filename.

The **more** command can be used to view all other text-based files stored in the local flash memory as well.

**Example 29-5** *ping.tcl Script Contents*

```
Router# more flash:ping.tcl
foreach address {
192.168.0.2
192.168.0.3
192.168.0.4
192.168.0.5
192.168.0.6
} { ping $address}
```

## Embedded Event Manager

# EEM Summary

The value in using automation and configuration management tools is the ability to move more quickly than is possible with manual configuration.

Automation helps ensure that the level of risk due to human error is reduced by using proven automation methods.

The following are some of the most common and repetitive configurations for which network operators leverage automation tools to increase speed and consistency:

- Device name/IP address
- Quality of service
- Access list entries
- Usernames/passwords
- SNMP settings
- Compliance

# Agent-Based Automation Tools

- This section covers a number of agent-based tools as well as some of the key concepts to help network operators decide which tool best suits their environment and business use cases.

## Agent-Based Automation Tools

# Puppet

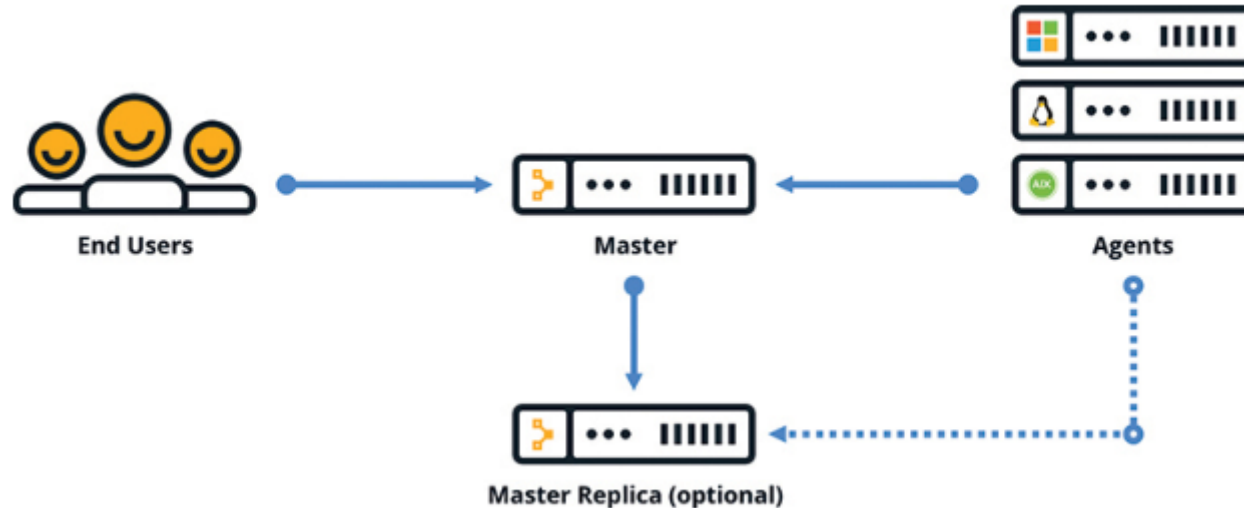
- Puppet is a robust configuration management and automation tool. Puppet works with many different vendors and is a commonly used tool used for automation.
- Puppet can be used during the entire lifecycle of a device, including initial deployment, configuration management, and repurposing and removing devices in a network.
- Puppet uses a puppet master (server) to communicate with devices that have the puppet agent (client) installed locally on the device.
- Changes and automation tasks are executed within the puppet console and then shared between the puppet master and puppet agents.
- These changes or automation tasks are stored in the puppet database (PuppetDB) so that the tasks can be saved to be pushed out to the puppet agents at a later time.

# Agent-Based Automation Tools

## Puppet (Cont.)

Figure 29-2 illustrates the communications path between the puppet master and the puppet agents, as well as the high-level architecture. The solid lines show the primary communications path, and the dotted lines indicate optional high availability.

**Figure 29-2** *High-Level Puppet Architecture and Basic Puppet Communications Path*



## Agent-Based Automation Tools

# Puppet (Cont.)

- Puppet agents communicate to the puppet master by using different TCP connections. TCP ports uniquely represent a communications path from an agent running on a device or node.
- Puppet can periodically verify the configuration on devices. This can be set to any frequency that the network administrator deems necessary. If a configuration is changed, it can be alerted on, and automatically put back to the previous configuration.
- There are three different installation types with Puppet. Table 29-3 describes the scale differences between the different installation options.

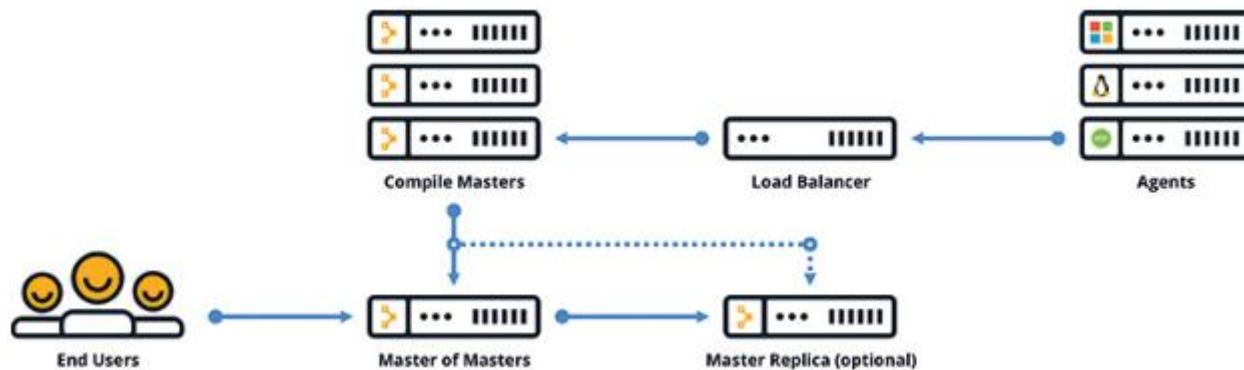
**Table 29-3 Puppet Installation Modes**

Installation Type	Scale
Monolithic	Up to 4000 nodes
Monolithic with compile masters	4000 to 20,000 nodes
Monolithic with compile masters and standalone PE-PostgreSQL	More than 20,000 nodes

## Agent-Based Automation Tools

# Puppet (Cont.)

- The recommended deployment is a monolithic installation, which supports up to 4000 nodes. In a very large scale environment, best practices are high availability and centralized management. Administrators may need a master of masters (MoM) to manage the distributed puppet masters and their associated databases.
- Large deployments also need compile masters, which are load-balanced Puppet servers. Figure 29-3 shows a typical large-scale enterprise deployment model of Puppet and its associated components.



**Figure 29-3** *Large-Scale Puppet Enterprise Deployment*



## Agent-Based Automation Tools

# Puppet Modules

Puppet modules contain these components:

- Manifests
- Templates
- Files

Manifests are the code that configures the clients or nodes running the puppet agent.

Manifests are pushed to the devices using SSL and require certificates to ensure the security of communications between the puppet master and the puppet agents.

Manifests can be saved as individual files and have a file extension .pp.

One module called `cisco_ios`, contains many manifests and leverages SSH to connect to devices.

## Agent-Based Automation Tools

# Puppet Manifest

- Example 29-6 shows an example of a manifest file, named NTP\_Server.pp, that configures a Network Time Protocol (NTP) server on a Cisco Catalyst device.
- This example shows that the NTP server IP address is configured as 1.2.3.4, and it uses VLAN 42 as the source interface. The line ensure => 'present' means that the NTP server configuration should be present in the running configuration of the Catalyst IOS device on which the manifest is running.

### Example 29-6 *Puppet NTP\_Server.pp Manifest*

```
ntp_server { '1.2.3.4':  
    ensure => 'present',  
    key => 94,  
    prefer => true,  
    minpoll => 4,  
    maxpoll => 14,  
    source_interface => 'Vlan 42',  
}
```

## Agent-Based Automation Tools

# Puppet Manifest (Cont.)

- Puppet leverages a domain-specific language (DSL) as its “programming language.” It is based on the Ruby language.
- Example 29-7 shows a manifest file called MOTD.pp that is used to configure a message-of-the-day (MOTD) banner on Catalyst IOS devices.

### **Example 29-7** *Puppet MOTD.pp Manifest*

```
banner { 'default':  
  motd => 'Violators will be prosecuted',  
}
```

## Agent-Based Automation Tools

# Chef

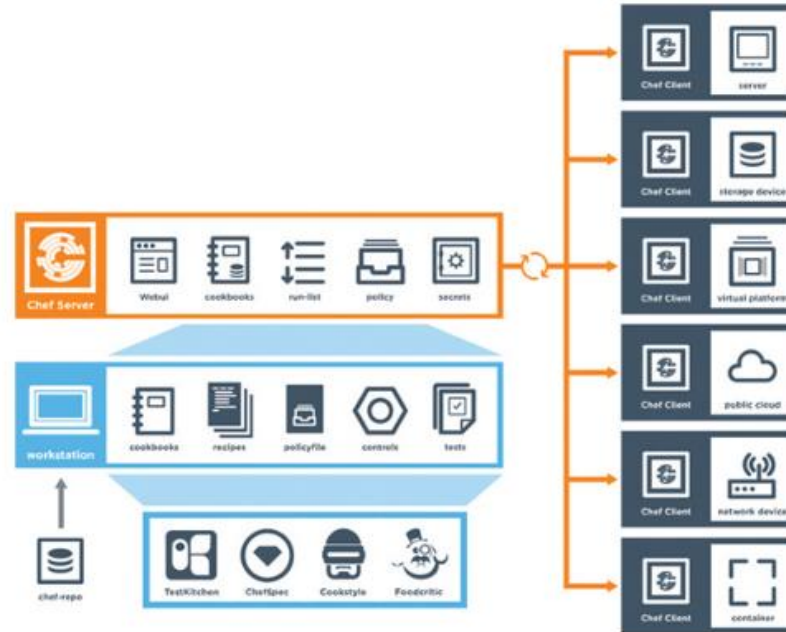
- Chef is an open source configuration management tool. Chef is written in Ruby and Erlang, using Ruby for writing code within Chef.
- Configuration management tools function in two different types of models: push and pull.
- Chef is similar to Puppet in several ways:
  - Both have free open source versions available
  - Both have paid enterprise versions available
  - Both manage code that needs to be updated and stored
  - Both manage devices or nodes to be configured
  - Both leverage a pull model
  - Both function as a client/server model

# Agent-Based Automation Tools

## Chef's Structure

- Chef's structure, terminology, and core components are different from those of Puppet.
- Figure 29-4 illustrates the high-level architecture of Chef and the communications path between the various areas within the Chef environment.

**Figure 29-4**  
*High-Level Chef Architecture*



# Agent-Based Automation Tools

## Chef Components

Although the core concepts of Puppet and Chef are similar, the terminology differs. Whereas Puppet has modules and manifests, Chef has cookbooks and recipes. Table 29-4 compares the components of Chef and Puppet and provides a brief description of each component.

Table 29-4 Puppet and Chef Comparison

Chef Components	Puppet Components	Description
Chef server	Puppet master	Server/master functions
Chef client	Puppet agent	Client/agent functions
Cookbook	Module	Collection of code or files
Recipe	Manifest	Code being deployed to make configuration changes
Workstation	Puppet console	Where users interact with configuration management tools and create code

## Chef Components (Cont.)

- Code is created on the Chef workstation. This code is stored in a file called a recipe
- Once a recipe is created on the workstation, it must be uploaded to the Chef server in order to be used in the environment. **knife** is the name of the command-line tool used to upload cookbooks to the Chef server.
- The command to execute an upload is **knife upload *cookbookname***.
- The Chef server can be hosted locally on the workstation, hosted remotely on a server, or hosted in the cloud.

There are four types of Chef server deployments:

- **Chef Solo** - The Chef server is hosted locally on the workstation.
- **Chef Client and Server** - This is a typical Chef deployment with distributed components.
- **Hosted Chef** - The Chef server is hosted in the cloud.
- **Private Chef** - All Chef components are within the same enterprise network.

- The Chef server sits in between the workstation and the nodes. All cookbooks are stored on the Chef server which holds all the tools necessary to transfer the node configurations to the Chef clients.
- OHAI collects the current state of a node to send the information back to the Chef server through the Chef client service. The Chef server then checks to see if there is any new configuration that needs to be on the node by comparing the information from the OHAI service to the cookbook or recipe.
- When a node needs a recipe, the Chef client service handles the communication back to the Chef server to signify the node's need for the updated configuration or recipe.



# Agent-Based Automation Tools

## Recipe File

Example 29-8 shows a recipe file constructed in Ruby.

**Example 29-8** *Chef demo\_install.rb Recipe*

```
#
# Cookbook Name:: cisco-cookbook
# Recipe:: demo_install
#
# Copyright (c) 2014-2017 Cisco and/or its affiliates.
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# In our recipes, due to the number of different parameters, we prefer to align
# the arguments in a single column rather than following rubocop's style.
```

```
Chef::Log.info('Demo cisco_command_config provider')

cisco_command_config 'loop42' do
  action :update
  command '
    interface loopback42
      description Peering for AS 42
      ip address 192.168.1.42/24
    '
end

cisco_command_config 'system-switchport-default' do
  command 'no system default switchport'
end

cisco_command_config 'feature_bgp' do
  command ' feature bgp'
end

cisco_command_config 'router_bgp_42' do
  action :update
  command '
    router bgp 42
      router-id 192.168.1.42
      address-family ipv4 unicast
        network 1.0.0.0/8
        redistribute static route-map bgp-statics
      neighbor 10.1.1.1
```

## Agent-Based Automation Tools

# SaltStack Overview

- SaltStack is built on Python with a Python interface so a user can program directly to SaltStack by using Python code.
- However, most of the instructions or states that get sent out to the nodes are written in YAML or a DSL. These are called Salt formulas.
- Another key difference from Puppet and Chef is SaltStack's overall architecture. SaltStack uses systems, which are divided into various categories. SaltStack has masters and minions.
- SaltStack uses a distributed messaging platform called 0MQ (ZeroMQ). SaltStack is an event-driven technology that has components called reactors and beacons. A reactor lives on the master and listens for any type of changes in the node or device that differ from the desired state or configuration.
- Beacons live on minions. If a configuration changes on a node, a beacon notifies the reactor on the master. This is the remote execution system and it helps determine whether the configuration is in the appropriate state on the minions. These actions are called jobs which when executed can be stored in an external database.

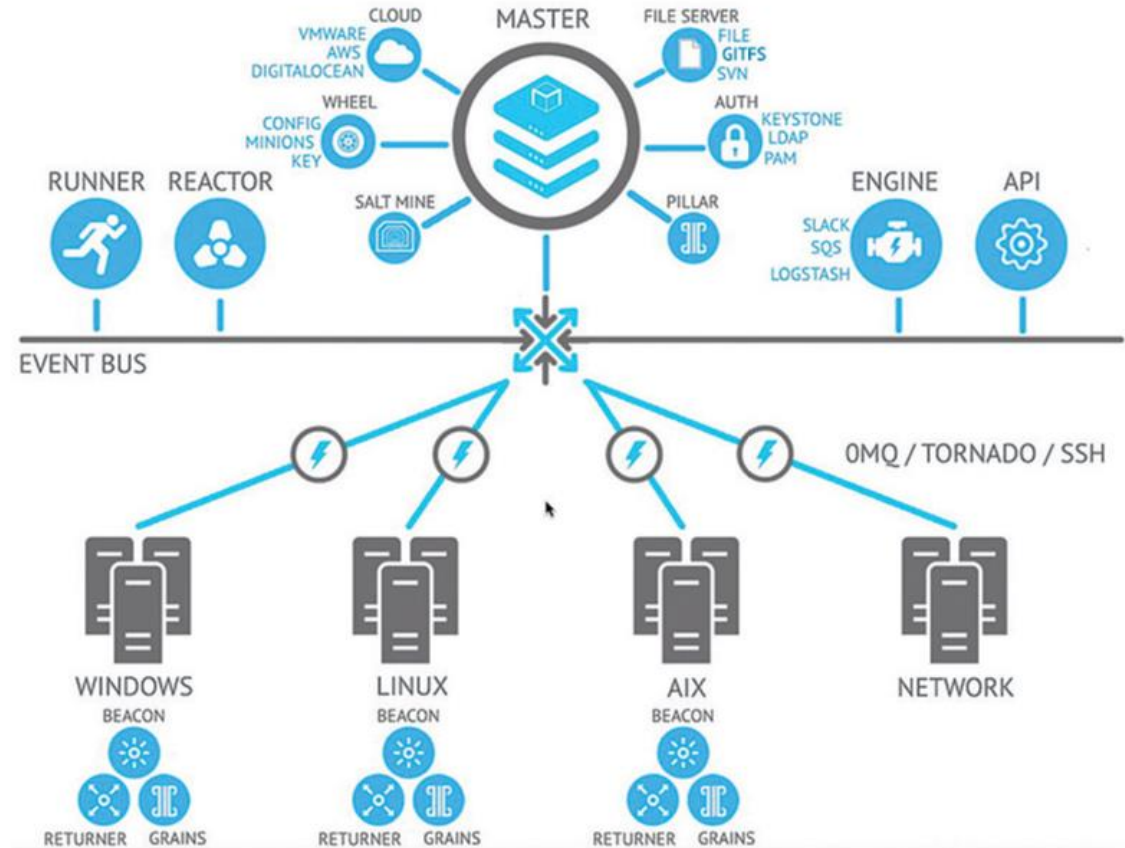
## SaltStack Overview (Cont.)

- SaltStack uses pillars and grains to control state and send configuration changes. SaltStack grains are run on the minions to gather system information to report back to the master. This information is typically gathered by the salt-minion daemon. Grains can provide specifics to the master about the host.
- Pillars store data that a minion can retrieve from the master. Minions can be assigned to pillars.
- Data can be stored for a specific node inside a pillar, keeping it separate from any other node that is not assigned to this particular pillar. Confidential data can be secured and only shared with assigned minions.

# Agent-Based Automation Tools

## SaltStack Architecture

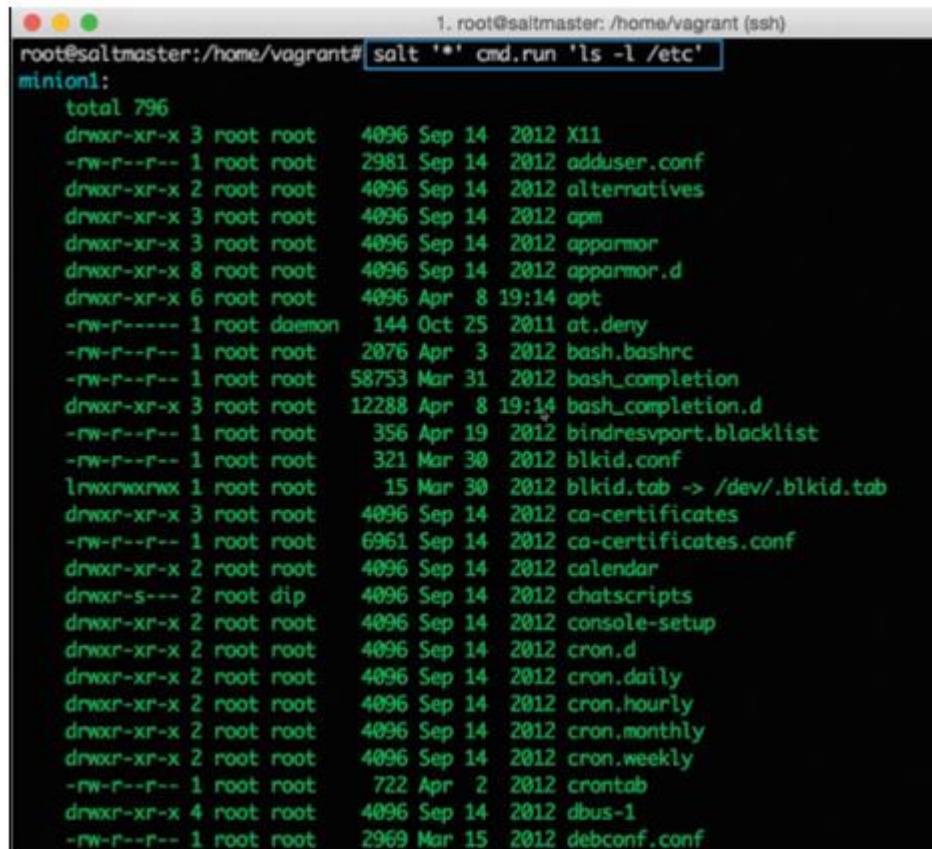
- SaltStack can scale to a very large number of devices. SaltStack also has an enterprise version and a GUI called SynDic.
- Figure 29-5 shows the overall architecture of SaltStack and its associated components.



# Agent-Based Automation Tools

## SaltStack CLI

- SaltStack has its own DSL. The SaltStack command structure contains *targets*, *commands*, and *arguments*.
- The target is the desired system that the command should run.
- The command structure uses the module.function syntax followed by the argument.
- An argument provides detail to the module and function that is being called on in the command.
- Figure 29-6 shows the correct SaltStack syntax as well as a command called cmd.run.



The image shows a terminal window with a title bar indicating the user is root@saltmaster in the directory /home/vagrant, connected via SSH. The prompt is root@saltmaster:/home/vagrant#. The command entered is salt '\*' cmd.run 'ls -l /etc'. The output shows the command was successful on the minion1, displaying the total size of 796 bytes and a list of files in /etc with their permissions, owner, group, size, date, and filename.

```
1. root@saltmaster: /home/vagrant (ssh)
root@saltmaster:/home/vagrant# salt '*' cmd.run 'ls -l /etc'
minion1:
total 796
drwxr-xr-x 3 root root 4096 Sep 14 2012 X11
-rw-r--r-- 1 root root 2981 Sep 14 2012 adduser.conf
drwxr-xr-x 2 root root 4096 Sep 14 2012 alternatives
drwxr-xr-x 3 root root 4096 Sep 14 2012 apm
drwxr-xr-x 3 root root 4096 Sep 14 2012 apparmor
drwxr-xr-x 8 root root 4096 Sep 14 2012 apparmor.d
drwxr-xr-x 6 root root 4096 Apr 8 19:14 apt
-rw-r----- 1 root daemon 144 Oct 25 2011 at.deny
-rw-r--r-- 1 root root 2076 Apr 3 2012 bash.bashrc
-rw-r--r-- 1 root root 58753 Mar 31 2012 bash_completion
drwxr-xr-x 3 root root 12288 Apr 8 19:14 bash_completion.d
-rw-r--r-- 1 root root 356 Apr 19 2012 bindresvport.blacklist
-rw-r--r-- 1 root root 321 Mar 30 2012 blkid.conf
lrwxrwxrwx 1 root root 15 Mar 30 2012 blkid.tab -> /dev/.blkid.tab
drwxr-xr-x 3 root root 4096 Sep 14 2012 ca-certificates
-rw-r--r-- 1 root root 6961 Sep 14 2012 ca-certificates.conf
drwxr-xr-x 2 root root 4096 Sep 14 2012 calendar
drwxr-s--- 2 root dip 4096 Sep 14 2012 chatscripts
drwxr-xr-x 2 root root 4096 Sep 14 2012 console-setup
drwxr-xr-x 2 root root 4096 Sep 14 2012 cron.d
drwxr-xr-x 2 root root 4096 Sep 14 2012 cron.daily
drwxr-xr-x 2 root root 4096 Sep 14 2012 cron.hourly
drwxr-xr-x 2 root root 4096 Sep 14 2012 cron.monthly
drwxr-xr-x 2 root root 4096 Sep 14 2012 cron.weekly
-rw-r--r-- 1 root root 722 Apr 2 2012 crontab
drwxr-xr-x 4 root root 4096 Sep 14 2012 dbus-1
-rw-r--r-- 1 root root 2969 Mar 15 2012 debconf.conf
```

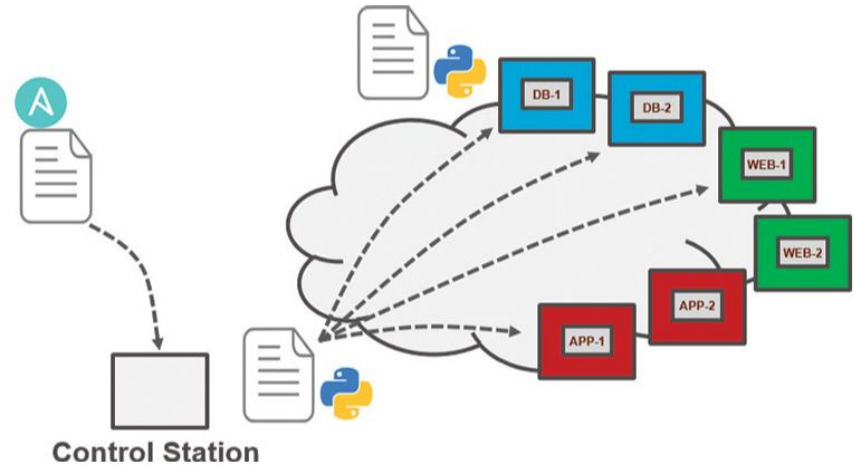
# Agentless Automation Tools

- This section covers a variety of agentless tools as well as some of the key concepts to help network operators decide which tool best suits their environment and business use cases.

# Agentless Automation Tools

## Ansible Overview

- Ansible is an automation tool that is capable of automating cloud provisioning, deployment of applications, and configuration management.
- Ansible is agentless, so no software needs to be installed on the client machines. Ansible communicates using SSH. Ansible can use built-in authorization escalation when it needs to raise the level of administrative control.
- Ansible sends all requests from a control station. Figure 29-8 illustrates the Ansible workflow.



# Ansible Playbook Components

- Ansible uses playbooks to deploy configuration changes or retrieve information from hosts within a network. An Ansible playbook is a structured sets of instructions. An Ansible playbook contains multiple plays, and each play contains the tasks that each player must accomplish in order for the particular play to be successful.
- Table 29-5 describes the components used in Ansible and provides some commonly used examples of them.

Components	Description	Use Case
Playbook	A set of plays for remote systems	Enforcing configuration and/or deployment steps
Play	A set of tasks applied to a single host or group of hosts	Grouping a set of hosts to apply policy or configuration to them.
Task	A call to an Ansible module	Logging in to a device to issue a <b>show</b> command to retrieve output.



# Agentless Automation Tools

## Ansible Playbooks in YAML

- Ansible playbooks are written using YAML (Yet Another Markup Language). Ansible YAML files usually begin with a series of three dashes (---) and end with a series of three periods (...). Example 29-9 shows a YAML file that contains a list of musical genres.
- YAML uses dictionaries that are similar to JSON dictionaries as they also use key/value pairs. Example 29-10 shows a YAML dictionary containing an employee record.

**Example 29-9** *YAML List Example*

```
---  
# List of music genres  
Music:  
  - Metal  
  - Rock  
  - Rap  
  - Country  
...
```

**Example 29-10** *YAML Dictionary Example*

```
---  
# HR Employee record  
Employee1:  
  Name: John Dough  
  Title: Developer  
  Nickname: Mr. DBug
```

## Agentless Automation Tools

# Ansible CLI Commands

Ansible has a CLI tool that can be used to run playbooks or ad hoc CLI commands on targeted hosts. This tool has very specific commands that you need to use to enable automation. The table below shows the most common Ansible CLI commands and associated use cases.

CLI Command	Use Case
ansible	Runs modules against targeted hosts
ansible-playbook	Runs playbooks
ansible-docs	Provides documentation on syntax and parameters in the CLI
ansible-pull	Changes Ansible clients from the default push model to the pull model
ansible-vault	Encrypts YAML files that contain sensitive data

# Agentless Automation Tools

## Ansible Configuration Example

- Example 29-14 shows an alternative version of the ConfigureInterface.yaml playbook named EIGRP\_Configuration\_Example.yaml, with EIGRP added, along with the ability to save the configuration by issuing a “write memory.”
- These tasks are accomplished by leveraging the ios\_command module in Ansible. This playbook adds the configuration shown here to the CSR1KV-1 router.

**Example 29-14** *Ansible EIGRP\_Configuration\_Example.yaml Playbook*

```
---
- hosts: CSR1KV-1

gather_facts: false
connection: local

tasks:
  - name: Configure GigabitEthernet2 Interface
    ios_config:
      lines:
        - description Configured by ANSIBLE!!!
        - ip address 10.1.1.1 255.255.255.0
        - no shutdown
      parents: interface GigabitEthernet2

    host: "{{ ansible_host }}"
    username: cisco
    password: testtest

  - name: CONFIG Gig3
    ios_config:
      lines:
        - description Configured By ANSIBLE!!!
        - no ip address
        - shutdown
      parents: interface GigabitEthernet3

    host: "{{ ansible_host }}"
    username: cisco
    password: testtest

  - name: CONFIG EIGRP 100
    ios_config:
      lines:
        - router eigrp 100
```

## Agentless Automation Tools

# Puppet Bolt

- Puppet Bolt allows you to leverage the power of Puppet without having to install a puppet master or puppet agents on devices or nodes.
- It connects to devices by using SSH or WinRM connections. Puppet Bolt is an open source tool that is based on the Ruby language and can be installed as a single package.
- Puppet Bolt allows you to execute a change or configuration immediately and then validate it. There are two ways to use Puppet Bolt:
  - **Orchestrator-driven tasks** - Orchestrator-driven tasks can leverage the Puppet architecture to use services to connect to devices. This design is meant for large-scale environments.
  - **Standalone tasks** - Standalone tasks are for connecting directly to devices or nodes to execute tasks and do not require any Puppet environment or components to be set up in order to realize the benefits and value of Puppet Bolt.

# Agentless Automation Tools

## Puppet Bolt Command Line

- Individual commands can be run from the command line by using the command **bolt command run** *command name* followed by the list of devices to run the command against.
- After a script is built, execute it from the command line against the remote devices that need to be configured, using the command **bolt script run** *script name* followed by the list of devices to run the script against. Figure 29-14 shows a list of some of the available commands for Puppet Bolt.

```
Usage: bolt <subcommand> <action> [options]

Available subcommands:
  bolt command run <command>      Run a command remotely
  bolt script run <script>         Upload a local script and run it remotely
  bolt task run <task> [params]    Run a Puppet task
  bolt plan run <plan> [params]    Run a Puppet task plan
  bolt file upload <src> <dest>    Upload a local file

where [options] are:
  -n, --nodes NODES               Node(s) to connect to in URI format [protocol://]host[:port]
                                   Eg. --nodes bolt.puppet.com
                                   Eg. --nodes localhost,ssh://nix.com:2222,winrm://windows.puppet.com

                                   * NODES can either be comma-separated, '@<file>' to read
                                   * nodes from a file, or '-' to read from stdin
                                   * Windows nodes must specify protocol with winrm://
                                   * protocol is 'ssh' by default, may be 'ssh' or 'winrm'
                                   * port is '22' by default for SSH, '5985' for winrm (Optional)
  -u, --user USER                 User to authenticate as (Optional)
  -p, --password [PASSWORD]       Password to authenticate with (Optional).
                                   Omit the value to prompt for the password.
                                   --private-key KEY      Private ssh key to authenticate with (Optional)
  -c, --concurrency CONCURRENCY   Maximum number of simultaneous connections (Optional, defaults to 100)
  --modulepath MODULES            List of directories containing modules, separated by :
  --params PARAMETERS            Parameters to a task or plan
  --format FORMAT                Output format to use: human or json
  -k, --insecure                 Whether to connect insecurely
  --transport TRANSPORT          Specify a default transport: ssh, winrm, pcip
  --run-as USER                 User to run as using privilege escalation
  --sudo [PROGRAM]              Program to execute for privilege escalation. Currently only sudo is supported.
  --sudo-password [PASSWORD]    Password for privilege escalation
  --[no-]tty                    Request a pseudo TTY on nodes that support it
  -h, --help                    Display help
  --verbose                     Display verbose logging
  --debug                      Display debug logging
  --version                    Display the version
```

# Puppet Bolt Command Line

- Puppet Bolt tasks use an API to retrieve data between Puppet Bolt and the remote device.
- Tasks are part of the Puppet modules and use the naming structure *modulename::taskfilename*.
- Tasks can be called from the command line much like commands and scripts. You use the command `bolt task run modulename::taskfilename` to invoke these tasks from the command line.
- The *modulename::taskfilename* naming structure allows the tasks to be shared with other users on Puppet Forge.
- A task is commonly accompanied by a metadata file that is in JSON format. A JSON metadata file contains information about a task, how to run the task, and any comments about how the file is written.

# SaltStack Salt SSH (Server-Only Mode)

- SaltStack offers an agentless option called Salt SSH that allows users to run Salt commands without having to install a minion on the remote device or node. The main requirements to use Salt SSH are that the remote system must have SSH enabled and Python installed.
- Salt SSH can work in conjunction with the master/minion environment, or it can be used completely agentless across the environment. By default, Salt SSH uses roster files to store connection information for any host that doesn't have a minion installed. Example 29-16 shows the content structure of this file.

**Example 29-16** *Salt SSH Roster File*

```
managed:  
  host: 192.168.10.1  
  user: admin
```

# Agentless Automation Tools

## Comparing Tools

Table 29-7 provides a high-level comparison of the tools covered in this chapter.

Factor	Puppet	Chef	Ansible	SaltStack
Architecture	Puppet masters and puppet agents	Chef server and Chef clients	Control station and remote hosts	Salt master and minions
Language	Puppet DSL	Ruby DSL	YAML	YAML
Terminology	Modules and Manifests	Cookbooks and recipes	Playbooks and plays	Pillars and grains
Support for large-scale deployments	Yes	Yes	Yes	Yes
Agentless version	Puppet Bolt	N/A	Yes	Salt SSH



# Prepare for the Exam

## Prepare for the Exam

# Key Topics for Chapter 29

Description
EEM applets and configuration
Puppet
Chef
SaltStack (agent and server mode)
Ansible
Puppet Bolt
SaltStack SSH (server-only mode)
High-Level Configuration Management and Automation Tool Comparison

## Prepare for the Exam

# Key Terms for Chapter 29

Term
Cookbooks
Embedded Event Manager (EEM)
Grain
Manifest
Module
Pillar
Play
Playbook
Recipe
Tcl

