DevOps Tools
(Chef, Puppet, Saltstack and Ansible Comparison)

> "The tools we use reinforce the behavior; the behavior reinforces the tool. Thus, if you want to change your behavior, change your tools."
>
> – Adam Jacob, CTO, Chef

Before we start comparing DevOps tool. Let's go through few basic concepts…

What is Idempotency?

Different vendors define idempotence in slightly different ways. OpsCode defines it to mean that a script "can run multiple times on the same system and the results will always be identical." Ansible states that its scripts "will seek to avoid changes to the system unless a change needs to be made." This mean that their scripts can be run repeatedly and will only change something when the script and actual server configuration differ.

Idempotency ensures a system won't stray from its designated desired state, sending alerts to systems administrators if a requested change would cause a disruption.

In Chef word:
Convergence and idempotence are not Chef-specific. They're generally attributed to configuration management theory, though have use in other fields, notably mathematics.
The way we (Chef) talk about this is that Chef takes idempotent actions to converge the system to the state declared by the various resources. Every resource in Chef is declarative, and performs a test about the current state of the resource, and then repairs the system to match that.

Idempotent Behaviour -
Configuration management tools keep track of the state of resources in order to avoid repeating tasks that were executed before. If a package was already installed, the tool won't try to install it again. The objective is that after each provisioning run the system reaches (or keeps) the desired state, even if you run it multiple times. This is what characterizes these tools as having an *idempotent behavior*. This behavior is not necessarily enforced in all cases, though.

Benefits of idempotent configurations-

In a DevOps organization, the systems administrator can provision a highly defined and audited workload in the operational environment using configuration management, CD and DevOps tools, such as Jenkins and Chef or Puppet. However, without an idempotent provisioning tool, the administrator cannot guarantee that the same results will occur every time that workload

is provisioned in production. This lack of repeatability affects workload portability, as in multicloud environments or cloud migrations.

Even without portability requirements, users find that some configuration management tools result in major issues caused by phantom remnants of previous scripts and processes. Phantom script fragments affect memory and storage; sometimes scripts slow down or time out due to these artefacts.

An idempotent configuration management system ensures that such problems do not occur. The idea is to turn any systems administrator can: Decide what is required and "make it so."

Idempotence vs Convergence in Configuration Management -

About idempotence vs convergence: as you rightly point out, they are not the same, and there are very important differences. Idempotence is an operation that leaves the system unmodified if it's already in the desired state, whereas convergence is the property of a system of not making unnecessary changes (nor operations). The end result may be the same, but CFEngine emphasizes convergence over idempotence, and has A LOT of built-in logic to automatically avoid performing unnecessary operations.

What is Orchestration?

Orchestration basically comes to automate processes and workflows whereas automation basically automates specific tasks. Orchestration describes automated arrangement, coordination, and management of complex computer systems, and services. In reality, Orchestration is largely the effect of automation or systems deploying elements of control theory.

This usage of orchestration is often discussed in the context of service oriented architecture, virtualization, provisioning, Converged Infrastructure and dynamic datacenter topics. Orchestration in this sense is about aligning the business request with the applications, data and infrastructure.

Cloud service orchestration therefore is the:

- Composing of architecture, tools and processes by humans to deliver a defined service
- Stitching of software and hardware components together to deliver a defined Service
- Connecting and automating of work flows when applicable to deliver a defined service

It is critical in the delivery of cloud services because:

- Cloud is all about scale – automated work flows are essential [weasel words]
- Cloud service delivery includes fulfillment assurance and billing
- Cloud services delivery entails work flows in various technical and business domains

What is RESTful -

Representational state transfer (REST) or RESTful web services are one way of providing interoperability between computer systems on the Internet. REST-compliant Web services allow

requesting systems to access and manipulate textual representations of web services using a uniform and predefined set of **stateless** operations. Other forms of Web service exist, which expose their own arbitrary sets of operations such as WSDL and SOAP.

What is Domain-specific language (DSL) –

DSLs are small, expressive programming languages custom designed for specific tasks.
A DSL is like a mini language inside of a language. It's a specific syntax for describing a domain/area of your application. DSL can be external, internal or embedded.

Few DSL examples....

```
service 'apache' do
  supports :restart => true, :reload => true
  action: enable
end
```

For example, using an `if` statement with the `platform` node attribute:

```
if node['platform'] == 'ubuntu'
  # do ubuntu things
end

case node['platform']
when 'debian', 'ubuntu'
  # do debian/ubuntu things
when 'redhat', 'centos', 'fedora'
  # do redhat/centos/fedora things
end
```

The **Recipe DSL** is a **Ruby DSL** that is primarily used to declare resources from within a recipe. The Recipe DSL also helps ensure that recipes interact with nodes (and node properties) in the desired manner. Most of the methods in the Recipe DSL are used to find a specific parameter and then tell the chef-client what action(s) to take, based on whether that parameter is present on a node.

Because the Recipe DSL is a Ruby DSL, then anything that can be done using Ruby can also be done in a recipe, including `if` and `case` statements, using the `include?` Ruby method, including recipes in recipes, and checking for dependencies. Ref- https://docs.chef.io/dsl_recipe.html

*** Configuration management (CM) -**

Configuration management (CM) is a governance and systems engineering process that ensures the proper accounting of an enterprise's configuration items (CIs) and of the interrelationship between them in an operational environment. CIs consist of enterprise devices, hardware and software, and capabilities necessary to deliver services for an organization.
Configuration management is one of the operational processes identified in the IT Infrastructure

Library (ITIL) service management framework, although an enterprise need not adopt the ITIL framework to perform configuration management. Configuration management is referred to as Service Asset and Configuration Management in ITIL V3.

Key benefits of configuration management -

A configuration management tool can improve the organization's change-impact analysis, reducing the outages caused by production changes.
Configuration management combines valid service models, CI interdependency mapping and correlations made between CIs and requests for changes to help restore services faster during an outage.

A configuration management system provides audit and compliance support in historical operational accounting of devices and their utilization and modifications.

There are several potential business benefits to consider in a configuration management plan. Lower operational costs result from better understanding the total cost of ownership for the current IT service model. Organizational security improves when the company can detect unauthorized changes. The correlation between CIs and operational processes -- and how these support business services or affect key performance indicators -- can enable more informed business decisions.

Change and asset management -

Change management and configuration management are complementary, but not identical processes. Change management seeks to govern and ensure only authorized modifications are made to an item, while mitigating risk and impact to the whole. Change management is implemented via an established business process, such as a change advisory board reviewing requests for change. Configuration management deals with the identification, maintenance, reporting on and verification of items and their interrelationships.

Configuration management also differs from asset management in that it does not seek to manage financial accounting aspects of the CI.
Configuration and asset management govern the items under different lifecycles. For example, asset management manages a server from procurement through disposal.

Software configuration management -

Configuration management is also used in software development and deployment, where it is called software or unified configuration management (SCM or UCM). Developers and others involved in the project can use SCM to keep track of artifacts, including source code, documentation, problems, changes requested and changes made. Software configuration management provides structure to development steps, such as establishing baselines and reporting on the status of development processes.

- **Messaging protocols used in Software Configuration management (CM)-**

1) **ZeroMQ -**

ZeroMQ (also spelled 0MQ or ZMQ) is a high-performance asynchronous messaging library, aimed at use in distributed or concurrent applications. It provides a message queue, but unlike message-oriented middleware, a ZeroMQ system can run without a dedicated message broker.

2) **STOMP -**

MQTT (MQ Telemetry Transport or Message Queue Telemetry Transport) is an ISO standard (ISO/IEC PRF 20922) publish-subscribe-based "lightweight" messaging protocol for use on top of the TCP/IP protocol.
STOMP is a text-based messaging protocol emphasising (protocol) simplicity. It defines little in the way of messaging semantics, but is easy to implement and very easy to implement partially (it's the only protocol that can be used by hand over telnet). The publish-subscribe messaging pattern requires a message broker. The broker is responsible for distributing messages to interested clients based on the topic of a message.

MQTT is the only one of these three protocols to be text-based, making it more analogous to HTTP in terms of how it looks under the covers. Like AMQP, STOMP provides a message (or frame) header with properties, and a frame body. The design principles here were to create something simple, and widely-interoperable. For example, it's possible to connect to a STOMP broker using something as simple as a telnet client.

STOMP does not, however, deal in queues and topics—it uses a SEND semantic with a "destination" string. The broker must map onto something that it understands internally such as a topic, queue, or exchange. Consumers then SUBSCRIBE to those destinations. Since those destinations are not mandated in the specification, different brokers may support different flavours of destination. So, it's not always straightforward to port code between brokers.

3) **MQTT -**

MQTT is a binary protocol emphasising lightweight publish / subscribe messaging,

targeted towards clients in constrained devices. It has well defined messaging semantics for publish / subscribe, but not for other messaging idioms.

The design principles and aims of MQTT are much more simple and focused than those of AMQP—it provides publish-and-subscribe messaging (no queues, in spite of the name) and was specifically designed for resource-constrained devices and low bandwidth, high latency networks such as dial up lines and satellite links, for example. Basically, it can be used effectively in embedded systems. MQTT's strengths are simplicity (just five API methods), a compact binary packet payload (no message properties, compressed headers, much less verbose than something text-based like HTTP), and it makes a good fit for simple push messaging scenarios such as temperature updates, stock price tickers, oil pressure feeds or mobile notifications.

4) **AMQP -**

AMQP is a binary, application layer protocol, designed to efficiently support a wide variety of messaging applications and communication patterns, interoperability between different vendors. It provides flow controlled, message-oriented communication with message-delivery guarantees such as at-most-once (where each message is delivered once or never), at-least-once (where each message is certain to be delivered, but may do so multiple times) and exactly-once (where the message will always certainly arrive and do so only once), and authentication and/or encryption based on SASL and/or TLS. It assumes an underlying reliable transport layer protocol such as Transmission Control Protocol (TCP).

The AMQP specification is defined in several layers: (i) a type system, (ii) a symmetric, asynchronous protocol for the transfer of messages from one process to another, (iii) a standard, extensible message format and (iv) a set of standardised but extensible 'messaging capabilities.'

Two of the most important reasons to use AMQP are **reliability and interoperability**. As the name implies, it provides a wide range of features related to messaging, including *reliable queuing, topic-based publish-and-subscribe messaging, flexible routing, transactions, and security.* AMQP exchanges route messages directly—in fanout form, by topic, and based on headers. There's a lot of fine-grained control possible with such a rich feature set. You can restrict access to queues, manage their depth, and more.

- ## Choosing Your Messaging Protocol: AMQP, MQTT or STOMP -

Application architects need to use a messaging broker to speed and scale their applications, particularly in the cloud. Even once you select your messaging middleware application, application developers need to then select the protocol. Understanding the subtle differences between them can be difficult.

- **Messaging (Broker) Software -**

Messaging broker is used to speed and scale applications, particularly in the cloud.

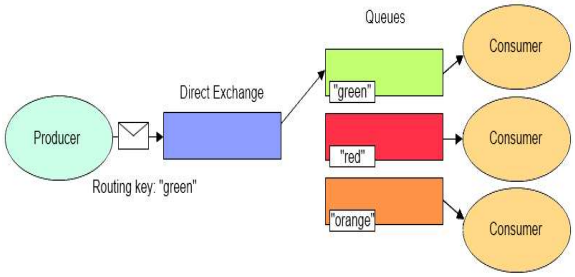
a) **RabbitMQ -**
RabbitMQ is open source message broker software. RabbitMQ was originally developed to support AMQP. Now also support STOMP and MQTT.

b) **Apache ActiveMQ -**
Apache ActiveMQ is an open source message broker written in Java together with a full Java Message Service (JMS) client. Supports STOMP, MQTT, AMQP, RESTful.

- **Message Queue Types -**

Message Exchanges control the routing of messages to queues. Each exchange type defines a specific routing algorithm which the server uses to determine which bound queues a published message should be routed to.

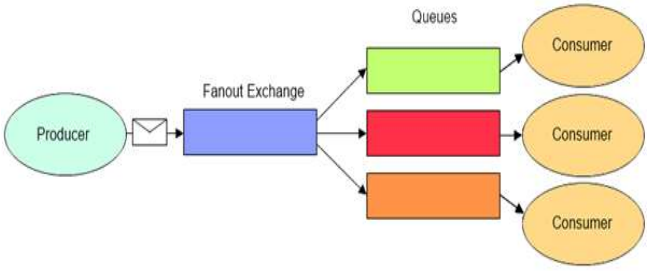
A) Direct Exchange-
The Direct exchange type routes messages with a routing key equal to the routing key declared by the binding queue. The Direct exchange type is useful when you would like to distinguish messages published to the same exchange using a simple string.



B) Fanout Exchanges-
The Fanout exchange type routes messages to all bound queues indiscriminately. If a routing key is

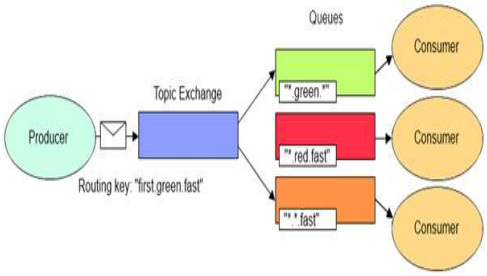
provided, it will simply be ignored.

Fanout exchange type is useful for facilitating the publish-subscribe pattern. When using the fanout exchange type, different queues can be declared to handle messages in different ways.



C) Topic Exchanges-

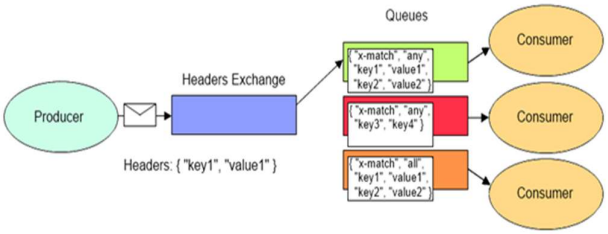
Topic exchange type routes messages to queues whose routing key matches all, or a portion of a routing key. With topic exchanges, messages are published with routing keys containing a series of words separated by a dot (e.g. “word1.word2.word3”). Queues binding to a topic exchange supply a matching pattern for the server to use when routing the message.



D) Headers Exchanges-

The Headers exchange type routes messages based upon a matching of message headers to the expected headers specified by the binding queue.

The Headers exchange type is useful for directing messages which may contain a subset of known criteria where the order is not established and provides a more convenient way of matching based upon the use of complex types as the matching criteria (i.e. a serialized object).



- **Message Sequence Guarantees-**

AMQP core specification explains the conditions under which ordering is guaranteed: messages published in one channel, passing through one exchange and one queue and one outgoing channel will be received in the same order that they were sent. RabbitMQ offers stronger guarantees since release 2.7.0.

What is MCollective (Marionette Collective)?

The Marionette Collective, also known as MCollective, is a framework for building server **orchestration** or **parallel job-execution systems**. Most users programmatically execute administrative tasks on clusters of servers.

MCollective has some unique strengths for working with large numbers of servers:

- Instead of relying on a static list of hosts to command, it uses metadata-based discovery and filtering. It can use a rich data source like PuppetDB, or can perform real-time discovery across the network.
- Instead of directly connecting to each host (which can be resource-intensive and slow), it uses publish/subscribe middleware to communicate in parallel with many hosts at once.

MCollective is Ruby framework for writing Orchestration systems. MCollective needs a publish/subscribe middleware system of some kind for all communications. When deploying MCollective, you need to:

- Pick a middleware type
- Get a connector plugin that supports it (note that ActiveMQ and RabbitMQ plugins are already included with MCollective's core install)
- Deploy and configure the middleware server(s)
- Configure the connector plugin on all MCollective servers and clients

*** RabbitMQ** is an open-source message broker written in Erlang; MCollective talks to RabbitMQ using the Stomp protocol. Although it works well with MCollective, it isn't as thoroughly tested with it as ActiveMQ is, so if your site has no preference, you should default to ActiveMQ.

*** Apache ActiveMQ** is an open-source message broker that runs on the JVM; typically, it's installed with a wrapper script and init script that allow it to be managed as a normal OS service. MCollective talks to ActiveMQ using the Stomp protocol.

NOTE: **MCollective's requirement for a STOMP server and ActiveMQ is recommended for use with MCollective.**

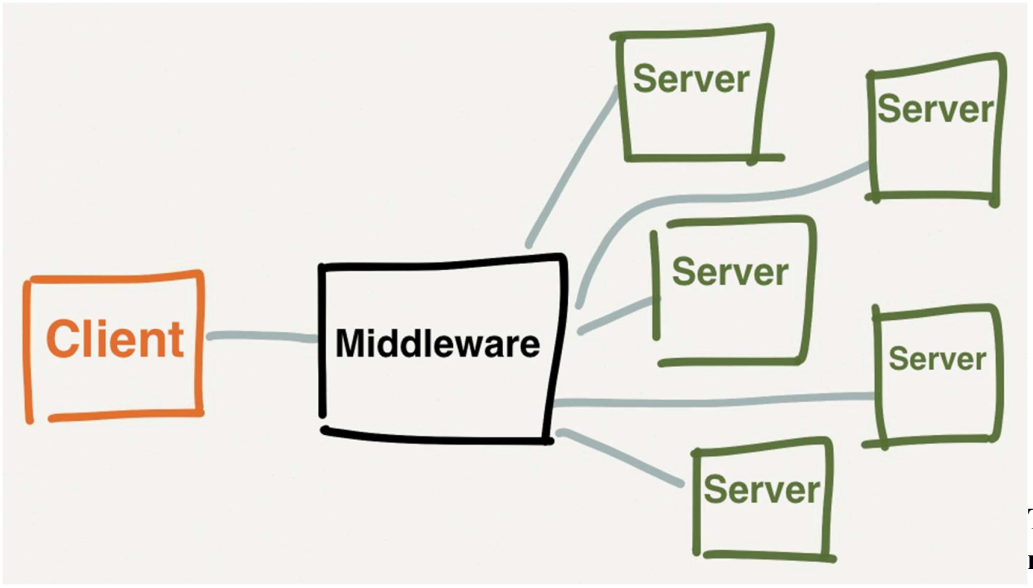
Although it often seems that MCollective goes hand-in-hand with Puppet, there's no technical limitation behind this. It can quite readily be used with any other configuration management tool, or none at all. The plugins supporting integration with Puppet may be a little more mature today, but users of Chef won't be missing out on anything.

What is MCollective, and What Does It Allow You to Do?

- Interact with clusters of servers, whether in small groups or very large deployments.
- Use a broadcast paradigm to distribute requests. All servers receive all requests at the same time, requests have filters attached, and only servers matching the filter will act on requests. There is no central asset database to go out of sync, because the network is the only true source.
- Break free from identifying devices through complex host-naming conventions, and instead use a rich set of metadata provided by each machine — from Puppet, **Facter**, or other sources — to address them.
- Use simple command-line tools to call remote agents.
- Write custom reports about your infrastructure.
- Use agent plugins to manage packages, services, and other common components created by the community.
- Write simple RPC style agents, clients, and web UIs in Ruby.
- Extremely pluggable and adaptable to local needs.
- Leverage rich authentication and authorization models in middleware systems as a first line of control.
- Include fine-grained authentication using SSL or RSA, authorization, and request auditing. For more information, see the Security Overview.
- Re-use middleware features for clustering, routing, and network isolation to realize secure and scalable configurations.

Mcollective Architecture-

The architecture of MCollective is based around three main components: **servers, clients,** and the**middleware.** Servers and clients also use various sub-components, which are mentioned in their respective sections.



Terminology notes:

If you're familiar with Puppet, note that MCollective uses the term "server" a bit differently.

- Instead of thinking "web server" or "puppet master server," think "SSH server." These are machines that mainly perform some other business purpose, but are also listening for MCollective requests.
- From the user's perspective, servers accept inbound requests and react to them. From the middleware's perspective, servers are just another kind of client — they proactively initiate a connection and subscribe to the types of messages they care about.

The term "agent" is also different. In Puppet, the agent is a daemon that fetches and applies configurations — the equivalent of the `mcollectived` server daemon. In MCollective, an **agent is just a bundle of actions distributed** as a plugin.

MCollective Servers -
An MCollective server (often just called a "node") is a computer that can be controlled via MCollective. Servers run the MCollective daemon (mcollectived), and have any number of agent plugins installed.

Mcollective Clients -
An MCollective client can send requests to any number of servers, use a security plugin to encode and sign the request, and use a connector plugin to publish it. These plugins must match the security and connector used by the servers. The client can also receive replies from servers and format the response for a user or some other system. The most common client is the mco command-line client, which can be used interactively or in scripts. You can also write clients in Ruby, such as backends for GUI apps or glue in a reactive infrastructure.

- **Software configuration management -**

1) Chef - (https://www.chef.io/chef/ **)**

Chef is a configuration management tool written in Erlang, and uses a pure Ruby DSL for writing configuration "recipes". These recipes contain resources that should be put into the declared state. Chef can be used as a client–server tool, or used in "solo" mode.

Chef Component-

Chef can be used in Client Server Mode (Chef-server needed) and Solo Mode (only chef-client need to install). In Solo mode, you can run cookbook on remote servers using any or **knife+ssh**, simply **ssh** remotely (run chef-solo command which is part of chef-client).

Main component of Chef are Chef-server (only required in Client-server model), Chef-Client, Shelf and Knife.

The **Chef server** acts as a hub of information. Cookbooks and policy settings are uploaded to the Chef server by users from workstations. (Policy settings may also be maintained from the Chef server itself, via the Chef management console web user interface.).

- ***Every request made by the chef-client*** *to the Chef server must be an authenticated request using the Chef server API and a private key*. When the chef-client makes a request to the Chef server, the chef-client authenticates each request using a private key located in /etc/chef/client.pem.
- Public key encryption is used by the Chef server. The public key is stored on the Chef server, while the private key is returned to the user for safe keeping. (The private key is a .pem file located in the .chef directory or in /etc/chef.)

A **chef-client** is installed on every node that is under management by Chef. The chef-client performs all the configuration tasks that are specified by the run-list and will pull down any required configuration data from the Chef server as it is needed during the chef-client run.

- ***RSA public key-pairs*** *are used to authenticate the chef-client with the Chef server every time a chef-client needs access to data that is stored on the Chef server.* This prevents any node from accessing data that it shouldn't and it ensures that only nodes that are properly registered with the Chef server can be managed.
- As part of every chef-client run, the chef-client authenticates to the Chef server using an RSA private key and the Chef server API.
- Chef server 12 enables SSL verification by default for all requests made to the server, such as those made by knife and the chef-client.

Knife: Use the knife command-line tool to interact with nodes or work with objects on the Chef server. Use the knife ssh subcommand to invoke SSH commands (in parallel) on a subset of nodes within an organization, based on the results of a search query made to the Chef server.
NOTE: Changed in Chef Client 12.19 to support ed25519 keys for ssh connections.

Chef-Shelf: is the repository structure in which cookbooks are authored, tested, and maintained:

- Cookbooks contain recipes, attributes, custom resources, libraries, files, templates, tests, and metadata
- The chef-repo should be synchronized with a version control system (such as git), and then managed as if it were source code

The directory structure within the chef-repo varies. Some organizations prefer to keep all of their cookbooks in a single chef-repo, while other organizations prefer to use a chef-repo for every cookbook.
Shelf is used to store cookbook content—files, templates, and so on—that have been uploaded to the Chef server as part of a cookbook version. Cookbook content is stored by content checksum. If two different cookbooks or different versions of the same cookbook include the same file or template, Bookshelf will store that file only once. The cookbook content managed by Bookshelf is stored in flat files and is separated from the Chef server and search index repositories.
All cookbooks are stored in a dedicated repository.

Chef-Solo:

chef-solo is a command that executes chef-client in a way that does not require the Chef server in order to converge cookbooks. chef-solo uses chef-client's Chef local mode, and **does not support** the following functionality present in chef-client / server configurations:

- Centralized distribution of cookbooks
- A centralized API that interacts with and integrates infrastructure components
- Authentication or authorization

Chef Client has built in Ohai is a tool that is used to detect attributes on a node, and then provide these attributes to the chef-client at the start of every chef-client run. Ohai is required by the chef-

client and must be present on a node. (Ohai is installed on a node as part of the chef-client install process.)

The types of attributes Ohai collects include (but are not limited to):

- Platform details
- Network usage
- Memory usage
- CPU data
- Kernel data
- Host names
- Fully qualified domain names
- Virtualization data
- Cloud provider metadata
- Other configuration details

What is Cookbook and Recipe in Chef?

A cookbook is the fundamental unit of configuration and policy distribution. A cookbook defines a scenario and contains everything that is required to support that scenario:

- Recipes that specify the resources to use and the order in which they are to be applied
- Attribute values
- File distributions
- Templates
- Extensions to Chef, such as custom resources and libraries

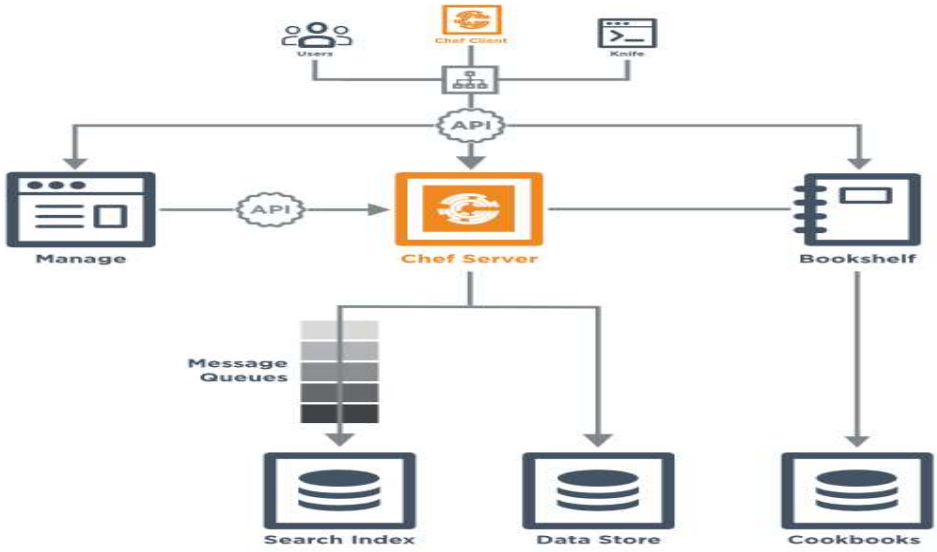
Ruby is the programming language that is the authoring syntax for cookbooks. Most recipes are simple patterns (blocks that define properties and values that map to specific configuration items like packages, files, services, templates, and users). The full power of Ruby is available for when you need a programming language.

Message Queues -

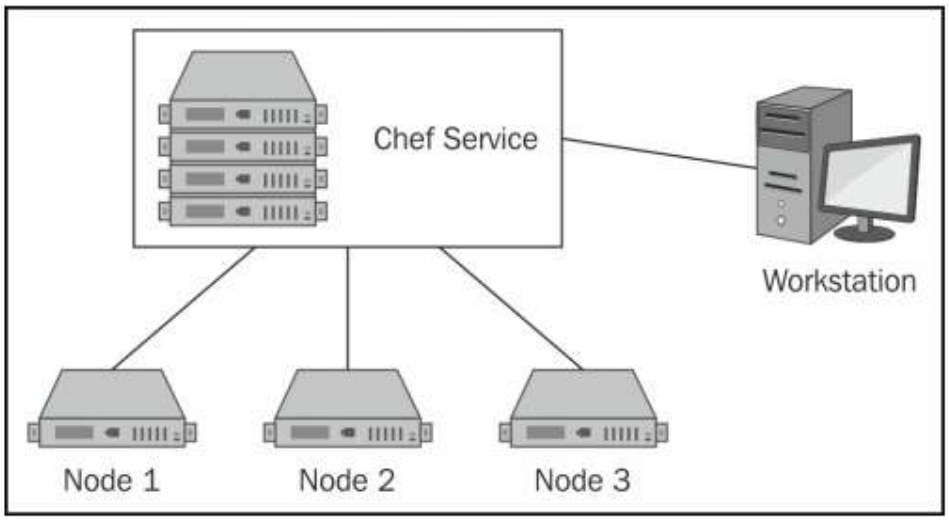
Messages are sent to the search index using the following components:
RabbitMQ is used as the message queue for the Chef server. All items that will be added to the search index repository are first added to a queue.
chef-expander is used to pull messages from the RabbitMQ queue, process them into the required format, and then post them to chef-solr for indexing.
chef-solr wraps Apache Solr and exposes its REST API for indexing and search.

Load Balancer - Nginx is an open-source HTTP and reverse proxy server that is used as the front-end load balancer for the Chef server. All requests to the Chef server API are routed through Nginx.

Chef Architecture –



The following diagram represents the Chef architecture:



The nodes communicate with the Chef service over HTTP (preferably HTTPS) using the `chef-client` script provided as part of the Chef client installation. This is a Ruby script that is responsible for connecting to the configured Chef service (self-hosted or using hosted Chef) and downloading the run list that is configured for that node along with any cookbooks and configuration data it needs. Once it has done that, `chef-client` will evaluate the run list in order to execute the recipes in the order in which they were specified.

The workstation also communicates with the Chef service using HTTP(S), but its job is a bit different. The workstation is where a system administrator will use the command-line utilities to interact with the data stored in the Chef service. From there, the administrator can read and modify

any of the system data as JSON, perform searches, and interact with the nodes through the `knife` command-line utility.

***Chef and Orchestration -**

When talking about the management of complex systems, orchestration is always a hot topic. This is because orchestration is often seen as the easiest way to represent and model complex systems, as well as provide a path to delivering complex systems.

Most often orchestration is represented through a topology model. What is a topology model you ask? A description of the order-of-operations across a group of machines. A common example is provisioning a database, cache layer, multiple application servers, web servers, and load balancer(s). This model will include distinct technology components that must interact, are interdependent, and more often than not the provisioning is accomplished through a very specific order.

Although orchestration solutions have been around for a long time, very rarely have they fit the bill. This is one of the reasons why Chef has opted to take a different approach, which we see proving more functional in the real world. In other words, we've worked hard to create a solution that solves the problem consistently in a wide range of complex, real world implementations.

At the core of ensuring a robust orchestration mechanism is following the principles that have made our core product Chef great: scale, idempotency, flexibility, and test-driven automation.

2) Puppet – (https://puppet.com **)**

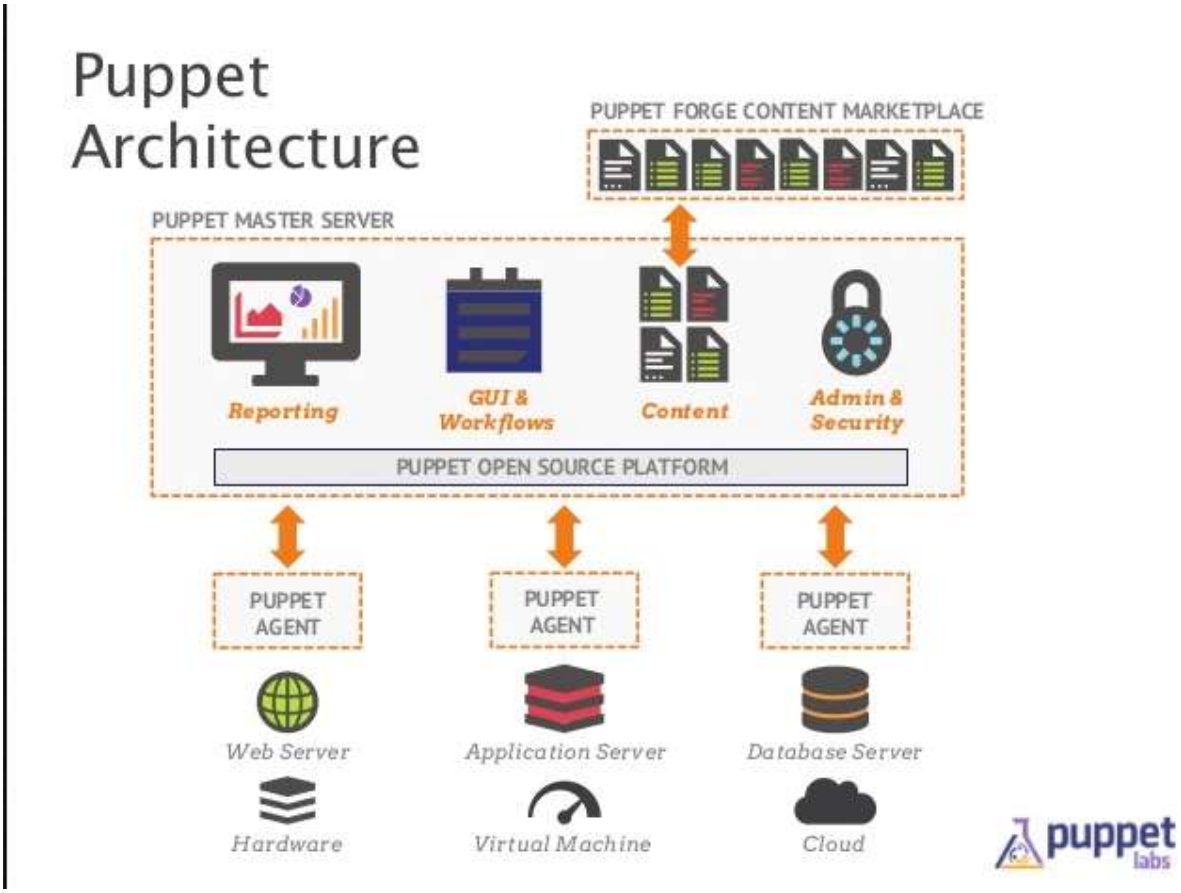
Puppet consists of a custom declarative language to describe system configuration, distributed using the client–server paradigm (using XML-RPC protocol in older versions, with a recent switch to REST), and a library to realize the configuration. The resource abstraction layer enables administrators to describe the configuration in high-level terms, such as users, services and packages. Puppet will then ensure the server's state matches the description. There was brief support in Puppet for using a pure Ruby DSL as an alternative configuration language starting at version 2.6.0. However this feature was deprecated beginning with version 3.1

Puppet Architecture:

- **Puppet Master (Server)** - is the JVM application that provides Puppet's core HTTPS services. Whenever Puppet agent checks in to request a configuration catalog for a node, it contacts Puppet Server.
- **Puppet agent** – The first component encountered in a Puppet run is the `agent` process. This was traditionally a separate executable called `puppetd`, but in version 2.6 we reduced down to one executable so now it is invoked with `puppet agent`, akin to how Git works. The agent has little functionality of its own; it is primarily configuration and code that

implements the client-side aspects of the above-described workflow.

- **Facter** - is a system profiling tool. It collects a plethora of system information like operating system, network interfaces, uptime, and so much more. Facter's facts can be used when writing Puppet code so your code can always do the right thing without any investigative logic required.
- **Puppet Modules**- Puppet features a declarative **Domain Specific Language** (**DSL**), which expresses the desired state and properties of the managed resources. Puppet code is written in `manifests`, which are simple text files with a `.pp` extension. Resources can be grouped in classes (do not consider them as classes as in OOP; they aren't). Classes and all the files needed to define the required configurations are generally placed in modules, which are directories structured in a standard way that are supposed to manage specific applications or a system's features (there are modules to manage Apache, MySQL, sudo, sysctl, networking, and so on). **Puppet is ruby based.**
- **PuppetDB** collects data generated throughout your Puppet infrastructure. It enables advanced Puppet features like exported resources, and is the database from which the various components and services in PE access data. Puppet agent run reports are stored in PuppetDB.



- **MCollective** - is a distributed task-based orchestration system. Nodes with MCollective listen for commands over a message bus and independently take action when they hear an authorized request. This lets you investigate and command your infrastructure in real time

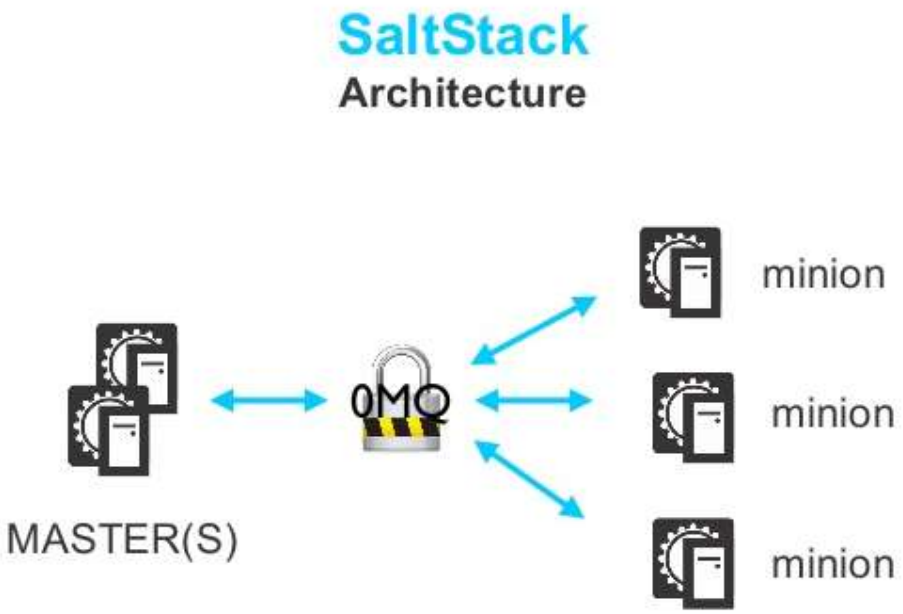
without relying on a central inventory.

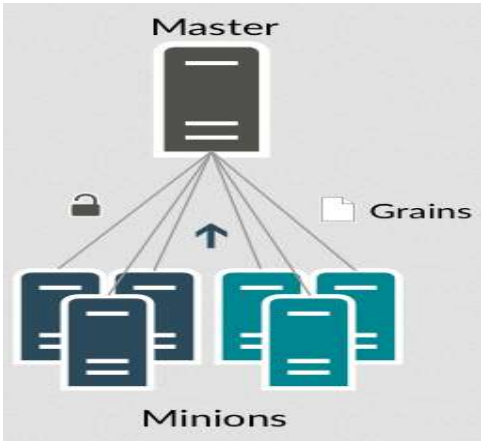
3) Slat – (https://saltstack.com **)**

Salt started out as a tool for remote server management. As its usage has grown, it has gained a number of extended features, including a more comprehensive mechanism for host configuration. This is a relatively new feature facilitated through the Salt States component. With the traction that Salt has gotten in the last bit, the support for more features and platforms might continue to grow.

SaltStack was originally created to be an extremely fast, scalable and powerful remote execution engine for efficient control of distributed infrastructure, code and data. Later, configuration management was built on top of the remote execution engine and leverages the same capabilities of the core.

Saltstack Architecture and component -





Salt Master –

Central management system. This system is used to send commands and configurations to the Salt minion that is running on managed systems.

Salt Minions –

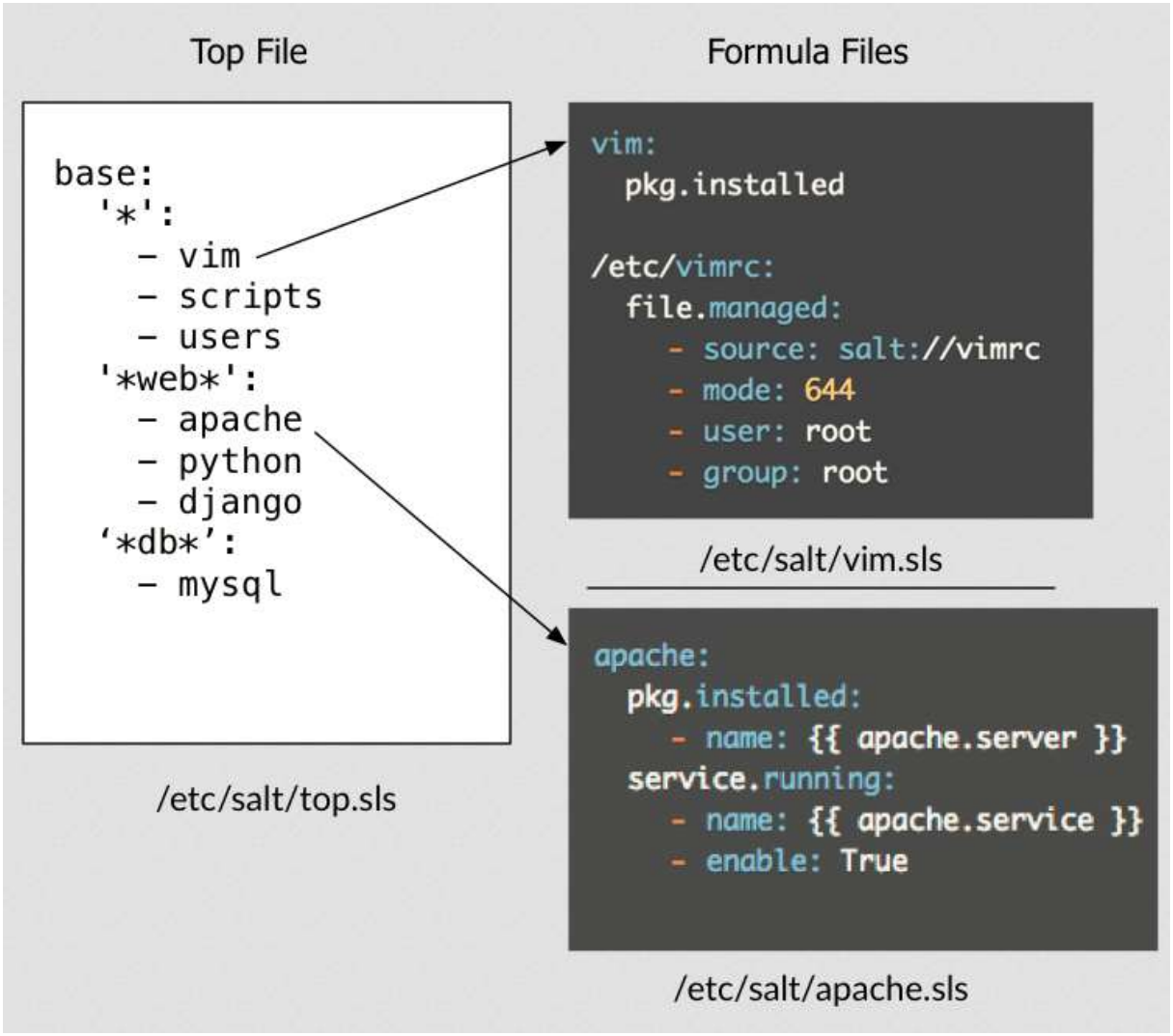
Minion is salt agent run. This system runs the Salt minion which receives commands and configuration from the Salt master.

Grains –

You can compare grains with Chef-ohai, Puppet-Facter.
System variables. Grains are static information about the underlying managed system and include operating system, memory, and many other system properties. You can also define custom grains for any system.

Top File -

Matches formulas and Salt pillar data to Salt minions. This is config file on salt-master.



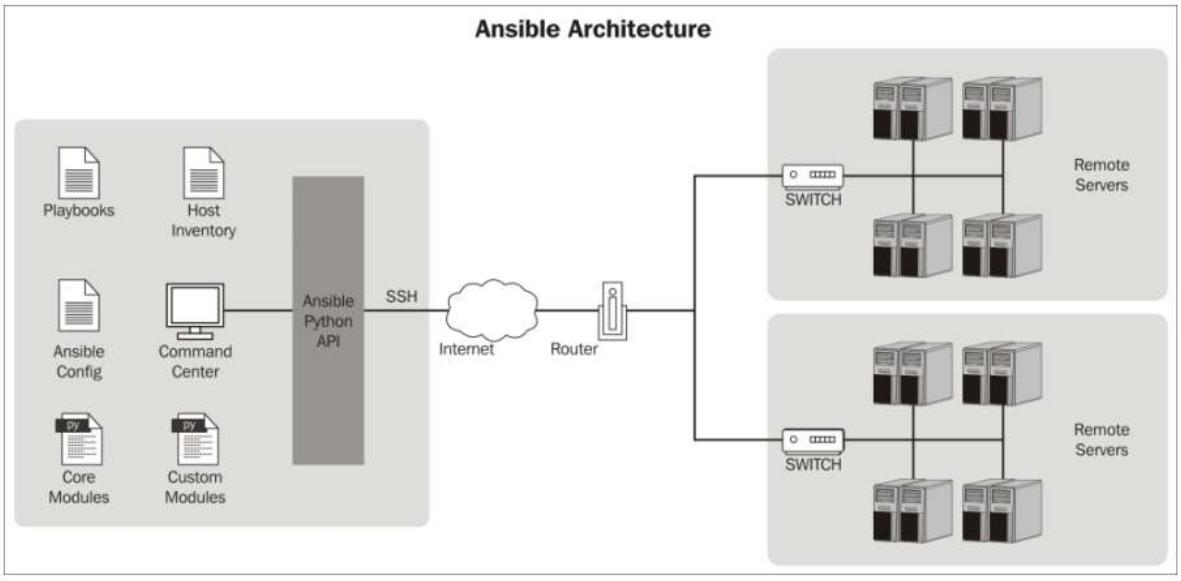
4) Ansible - (https://www.ansible.com **)**

Combines multi-node deployment, ad-hoc task execution, and configuration management in one package. Manages nodes over SSH and requires python (2.4 or later) to be installed on them. Modules work over JSON and standard output and can be written in any language. Uses YAML to express reusable descriptions of systems.

There's also the concept of "**Orchestration**", which is widely used in the industry to mean too many different things, to the point that I think we need new words for it that are less watered-down -- but haven't quite picked them out yet. This can mean any of the following very different things depending on who you are talking to:
basic ability to blast out commands to remote systems
basic ability to blast out resource instructions to a remote system
ability to trigger some configuration management automation to run
some basic ordering of what system should configure before some other system
the ability to build a general-purpose workflow system, such as a pipeline of dependent steps in an IT process

The Ansible architecture -

As you can see from the following diagram, the idea is to have one or more command centers from where you can blast out commands onto remote machines or run a sequenced instruction set via playbooks:



The host inventory file determines the target machines where these **plays** will be executed. The Ansible configuration file can be customized to reflect the settings in your environment. The remote servers should have Python installed along with a library named `simplejson` in case you are using Python Version 2.5 or an earlier version.

What is playbooks in Ansible?

Ansible playbooks are a way to send commands to remote computers in a scripted way. Instead of using Ansible commands individually to remotely configure computers from the command line, you can configure entire complex environments by passing a script to one or more systems.

The playbooks consist of one or more tasks that are expressed either with core modules that come with Ansible or custom modules that you can write for specific situations. The plays are executed sequentially from top to bottom, so there is no explicit order that you have to define. However, you can perform conditional execution on tasks so that they can be **skipped** (an Ansible term) if the conditions are not met. The playbooks are declarative in nature and are written in **YAML Ain't Markup Language** (**YAML**). This takes the declarative simplicity of such systems to a different level. Ref: http://docs.ansible.com/ansible/playbooks.html

Now Let's compare Software Configuration Management Tools -

Points to consider for comparison of software configuration management tools…

1) **Push & pull Model**
2) **Idempotency**
3) **Encrypted/Secure connection (between server and client)**
4) **Orchestration**
5) **Simplicity**
6) **Coding language**
7) **Code re-usability**
8) **Messaging protocol**
9) **Application Automation**

1) In the case of Puppet/Chef vs Ansible:

Between Puppet and Chef, Puppet absolutely requires more system resources than Chef does. A simple comparison of both enterprise version system requirements show that Puppet requires 15gig more disk space and at minimum 1gig more ram for basic requirements. I suspect this is due to the **Puppet multiple third party tools** built into Puppet to achieve its entire functionality.

For example, think about running commands against multiple nodes using Puppet. This requires the tool **MCollective** to be installed. However, with chef you can simple use **knife ssh** to execute commands against designated nodes.

How about node customization based on node role? In this event the tool **Puppet Hiera** is included with Puppet using the YAML configuration language. However, this functionality is built into the core of Chef and how **Chef as built using node attributes**. It requires no additional resources.

Chef also has a better code management tool. While using git is always best practice Puppet relies solely on git for specifying environments and versions. Chef has a built-in developer tool called Knife which gives you a great deal of control over your environment.

Which one to Choose: Puppet, Chef, Ansible and Saltstack?

- ***Puppet, Chef, Ansible and Saltstack present different paths to achieve a common goal of managing large-scale server infrastructure efficiently, with minimal input from developers and sysadmins.*** All four configuration management tools are designed to reduce the complexity of configuring distributed infrastructure resources, enabling speed, and ensuring reliability and compliance. This document explores the mechanism, value propositions and concerns pertaining to each configuration management solution.

- **Puppet** configuration management solution is built with **Ruby** and offers custom Domain Specific Language (DSL) and Embedded Ruby (ERB) templates to create custom Puppet language files, and offers a declarative paradigm programming approach. Puppet uses an agent/master architecture—Agents manage nodes and request relevant info from masters that control configuration info. The agent polls status reports and queries regarding its associated server machine from the master Puppet server, which then communicates its response and required commands using the XML-RPC protocol over HTTPS. This resource describes the architecture in detail. Users can also set up a master-less and de-centralized Puppet setup, as described here.

    - **Puppet** is considered a more operations and sysadmin-oriented solution when compared to Chef, though again—these role-based distinctions are becoming less relevant with each release. DevOps practitioners—both developers and operations staff alike—strive to achieve optimal conditions for continuous integration/delivery. Tooling is therefore increasingly evaluated based on its ability to achieve these ends effectively and efficiently in the context of an enterprise's unique needs.

- **Chef** support both **solo** and **client-server** architecture and offers configuration in a **Ruby DSL** using the imperative programming paradigm. Its flexible cloud infrastructure automation framework allows users to install apps to bare metal VMs and cloud containers. Its architecture is fairly like the Puppet master-agent model and uses a **pull-based approach,** except that an additional logical Chef workstation is required to control configurations from the master to agents. Agents poll the information from master servers that respond via SSH. Several SaaS and hybrid delivery models are available to handle analytics and reporting.

- **Ansible** is latest entrant in the market compared with Puppet, Chef and Salt, Ansible was developed to **simplify complex orchestration and configuration management tasks.** The platform is written in Python and allows users to script commands in YAML as an imperative programming paradigm. **Ansible offers multiple push models to send command modules to nodes via SSH that are executed sequentially**. Ansible doesn't require agents on every system, and modules can reside on any server. A centralized Ansible workstation is commonly used to tunnel commands through multiple Bastion host servers and access machines in a private network.

- **Salt** was designed to enable low-latency and high-speed communication for data collection and remote execution in sysadmin environments. The platform is written in Python and uses

the push model for executing commands via SSH protocol. Salt allows parallel execution of multiple commands encrypted via AES and offers both vertical and horizontal scaling. A single master can manage multiple masters, and the peer interface allows users to control multiple agents (minions) directly from an agent. In addition to the usual queries from minions, downstream events can also trigger actions from the master. The platform supports both master-agent and de-centralized, non-master models. Like Ansible, users can script using YAML templates based on imperative paradigm programming. The built-in remote execution system executes tasks sequentially.

It is not good to compare Chef/Puppet with Saltstack/Ansible. It is better to compare Puppet vs Chef and Ansible vs Saltstack!

1) Puppet offers the following capabilities:

- ✓ Orchestration
- ✓ Automated provisioning
- ✓ Configuration automation
- ✓ Visualization and reporting
- ✓ Code management
- ✓ Node management
- ✓ Role-based access control

Features and Highlights-

Puppet automation works by enforcing the desired state of an environment as defined in Puppet Manifests—files containing pre-defined information (i.e., resources) describing the state of a system. The core components that comprise Puppet are as follows:

- ✓ Puppet Server - the central server that manages Puppet nodes (agents)
- ✓ Puppet Agent - client software installed on managed nodes that enables communication and synchronization with the Puppet master
- ✓ Puppet Enterprise Console - a web GUI for analyzing reports and controlling infrastructure resources
- ✓ PuppetDB - data storage service for Puppet the data produced by Puppet

Other key components worth mentioning over others include **MCollective**, a framework for supporting server orchestration or parallel job execution, and **Hiera**—a hierarchical key/value

lookup utility for providing node-specific configuration data to Puppet (for keeping site-specific data out of manifests). Puppet has integrated MCollective, Hiera, and a myriad of other open source projects into its platform to provide comprehensive automation and management of mission-critical enterprise infrastructures. Many community-contributed add-ons are also available on Puppet Forge—an expansive library of open source modules for extending the platform's features and capabilities.

Pros:

- Strong compliance automation and reporting tools.
- Active community support around development tools and cookbooks.
- Intuitive web UI to take care of many tasks, including reporting and real-time node management.
- Robust, native capability to work with shell-level constructs.
- Initial setup is smooth and supports a variety of OSs.
- Particularly useful, stable and mature solution for large enterprises with adequate DevOps skill resources to manage a heterogeneous infrastructure.
- Model-based approach to enforce your "desired configuration state." I.e. you create manifests to show the server a model of how you want everything to look

Cons:

- Pulling Model by default. For Push need to use **MCollective.** Lacks push system, so no immediate action on changes. The pull process follows a specified schedule for tasks.
- Not the best solution available to scale deployments. The DSL code can grow large and complicated at higher scale.
- Using multiple masters complicates the management process. Remote execution can become challenging.
- **Support is more focused toward Puppet DSL over pure Ruby versions.**
- Lacks adequate error reporting capabilities.
- Better for System Administration automation. However new puppet version allowing developers to create manifests using pure Ruby.

2) Chef offer the following capabilities:

- ✓ Infrastructure automation
- ✓ Cloud automation
- ✓ Automation for DevOps workflow
- ✓ Compliance and security management
- ✓ Automated workflow for Continuous Delivery
- ✓ Chef-Server using RabbitMQ, AMQP protocol.

Chef Features and Highlights:

At the basic level, Chef is a tool for automation, provisioning and configuration management. The platform is made up of the following components:

- ✓ Chef Server - the main hub where Chef propagates and stores system configuration information and policies (i.e., recipes and cookbooks). The Chef management console is the web user interface for Chef Server.

- ✓ Chef Client - installed on every node being managed, the Chef Client performs configuration tasks on the local machine.

- ✓ Workstation - allows designated workstations to author/test/maintain cookbooks and upload them to Chef Server. Workstations are also used when utilizing the Chef development kit package.

- ✓ Chef Analytics - a platform that provides actions and run history, real-time reporting, and notifications around Chef automation activities.

- ✓ Chef Supermarket - an open source directory of community-contributed cookbooks

Pros:

- **A Chef for All Seasons**
- **One of the most flexible solutions for OS and Application middleware management.**
- **Designed for programmers.**
- Strong documentation, support and contributions from an active community.
- Very stable, reliable and mature, especially for large-scale deployments in both public and private environments.
- Chef offers hybrid and SaaS solutions for Chef server, analytics and reporting.
- **Sequential execution order.**
- **Chef-solo is very useful and can be easily used in any other remote execution platform. Chef-Solo is very scalable and can be ran on remote servers using SSH.**
- **Node attributes are part of Chef only** (Puppet Hiera)

Cons:

- Requires a steep learning curve.
- Initial setup is complicated.
- Lacks **push**, so no immediate action on changes. The pull process follows a specified schedule. **However, using Knife+SSH we can also push changes to servers. Other option is MCollective.**

3) Ansible offer the following capabilities:

- ✓ Streamlined provisioning
- ✓ Configuration management
- ✓ App deployment
- ✓ Automated workflow for Continuous Delivery
- ✓ Security and Compliance policy integration into automated processes
- ✓ Simplified orchestration

Features and Highlights-
Ansible is much more spiritually like Salt than to either Puppet or Chef. The focus of Ansible is to be streamlined and fast, and to require no agent installation and no dependencies on the machine to be managed, except for a Python interpreter.

Pros:

- ➢ **Push Model: Easy remote execution, and low barrier to entry.**
- ➢ Shares facts between multiple servers, so they can query each other.
- ➢ Powerful orchestration engine. Strong focus on areas where others lack, such as zero-downtime rolling updates to multi-tier applications across the cloud.
- ➢ Syntax and workflow is easy to learn for new users.
- ➢ Sequential execution order.
- ➢ **Supports both push and pull models.**
- ➢ Lack of master eliminates failure points and performance issues. Agent-less deployment and communication is faster than the master-agent model.
- ➢ High security with SSH.

Cons:

- ➢ **Increased focus on orchestration over configuration management.**
- ➢ **SSH communication slows down in scaled environments.**
- ➢ **Requires root SSH access and Python interpreter installed on machines, although agents are not required.**
- ➢ The syntax across scripting components such as playbooks and templates can vary.
- ➢ Underdeveloped GUI with limited features.
- ➢ The platform is new and not entirely mature as compared to Puppet and Chef.

4) SaltStack capabilities and use cases include:

- ✓ Orchestration and automation for CloudOps
- ✓ Automation for ITOps

- ✓ Continuous code integration and deployment
- ✓ Application monitoring and auto-healing
- ✓ DevOps toolchain workflow automation with support for Puppet, Chef, Docker, Jenkins, Git, etc.

Features and Highlights-

Salt, like Ansible, is developed in Python. It was also developed in response to dissatisfaction with the Puppet/ Chef hegemony, especially their slow speed of deployment and restricting users to Ruby. Salt is sort of halfway between Puppet and Ansible – it supports Python, but also forces users to write all CLI commands in either Python, or the custom DSL called PyDSL. It uses a master server and deployed agents called minions to control and communicate with the target servers, but this is implemented using the **ZeroMq** messaging lib at the transport layer, which makes it a few orders of magnitude faster than Puppet/ Chef. Salt's biggest advantage is its scalability and resiliency. You can have multiple levels of masters in a tiered arrangement that both distributes load and increases redundancy. Salt also uses YAML config files, organized into templates or packages called states.

Now the bad news. ***Some still feel that the master-minion setup is less secure than the pure SSH in Ansible. ZeroMq doesn't offer native encryption***, so Salt has to include an extra layer of Kevlar by way of its own AES implementation. And Salt uses persistent daemons for master-minion communication between nodes, which introduces its own (admittedly small) performance hit – though this setup excels in large deployments.

Pros:

- ➢ Effective for high scalability and resilient environments.
- ➢ Easy and straightforward usage past the initial installation and setup.
- ➢ Strong introspection.
- ➢ Feature-rich and consistent YAML syntax across all scripting tasks. Python offers a low learning curve for developers.

Cons:

- ➢ Installation process may not be smooth for new users.
- ➢ Web UI offers limited capabilities and features.
- ➢ **Not the best option for OSs other than Linux.**
- ➢ The platform is new and not entirely mature as compared to Puppet and Chef.

Configuration management: push versus pull based topology -

The more established configuration management (CM) systems like Puppet and Chef use a **pull-based approach:** clients poll a centralized master periodically for updates.
Some of them like Ansible and Saltstack offer **push-based** approach.

In a 'pull' system, clients contact the server independently of each other, so the system as a whole is more scalable than a 'push' system

Using things like **MCollective**, you can convert Puppet and Chef into a **push based system.** Generally, it's trivial to convert a pull system to a push based one, but not always simple to go the other way.

Puppet	Chef
1) Best for System Configuration automation 2) Message Queue: ActiveMQ and STOMP Protocol 3) No master-less or standalone agent like chef-solo 4) Focus on Puppet-DSL 5) Node attributes using external Hiera **6) There is no implied Sequential(ordering) in puppet manifests.** 7) Puppet manifests are **declarative** 8) No Code re-usability or difficult. Also, custom syntax 9) Puppet-Factor: key=value	1) Best for Infrastructure and Application Automation 2) Message Queue: RabbitMQ and AMQP Protocol 3) Master-less and standalone Chef-solo is very useful. 4) Ruby-DSL builtin adding advantage to Chef. 5) Node attributes are buitin 6) **Sequential execution order** (cookbook:recipe) 7) Chef recipe **imperative** 8) Code re-usability due to Ruby. Also, native ruby syntax. 9) Chef-Ohai: nested Hash 10)
1) Ruby Based 2) Pull based ~ agent will pull changes from puppet master 3) Push based using MCollective 4)	1) Ruby based 2) Pull based ~ agent will pull changes from Chef-server 3) Push based using MCollective

	Pros	Cons
Ansible	1) Agent-less deployment and communication (NOTE: Ansible is using SSH to connect to remote server, mean ssh is agent for ansible!) 2) CLI supports almost any programming language 3) Uses Python, which is omnipresent in Linux distros 4) Excellent security using SSH / SSH2	1) Prone to performance issues at times 2) Introspection (i.e., seeing Playbook variable values) is lacking 3) idempotency is poor and playbook need to have logic and idempotency implemented.

	5) Additional Tower dashboard allows for visual management of nodes/resources (available in commercial version)	
Salt	1) Quickly scalable, very resilient and efficient because of multi-master capability 2) Use of minions offers more options and flexibility than Ansible 3) No GUI yet (currently under development)	1) Forces users to learn Python or PyDSL 2) Underdeveloped GUI 3) Minions not as efficient as agent-less communication for small-scale deployment

Thank you,
Arun Bagul
IndianGNU.org
Email: arunbagul@gmail.com Or
arunbagul@indiangnu.org

Date: Oct, 2016
Version: v3