LINCHPIN: A YAML TEMPLATE BASED CROSS CLOUD RESOURCE PROVISIONING TOOL

by

SAMVARAN KASHYAP RALLABANDI

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON

DECEMBER 2016

Acknowledgements

I would like to extend my heartfelt thanks to Mr. David Levine, my supervisor who inspired and motivated me to do this thesis. This thesis would not have been possible without his encouragement. I extend my sincere gratitude to Mr. David Levine for his invaluable support and supervision.

My heartfelt thanks to my committee members Dr. John Robb, and Dr. Farhad Kamangar, and the computer science department, UTA for their invaluable cooperation, and support. Finally, I would like to thank all my friends who helped me through this journey.

October 26, 2016

Abstract

LINCHPIN: A YAML TEMPLATE BASED CROSS CLOUD RESOURCE PROVISIONING TOOL

Samvaran Kashyap Rallabandi, MS

The University of Texas at Arlington, 2016

Supervising Professor: David Levine

A cloud application developed, will have a specific requirement of particular cloud resources and software stack to be deployed to make it run. Resource templates enable the environment design and deployment required for an application. A template describes the infrastructure of the cloud application in a text file which includes servers, floating/public IP, storage volumes, etc. This approach is termed “Infrastructure as a code.” In Amazon public cloud, OpenStack private cloud, Google cloud these templates are called as cloud formation templates, HOT (Heat orchestration templates), Google cloud templates respectively. Though the existing template systems give a flexibility for the end user to define multiple resources, they are limited to the provision in single cloud provider with a unique set of cloud credentials at a time. Due to this reason, vendor lock-in arises for the service consumer.

The current thesis addresses the vendor lock-in problem by proposing a framework design and implementation of provisioning of the resources in the cross-cloud environments with YAML templates known as “Linchpin.” Linchpin also follows “Infrastructure as code” approach, where the full requirements of the users are manifested into a predefined YAML structure, which is parsed by underlying configuration and deployment tool known as Ansible to delegate the provisioning to the cloud APIs. Current framework not only solves the vendor lock-in issue also enable the user to do cross-cloud deployments of the application. In this thesis, a comparative study of the existing template-based orchestration frameworks with Linchpin on the provisioning time of the virtual machines. Further, it also Illustrates a novel way to generate Ansible based inventory files for post provisioning activities such as the installation of software and configuring them.

Table of Contents

List of Illustrations

List of Tables

Chapter 1

Introduction

An application which utilizes on-demand cloud services will have specific requirements concerning its deployment. Regardless of the cloud provider, each service provisions a set of resources at the application's disposal. Resources allocated to the application can be grouped into multiple resource groups based on its requirement. Each cloud vendor has devised their component to define the set of resources and resource groups as stacks with the help of template files. For example, OpenStack [1] and Amazon web services [2] based clouds use HOT (Heat orchestration templates) [3] and Cloud Formation templates [4] to orchestrate resources on their respective clouds. The resource groups created by initiating provisioning of templates are termed stack. Usage of these templates makes the buildup and teardown of the cloud resources easier when compared to writing individual scripts requesting service provider API to cater the resources for applications. Moreover, Existing templates also enable configuration of the resources using cloud-init [5] scripts. Thus, it resulted in the rise in popularity among the developers and cloud administrators for On-demand template-based provisioning of application/project requirements at their will.

Each cloud vendor has an intrinsic way to handle the provision/teardown resources involving a plethora of parameters which makes it difficult for any developer to switch from one cloud provider to another and provision resources across multiple cloud providers. Though the efforts have been made in the past to come up with a unified taxonomy of cloud frameworks [6], they have not been adapted lately for provisioning systems.

In the current scenario of templates, resource provisioning has some limitations as follows

1. Unified template for provisioning of multi-cloud resources.
2. Provisioning multi-cloud resources.
3. Post-provisioning configurations of resources.

Though every framework has its way of overcoming the limitations, they inherently lack the standard features needed for the multi-cloud deployments.

Chapter 2

Motivation, Goals and Organization

2.1 Motivation:

Provisioning of multi-cloud resources and configuring them has always been tedious tasks for IT administrators. With Organizations based on the cloud using multiple cloud accounts to manage frameworks such as AWS [2], Azure [8], and Google Cloud [9] it has been difficult to allocate and deallocate the resources in multi-cloud and cross-cloud environments. Though the orchestration frameworks such as OpenStack Heat [9], and Amazon Cloud Formation [4] have been able to resolve the coordination among the cloud resources, they are limited. Due to this, a higher learning curve arose to manage multiple cloud accounts.

Having requirements laid out in the form of cloud template files, helps users to isolate the resource provisioning of the cloud application development. Further, DevOps tools such as Puppet [10], Chef [11], Ansible [12] have enabled dynamic configuration and deployment of environments. However, these tools can also be able to provision cloud resources with the help of API based plugins for managing the complete life cycle of provisioning, i.e., deployment, and teardown of the resources. The current motivation underlies in having a DevOps tool to orchestrate multi-cloud resources with a single cloud template mimicking the orchestration frameworks. This not only provides efficient deployment and teardown of development/production environments at ease but also gives a reliable, flexible solution to orchestrate services across multiple accounts.

2.2 Goals:

The primary objective of the thesis is to propose and implement a modular, syntax particular framework dependent on a DevOps tool enabling orchestration of multi-cloud resources using infrastructure defined in a template file. It also demonstrates the multi-cloud management of the resources by imitating the functionality of existing orchestration tools.

Further, it investigates the proof of concept having a cross-cloud deployment [13] of the environments using the same framework. Supplementing the proposed framework's performance is compared against the existing orchestration frameworks such as OpenStack HEAT [9], Amazon cloud formation [4], and Google cloud templates [14].

2.3 Organization of Thesis

This thesis starts with an introduction to the thesis in chapter 1. Chapter 2 discusses motivation and goals and organization of the thesis. Chapter 3 introduces types of cloud computing services. Chapter 4 details comparison between virtual machines and containers. Chapter 5 provides definitions of Continuous Integration and Continuous Delivery Processes. Chapter 6 discusses the significance of unified taxonomy of cloud providers. Chapter 7 discusses the Internet provided by cloud providers. Chapter 8 describes cloud authentication models of Amazon and Google cloud. Chapter 9 gives detailed examples of multiple cross-cloud deployment use cases. Chapter 10, 11, 12 focus on defining configuration management tools, Infrastructure as code approach and benefits of the multi-cloud strategy. Chapter 13, 14 gives a brief introduction to configuration tools and orchestration tools respectively. Chapter 15 gives an introduction to proposed solution Linchpin. Chapter 16 describes a framework, features, and workflows of Linchpin processes. Chapter 17 discusses the experiments made and performance of Linchpin. The thesis concludes with Chapter 18 with future work outlined in Chapter 19.

Chapter 3

Cloud Computing services

There can be many services provided by cloud vendors. Broadly, cloud services can be classified into three types:

1. Infrastructure as service [15]
2. Platform as a service [15]
3. Software as a service [15]

Each type of service has its level of flexibility and scalability in the usage of service specific resources.
For example, in the case of Infrastructure as the service type of cloud services such as AWS EC2 [16] and Google Compute Engine [17], and Rackspace Servers [18], end-user will be provided with remote access to hosted infrastructure. Users will have complete control of the infrastructure as an endpoint accessed through public IP address. Though it gives an excellent control, there would be a lot of manual intervention of clients to maintain their operating systems. The scalability costs involved in deployment and maintenance of application on the IaaS model depend on end-user awareness of the configuring services to communicate with each other. Thus, creating a burden of hiring more competent admins to maintain cloud-based services resulting in a more expensive solution.

On the other hand, Platform as service type cloud services such as AWS Elastic Beanstalk [19], Google App Engine [20], and Red Hat Openshift [21] abstracts the infrastructure layer to the end user and gives the user an interface to run their developed code and run it on a scalable app. In PaaS model, though users have the option to scale up the instances of applications running over a period. They would not have access to the underlying infrastructure neither remotely or physically.

When coming to Software as Service model, targeted end users are neither using a hosted infrastructure nor a platform to run their application code. Users of SaaS model are unsophisticated users who do not care about hosting and maintenance of the server, rather, they would be making use of software in the cloud with a flashy user interface which requires minimal knowledge. Dropbox [22], Box [23], Gmail [24], Office 365 [25] are good examples for SaaS model based services.

Having many services among SaaS, PaaS, IaaS to manage, there is no unified solution to manage all of them services through a single endpoint. Organization using these services face difficulties due to costs involved in them. Current thesis tries to address the problems in areas of IaaS services.

Chapter 4

Virtual Machines vs. Containers

Containers gives us a packaged runtime environment of an application including its dependencies, libraries by abstracting the underlying operating systems. According to VMware, a virtual machine is a software computer that, like a physical computer, runs an operating system and applications [26]. The virtual machine comprises a set of specification and configuration files backed by the physical resources of a host. Every virtual machine has dedicated resources allocated to a host machine whereas container’s resources are shared among the other processes. Docker is one of the best example for container-based solutions and VirtualBox [27], Xen [28] and HyperV [29] are products which are virtual machine providing solutions.

Due to batteries included approach and light weight nature of containers, they gained high popularity recently [30]. Further containers are proved to be having advantages on the faster deployments and portability. However, when compared to virtual machines, containers are still way behind in certain use cases.

Virtual machines and containers are inherently different when it comes to comparison. Various technological differences can be listed out between containers and virtual machines.

4.1 Technical Differences:

Hypervisor dependency: Virtual machines are spawned with a specific format tied to its hypervisor [31] making it difficult to port them from one hypervisor to another. In contrast, containers are hypervisor independent and need not be converted.

Backend support: Mimicking standalone machine, virtual machines can be reused and modified at a higher rate than containers. Thus, making it suitable for long living applications. On the other hand, containers are suitable for short lived lightweight stacks. Any environment which tends to change frequently, containers would not be a good deployment approach when compared to virtual machines. Unlike containers, virtual machines as they don't go stale and updates can be triggered easily with heavy software stack.

Size: Image size of a virtual machine varies from 2Gb to 30 Gb due to the primary operating system overhead, whereas the containers are in size of few megabytes which insignificant in comparison with virtual machines, However, the containers approach is suitable for only the application stacks which are sized to be minimalistic in nature. When we try to install heavy application frameworks such as OpenStack and make a container out of it, the containers tend to fail due to lack of storage support and be heavier losing their portable nature.

Speed: Virtual machines usually have the dedicated virtual space and memory in the hypervisor host. At any point of time, a virtual machine booted is reserved for space and memory allocated to it unless hypervisor is over committed. If the dedicated resources are available for virtual machines, they perform better than containers. However, in best practices, containers are running on a virtual machine for efficient isolation and utilization of resources.

Host OS: Containers are limited to Linux distributions whereas, in virtual machines we can select from a wide variety of distributions. With the limited support of software stack, it confines the user to a finite set of environments.

Networking: As a standalone solution, the virtual machine also supports drivers for creating software defined networks among themselves.Though, there are some limited network configuration options, containers tend to work mostly on a NAT configuration of the host server network [45]. Another advantage of virtual machines is one can configure their virtual machine to work with external network hardware, which is better option to have when we have an application which tends to change their requirements over a period.

Snapshots: Unlike virtual machines, containers facilitate the creation of snapshots of images on to a repository. However, they lack in support for retaining the state of memory because of the shared resources on the host machine. Container implementations such as Docker have additional support to version control the whole application like Git which cannot be a part of virtual machines due to high volatile nature and massive memory footprint of virtual machines.

Management: As the application are more prone to changes the administration of the virtual machines based applications is much easier than containers based ones which need to be replaced in order not to go stale. For example, a production setup running on a bunch of containers needs to be replaced, deployed and tested again periodically for every update. Whereas in the case of virtual machines, one can SSH into them and refresh the code for the projects.

Security: A user inside a container will have a root access to the file system inside the container. The further container also has access to system namespaces like process, network, hostname. According to Steven J from ITworld [31], this is a severe security concern because having access to the system level can be breached at any point in time being a super user inside a container.

According to Philosophy of containers, one can say that containers are built for developers who work on simple stack deployments, whereas virtual machines are designed for IT admins who work with complex enterprise level deployments. In addition to that, containers are processed/stack focused solution.

Though Docker and LXC containers are gaining a good stance in application deployment world, when it comes to large enterprise level configuration, containers lack their features in complex management and support for multiple distributions. However, solutions like running Docker containers on virtual machines and Docker containers on a bare metal still wins the single stack deployments with optimal performance making virtual machine solution more sustainable. Thus virtual machines are still prominent despite the container app revolution [33].

Chapter 5

Continuous Integration and Continuous Delivery

With a very high competition among the software solution providers, time to market has always been a crucial factor in term of product development. Many companies have been trying to reduce the time to market, by introducing methodologies in the Agile lifecycle. Among the recent methods, Continuous Integration and Continuous Delivery are prominent.

According to TechTarget [34],

“Continuous integration (CI) is a software engineering practice in which isolated changes are immediately tested and reported on when they are added to a larger code base.”

The primary objective of continuous integration is to provide feedback oriented defect detection and fixing in the software code.

Paul Duvall, co-author of *Continuous Integration: Improving Software Quality and Reducing Risk* [36], quotes the best practices of CI as follows:

- Committing code frequently.
- Categorizing developer tests.
- Using a dedicated integration build machine.
- Using continuous feedback mechanisms.
- Staging builds.

Continuous Delivery [36] is usually the following step after Continuous Integration; once the software is developed and integrated, the build will be deployed on to production phase. Tools such as Jenkins enable the continuous integration of code.

Continuous delivery (CD) [36] is a software engineering approach in which teams produce software in short cycles, ensuring that the software is reliable to be released at any time. Post completion of continuous integration, software code is being processed through continuous delivery practices resulting in a software release.

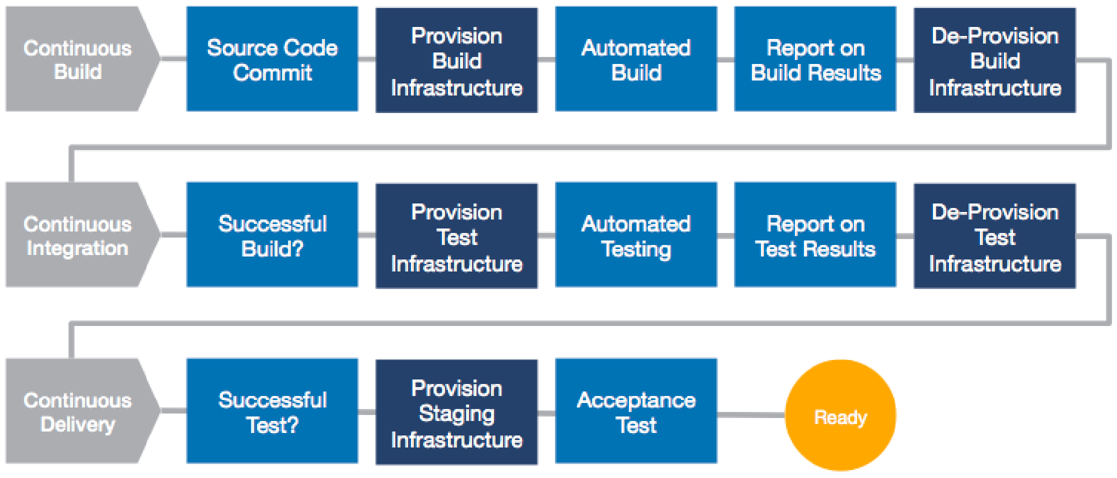


Figure 5-1 Continuous Build, Continuous Integration and Continuous Delivery process at Right scale [37]

In the Figure 5-1, continuous delivery is a three-stage process.

Stage1: Continuous build: Continuous build is often a misnomer to continuous integration. Continuous build is a process in which the source code is being checked out regularly from the repository onto a build infrastructure and provisioned. Later results are reported on a successful build.

Stage2: Continuous Integration: Followed by a successful build from the continuous build The build is deployed onto newly provisioned test infrastructure. Later, provisioned builds run the integration tests and the provisioned infrastructure is torn down.

Stage3: Continuous Delivery: During the delivery process, the software code ran against acceptance tests in the staging environment. If the results are promising, then the software is released.

The above methodology is a proven optimal method to achieve robust software with remote teams working on multiple stages of software development for faster delivery. In real time, the whole infrastructure of the continuous integration and deployment process is provisioned on a private or public cloud providers. During every stage of the project, we can also observe a significant number of instances are being provisioned and tore down. Due to this, each organization will develop its mechanisms to provision and configure the environments on the cloud providers. Having resources provisioned on a multi-cloud environment helps the organizations to cut down the costs for the infrastructure [44] in each phase of continuous integration and delivery.

Chapter 6

Unified Taxonomy of IaaS

With the advent of cost-effective solutions like pay per use and on demand scalability, enterprise IT infrastructure has been migrating their traditional resources towards cloud resources significantly. Due to this, a high competition arose among the vendors such as AWS, Microsoft Azure, and Google Cloud to allure their customers with lucrative cloud service offerings. Though services offered by multiple cloud providers are similar with subtle differences, there isn't any common vocabulary or standard to define a cloud service types and their parameters.

For example:

AWS, Microsoft Azure and Google Cloud offers compute services named as AWS EC2, Microsoft Azure Servers and Google Compute Engine. Each service has a different set of parameters to be passed to boot a virtual machine in the cloud.

Excluding authentication parameters, the following are the parameters to boot instances in different cloud platforms respectively:

common_term	Amazon EC2	Google Compute	Microsoft Azure
Instance name	-	Name	service_name
image_id	ami_id eg: ami_16238	Image	image_name
region_name	region_name	Zone	Location
security_group_name	security_group_name	firewall_rules (optional)	network_config
size_of_instance	instance_type	image_type	role_size
key_pair	key_pair	security_key (optional)	-

Table 6-1 Common terms among cloud providers

In above example, through the parameters we could observe that they convey the same meaning even though they differ in terminology. For example, the region in which virtual machines are provisioned also known as region in AWS EC2, zone in Google cloud and location in Azure.

In an end user perspective, all cloud providers together can be envisioned as a black box. Since each cloud provider has a similar services and REST APIs to interact and each service offered by the cloud vendor, has a core set of parameters needs to be passed in the request. One can provide a single library to interact with all the providers. Apache libcloud [38] and Daisen cloud [39] are some of the abstraction libraries to achieve the same.

Efforts have been made in the past to come up with common ontology and unified taxonomy of cloud frameworks [6]. Despite many proposals, there has not been any implementation or effort to enforce vocabulary. Current tool Linchpin, acts as a proof of concept and framework which adds a higher level of abstraction to common cloud services and orchestrate them seamlessly.

Chapter 7

Internet of cloud providers

The Internet is always limited to the bandwidth allocated to an organization or a single user. For example, for any organization, the Internet bandwidth is restricted by the lines provided by the ISP's. In a cloud-driven infrastructure, the Internet speeds among the machines between the regions are taken care by the cloud providers. Cloud providers such as Amazon and Google, have their infrastructure on the best lease lines with very less network outage and high Internet speed acting like a point to point connections.

7.1 Advantages of outsourcing the Internet to the cloud:

The following are advantages of the outsourcing Internet to cloud provider:

Less network outage:

According to cloud harmony [40], by comparison, Microsoft Azure and Google Cloud Platform each had more than five-fold the downtime when compared to year 2015. Azure experienced 71 outages totaling 10 hours and 49 minutes in services tracked by CloudHarmony, while 167 outages across 11 hours and 34 minutes were recorded in Google's cloud. Hence high bandwidth applications on multiple clouds will reduce the network outage.

Cloud providers tend to give higher bandwidth to the machines allocated. According to AWS docs, AWS EC2 gives following bandwidths to the end user

Output:

```
[ ID] Interval       Transfer        Bandwidth
[  5]  0.0-60.0 sec  6.71 GBytes   960 Mbits/sec
[  4]  0.0-60.0 sec  6.69 GBytes   957 Mbits/sec
[  6]  0.0-60.0 sec  6.55 GBytes   937 Mbits/sec
[  7]  0.0-60.0 sec  6.84 GBytes   980 Mbits/sec
[  8]  0.0-60.0 sec  6.68 GBytes   956 Mbits/sec
[  9]  0.0-60.0 sec  6.76 GBytes   968 Mbits/sec
[ 10]  0.0-60.0 sec  6.55 GBytes   938 Mbits/sec
[ 12]  0.0-60.0 sec  6.77 GBytes   969 Mbits/sec
[ 11]  0.0-60.0 sec  6.70 GBytes   960 Mbits/sec
[ 13]  0.0-60.0 sec  6.80 GBytes   973 Mbits/sec
[SUM]  0.0-60.0 sec  67.0 GBytes  9.60 Gbits/sec
```

The above observations show an average bandwidth of 959.8 Mbits/sec is observed which is way more the average bandwidth of corporate. Thus, having a minimal Internet to manage the instances by outsourcing the Internet would actually minimize the expenses of the end user.

Network Monitoring:

Networking monitoring is easy when the machines are within a cloud environment. Cloud providers also provide on demand and monitoring of networking traffic using a service level tools which also alert users unpredicted or suspicious traffic. One can easily detect the malicious traffic and prevent them using the security groups and firewalls provided by the cloud vendor. Amazon CloudWatch [40] acts as a best example for monitoring service for AWS cloud resources which are running on AWS. It is also used to alert the user when the AWS usage metrics go beyond a certain threshold. Due to the multi-region deployments of the instances, the applications hosted by the cloud providers are less prone to failure than applications hosted on the site.

With all above mentioned advantages one can say that moving an application to cloud helps the administrators not only to efficiently manage resources by monitoring the external factors which are responsible for the failure but also avoid the downtime of the applications hosted on the cloud.

Chapter 8

Cloud authentication models

Authentication is a crucial factor in determining the user's accessibility of the cloud resources. Cloud providers enforce various cloud authentication models, for denial or allowance of services to users. Amazon uses IAM [43] for Authentication, whereas Google uses users and service accounts [46] to determine the authenticity.
This chapter discusses the both authentication models of Amazon and Google Identity Access Manager:

8.1 Amazon IAM:

Amazon provides identity access management portal, where an admin user can create policies and users. Each policy defines set of access permissions to users and group to allow or deny usage of the resources.

Amazon IAM has the following features:

1. Shared access to AWS Account: Access to a given Amazon account can grant and revoke without sharing the passwords.
2. Granular permissions: Resource level, operational level permissions can be set for AWS resources seamlessly.
3. Secure access to AWS resources: Services such as Amazon RDS and DynamoDB can be shared across EC2 instances securely.
4. Identity Federation: Provides temporary or consistent access to AWS account based on third party providers.
5. Identity Audit: Using services such as AWS Cloud Trail one can also verify the requests among the IAM identities.
6. Eventually Consistency: Changes made to account from one region, though they are not available immediately, they will be eventually consistent.

In addition to the above features AWS provides following methods of authentication with the cloud resources:

1. Password based: Simple email and password are used to login to as console which acts a single user end point for all the cloud services.
2. AWS MFA (Multi-Factor Authentication): In addition to email and password, a temporary token is being generated for external AWS MFA device to get into AWS console.
3. Access Keys and Secret ID: Access keys and secret keys are one method to manage the AWS services programmatically using AWS SDKs like boto and Apache libcloud
4. CloudFront key pairs: AWS offers a shared key pairs storage to access AWS EC2 instances.
5. X.509 certificates: Used to make secure SOAP-protocol requests to some AWS services.

8.2 Google Cloud IAM:

Google also provides authentication mechanisms similar to AWS IAM [40]. The admin console is independent of the project the user is operating on. Unlike AWS, Google Cloud has few Auth mechanisms. Which are as follows:

1. Service accounts:

   A service account acts as an application routing request to APIs on behalf requestor. Using role-based service accounts. The role can assume by the member and each role authority of managing all the operations of a specific service. Each service account associated with a role and private key acts as a credential to access the service.

2. OAuth Id and secret:

   Provides OAuth credentials, i.e., client id and secret key for a managed service. These types of credentials are used prominently in web based applications

3. API Key:

   Generated API key to the services which support authentication for specific services of Google cloud which is passed by the application when consuming the REST services Google cloud.

Google IAM and Amazon IAM provides rich features for authenticating and authorizing the users. When it comes to usage of this authentication by the application service accounts and AccessID, secret key are used to manage the cloud services in AWS and Google Cloud respectively. The credentials can be referred through a file. To maintain them, credentials are ideally stored in encrypted files or a database for security. Organizations having multiple accounts face difficulty to maintain them. In this thesis, a credential store is implemented for easier management of credentials of cloud providers by abstracting all the credentials into one place.

Chapter 9

Cross-cloud deployments: use cases

A deployment in which the application services are spread across the multiple cloud providers and interconnected with each other using configuration script is known as cross-cloud deployment [47]. Most of the cloud deployments are homogenous in nature, i.e., a cloud-based application is hosted using a single cloud provider services. In the event of cross Cloud deployment, the services of a deployed applications are not bound to single provider rather are distributed across the cloud providers based on service level costs or performance. Given a software or web-based application having its components spread over multiple cloud providers has advantages [49] as follows:

Advantages:

1. A wide variety of cloud services to choose
2. Avoid vendor lock-in
3. Cutting down cloud infrastructure costs
4. High availability

The following are multiple use cases where the user can get benefitted from across cloud deployments:

Cross-cloud deployment use cases:

9.1 Scenario 1: Hadoop deployments: [49]

Apache Hadoop is a framework open sourced for data processing on distributed data storages and Hadoop clusters built by using low-cost commodity hardware. With the advent of compute service, enterprises started moving their deployments from an on-premise infrastructure to cloud-based infrastructure to avoid the expenses on hardware maintanence. Virtually unlimited scaling up of Hadoop instances is another advantage.

The architecture of Hadoop is a master-slave architecture as depicted below:

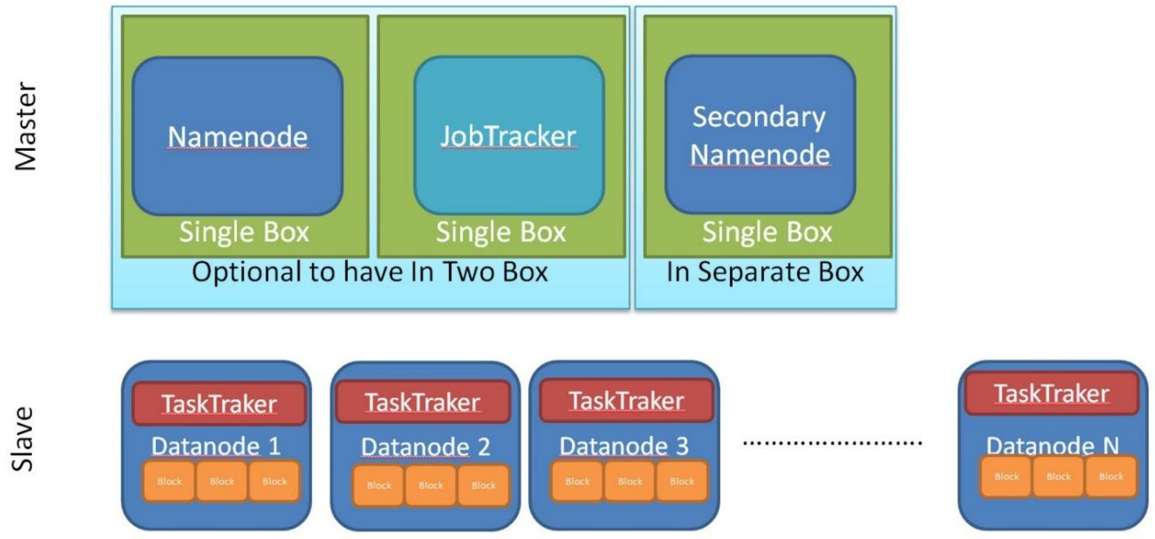


Figure 9-1 Hadoop Master-slave architecture [49]

Master node of the Hadoop architecture consists of mainly Name Node and Job Tracker.

Name node is responsible for files and namespace management which include the opening, closing deletion of files and directories on the Data nodes along with access permissions.

Job Tracker is an endpoint for users to communicate with the map reduce jobs submitted to the Hadoop. Job tracker is also responsible for managing distributed queue of tasks and assignment of tasks to task tracker.

Slaves are the data nodes formatted in Hadoop file system. Slaves accept the data and store it across themselves. There can be any number of slaves in a Hadoop master-slave architecture. Each of the slaves has a task tracker.

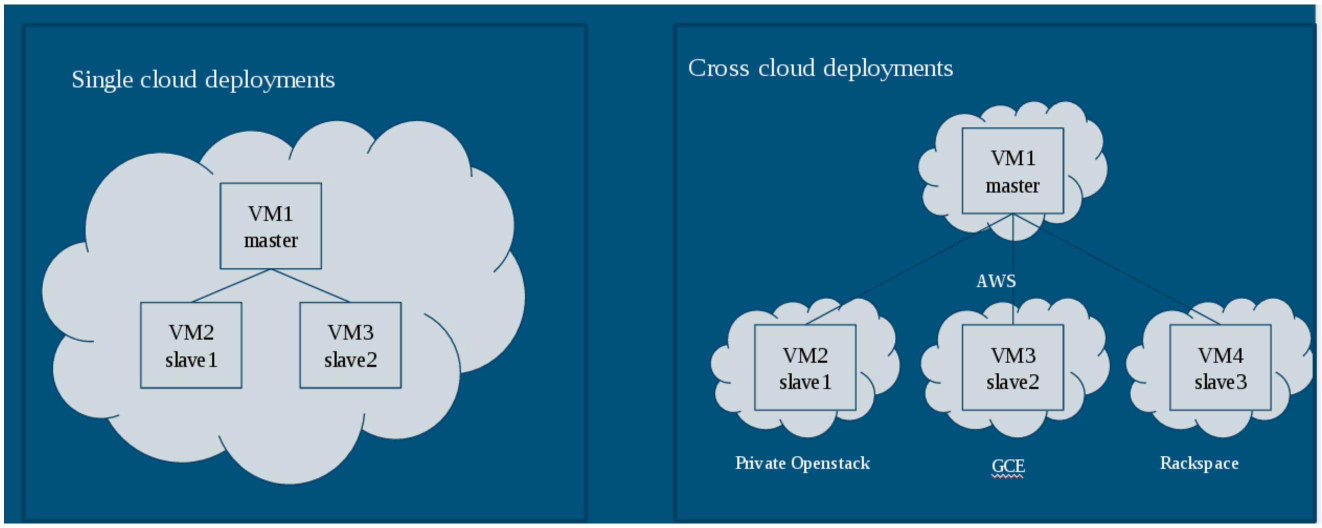


Figure 9-2 Hadoop master-slave single cloud and cross-cloud deployments

The entire deployment of a Hadoop cluster can be done within a single cloud provider compute instances. Consider a scenario where an end user has chosen cloud provider’s (for, e.g., Google cloud) compute service for the deployment of a Hadoop cluster. Ideally, it consists of a master node and more than two slave data nodes. Master nodes does all the management and monitoring of data processing tasks, whereas the data node is for data processing and storage. One might want a high-performance storage optimized virtual machines for data node and less configuration virtual machine for a master node. Having deployed the whole cluster on a cloud provider, one usually does not make use of the compute service furnished by another provider even if the another cloud provider is providing a high-performance, cost-effective virtual machines due to the difficulty in provisioning and configuring the other instance. This kind of scenarios raise a vendor-lock-in where the user is forced to use the only one cloud provider. With the help of cross-cloud deployments, one can distribute loads amongst different clouds makes the Hadoop clusters less prone to the failure and high availability, because the probability of multiple cloud providers going down is less when compared to single cloud failure. Thus, making it more resilient and profitable at the same time.

9.2 Scenario 2: Web Server deployments:

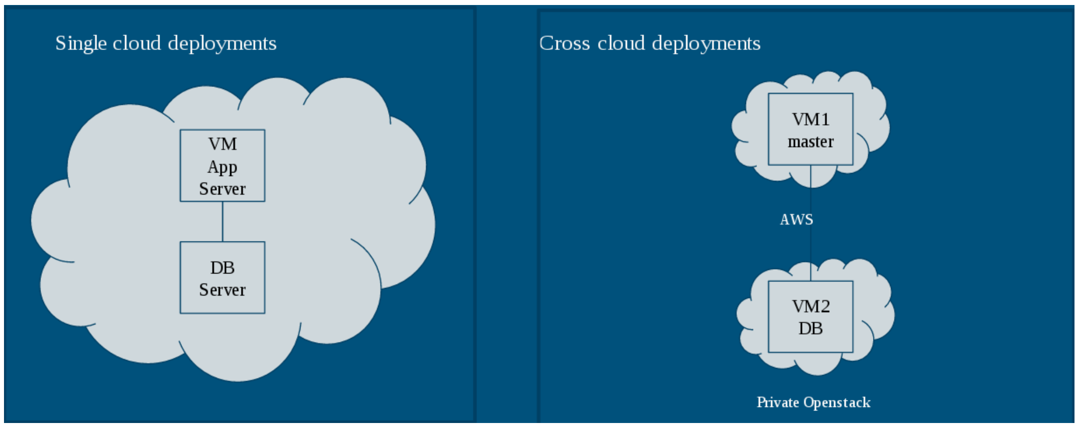


Figure 9-3 2 tier Web application single and cross-cloud deployments

Web applications are the most common use case of deployments in a cross-cloud environment. Users always would like to spread their resources in public and private cloud. A simple two tier web application with an application and database server is deployed on a cloud provider. An enterprise level application working confidential data would like to maintain their data on a private cloud provider such as Openstack and host the application server accessing the data on a public cloud for auto-scaling features. A cross-cloud deployment of app server helps the users achieve the in-house database with autoscaled public cloud web application server together.

9.3 Scenario 3: Arbitrary deployments:

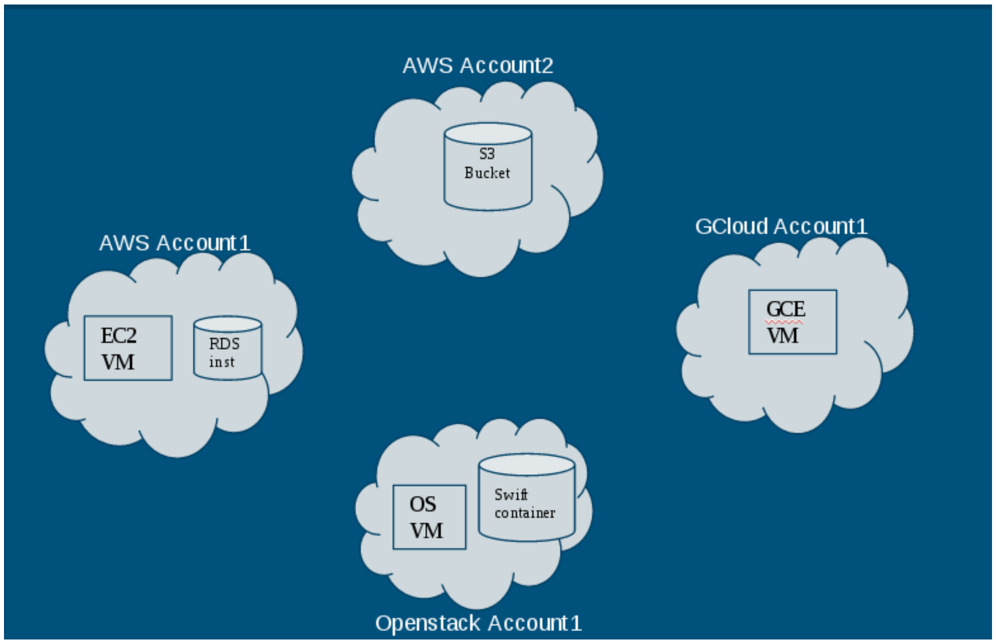


Figure 9-4 Arbitrary cross-cloud deployments

With varying application requirements, day to day cloud deployments inclines towards being random in nature. There can be many cloud accounts involved in provisioning the resources distributed across all the account of cloud providers.

With scenarios mentioned above we can conclude that having a distributed resource deployment among different cloud providers can not only be cost effective but also increases the easier management and availability of the application. Given a tool to provision the multi-cloud resources, organization who cannot compromise on the downtime can be most benefitted with services from different cloud services.

Chapter 10

Configuration Management tools and Types

10.1 Configuration management:

Applications deployed on the cloud or on premise needs a little or heavy configuration depending on the how complex the software is designed. Configuration management involves in every stage of the software development life cycle, i.e., from creating development environment to the delivery of the product designed. Many configuration management tools came into existence to solve the config management of various software systems in IT. Among all the CM tools most prominent are Puppet [10], Chef [11] and Ansible [12].

Configuration management tools (CM tool) manage a single/group of nodes know as targets. A target node can be a machine, virtual machine or a network resource identified by a fully qualified domain name, an IP address or a hostname. Each CM tool has its set of modules to achieve specific tasks to ensure a state of a node. A node is ascertained to attain a state when it confirms to state file specification (Configuration file). Configuration files defines a set of rules and regulations about how a target environment should be. The Content in the state file is written in domain specific language on CM tool used. For example, Ansible uses YAML format for the state file, whereas Puppet follows manifest files in Puppet declarative language.

CM tools are used not only for the application configuration, but also designed for various tasks such as cloud provisioning, app deployment, and orchestration and monitoring of resources with or without the support of the third-party modules.

Following are the prominent properties to for a config management tool [51,52,53]:

1. State enforcement of targets nodes
2. Idempotency of Software Installations and Configurations
3. Secure Authentication with node

There are two categories of automation tools:

1. Pull model

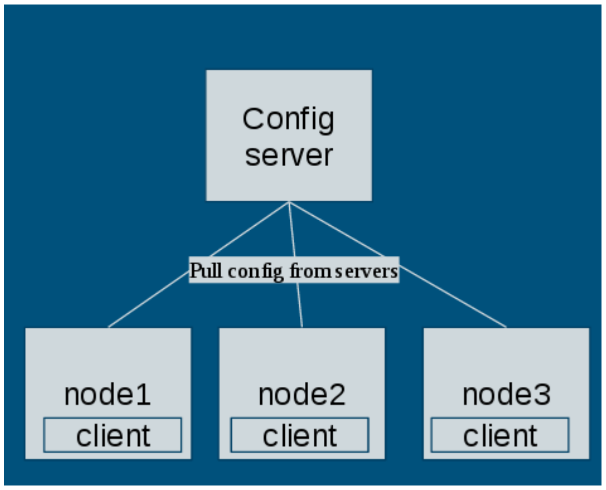


Figure 10-1 Pull model in configuration management tools

In this model a master-slave type of communication is set up between clients and configuration server. Each node managed by configuration server is installed with client software. Client software communicates with the configuration server to get all the necessary software and files to configure themselves periodically.

Each target node is configured with the client which is packaged with the environment to enforce the configurations. It also need not wait for any command from the configuration server to executes the scripts. Pull model is most suitable for the deployments which do not have an order dependency. Order dependency is a state in which configuration of a node is highly dependent on another. Client software on target nodes polls new configurations from the configuration server which makes the changes on the client nodes eventual but not instantaneous. As the client node always maintains details of the master node, if the client node is compromised by any attack, there are high chances for security violations in the master node or corrupting the whole system.

Push model

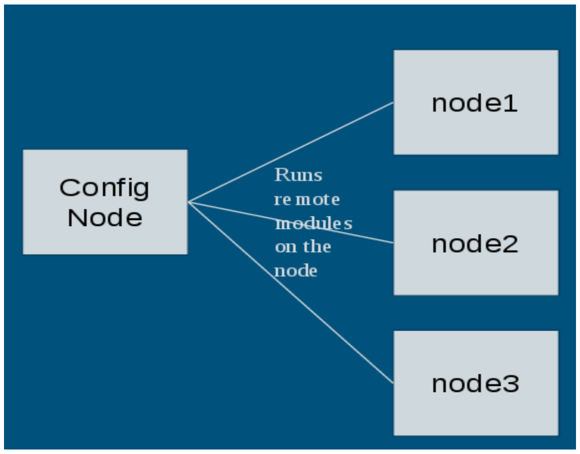


Figure 10-2 Push model in configuration management tools

Push model in contrast to Pull, a configuration server connects to the remote servers and run the configuration scripts. In push model the configuration of a node is entirely dependent on the server, i.e., unless and until server issues a configuration scripts to be run nodes are incapable of achieving the desired configuration state. Due to this, when performing a massive scale deployments ranging from 100 to 1000 server, one must scale up the configuration server to manage the instances. Due to the high coordination between the configuration server and client nodes, Push model can also achieve conditional configuration for dependent deployments. Further, on demand connection-oriented Push is highly secure because server details are not getting stored on managed nodes. The changes made are always instantaneous as there is no poll to master node.

Comparing both models, we can conclude that Push model is easier to manage nodes as there are no installations or client polling is involved. However, for large scale deployments push model might often fail due to process overload of managing a large number of remote connections together. One should also note that additional customizations like installation of client software in targets are required when we use pull model where as in Push it is not necessary.

Chapter 11

Infrastructure as Code

Infrastructure as Code (IAC) [53] is infrastructure management methodology where the operations team can define their infrastructure requirements in version-controlled files and use them to automate the provisioning activities rather than using a manual process. Infrastructure as Code is referred as programmable infrastructure because of runtime ever changing parameters of the infrastructure. IAC concept is highly similar to programming scripts written by the IT administrators on a day to day basis for process automation. Scripts in general are defined as series of instructions written in high-level programming languages such as shell, Python, or Perl to automate redundant tasks. These tasks can be varying from simple installation of software to configuring them multi-node setup of cluster across. In IAC the scripts are written in more high-level descriptive and easily understandable language to do the same. These languages are custom and designed according to specifications of configuration management tools like Ansible Puppet and Chef.

In IAC based coding an IAC script is written compatible to configuration management tools which parse the script and perform the operations accordingly. As an example an IAC script written in Ansible YAML syntax can perform operations as follows:

- Installation of software
- Configure user accounts to set permissions for the software
- Ensure the installed software is running correctly or not
- Copy the configuration files from an external repository

IAC process is also similar to the software development life cycle. While provisioning infrastructure, the requirements of the application are being gathered. Based on the needs of the machines, configuration scripts are laid out. Once the scripts are ready, they are rigorously tested before they go on for the usage.

Though IAC has its benefits in ease of management, it has its downsides too. One has to be very careful about while development of IAC scripts because there should not be any scope for errors in IAC because they can proliferate quickly making the whole deployment glitch or go down. Ansible, Puppet, and Chef are some of the configuration tools enabling IAC type management scripts.

// Chapter 12

Multi-cloud strategy and benefits

A deployment strategy in which two or more cloud service providers are involved in the design of computing environment is said to be a multi-cloud strategy [54]. Using multi-cloud strategy one can avoid the data loss or downtime of the application in a cloud environment. Further, it can also increase the performance at a much lesser price than single cloud service strategy. Clouds are prone to failure regarding network or hardware. Failures can also be due to a natural disaster or a side effect of cyber warfare. Customers having certain functional requirements for their software need not find all services with one cloud provider. The requirements can be as accurate as the load time of website or compute optimized hardware for the software. With a wide variety of the services provided by all the cloud providers, customer can carefully choose a right mix of requirements from all the cloud providers.

Overall benefits of the multi-cloud approach are as follows:

1. Infrastructure and software redundancy: With multi-region deployments across the multiple clouds, the software will be fault tolerant, i.e., even if it fails in one cloud provider, the other can get back application up.

2. Optimized network routing: All the cloud providers connected through high-speed lease lines simulate a point-to-point connection between machines. Further, some clouds are good at small data transfers; others might be good for larger requests. Thus, potential intelligent routing can achieve less response time for across users of application across the world.

3. Avoid catastrophic failures: According to [54], on August 7, 2011, Amazon experienced an outage at its cloud computing hub located in Dublin, Ireland, apparently caused by an electrical transformer malfunction. On February 29, 2012, Microsoft's Azure cloud management system experienced an outage that adversely affected

users in few places of the United States and Europe for several hours. If the multi-cloud approach was implemented the failure could have been prevented.

Thus, with the given advantages, the multi-cloud approach is always fruitful for large scale deployments which cannot afford failures.

Chapter 13

Automation Tools and Tradeoffs

There are many automation tools which are suitable for cloud service automation. Among them, Chef, Puppet, and Ansible are most popular ones.

This chapter discusses various automation tools and their comparison matrix.

13.1 Chef:

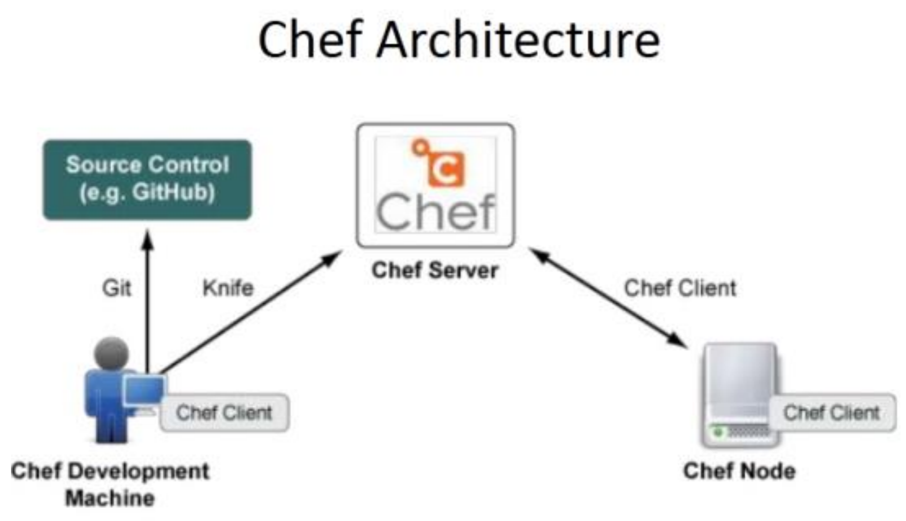


Figure 13-1 Chef architecture [55]

Overview:

Chef is a popular automation tool among the system administrators. Chef also follows infrastructure as a code approach and is agnostic of the environment which administrators operate. It also operates in the cloud, on-premise, and hybrid environment using a chef syntax based files known as cookbooks. Infrastructure maintained through chef can be configured and deployed with ease.

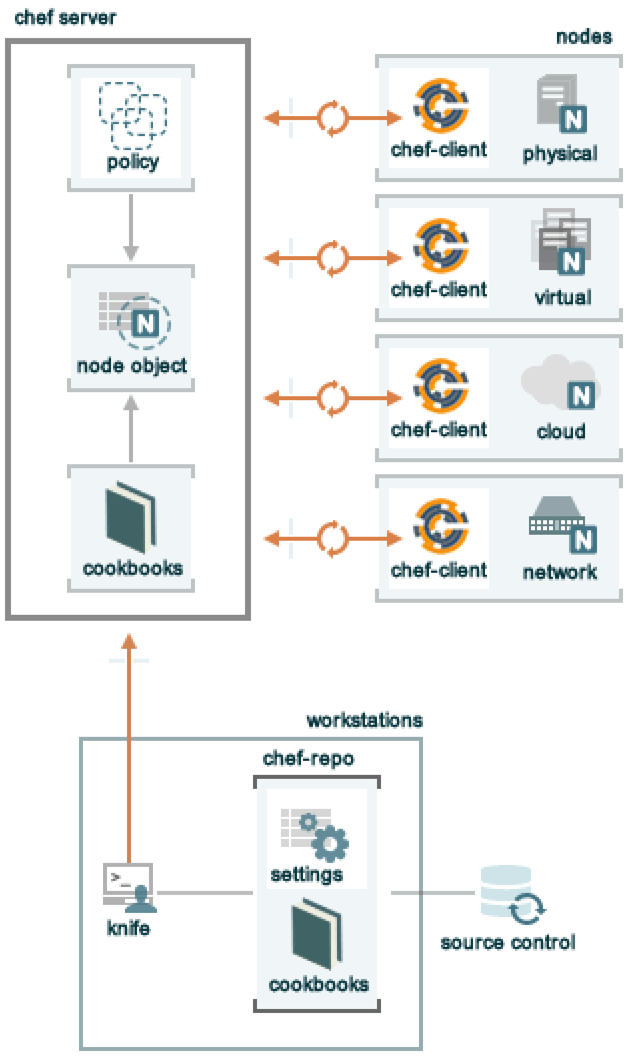


Figure 13-2 Chef Components

The following are the core Chef components essential for configuration management:

1. Workstation
2. Nodes
3. Chef server

Chef workstation:

The work station is management component of chef architecture. It is responsible for managing the chef server, installing chef development kit, and provide command line tools to upload, download and manage the configuration cookbooks on to the remote repositories. It also maintains a local repository of cookbooks which will be eventually pushed on to remote chef server to perform the automation. There can be more than one workstation in an organization managing the nodes maintained by chef server.

Duties of the workstation:

- Developing cookbooks and recipes.
- Keeping the chef-repo synchronized with version source control.
- Using command-line tools.
- Configuring organizational policy, including defining roles and environments and ensuring that critical data is stored in data bags.
- Interacting with nodes, as (or when) required, such as performing a bootstrap operation.

Nodes:

A node can be defined as a resource that is solely managed by Chef Server. A node can be a physical machine, virtual machine, cloud resource, or a network switch where a chef-client will be installed to run the automation scripts.

Chef server:

To manage and maintain all the automated and configured resources, Chef servers act a single endpoint for all the requests. The requests are made from the workstation to initiate the updates on the cookbooks which in turn will be pulled by the respective clients installed on the managed resources. It also acts as a repository of configuration information, policy compliance, and metadata for each registered chef-client. Chef server is usually configured to auto scale through load balancer for a higher load of requests.

Chef workflow:

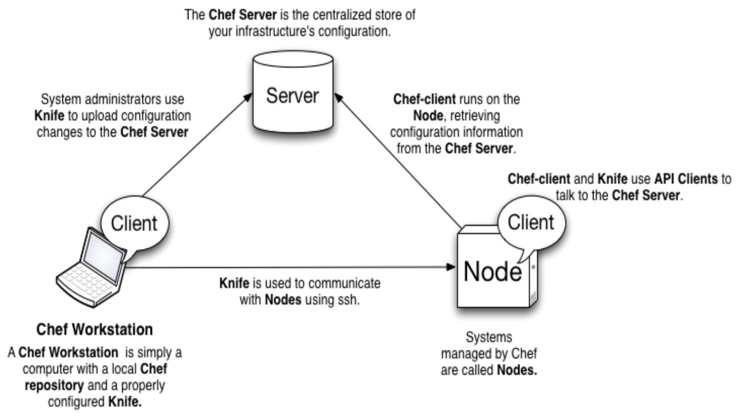

The **Chef Server** is the centralized store of your infrastructure's configuration.
Server
System administrators use **Knife** to upload configuration changes to the **Chef Server**
Chef-client runs on the **Node**, retrieving configuration information from the **Chef Server**.
Chef-client and **Knife** use **API Clients** to talk to the **Chef Server**.
Client
Client
Node
Knife is used to communicate with **Nodes** using ssh.
Chef Workstation
A **Chef Workstation** is simply a computer with a local **Chef repository** and a properly configured **Knife.**
Systems managed by Chef are called **Nodes.**

Figure 13-3 Chef workflow

To automate and enforce configuration, the following workflow is followed:

1. Using knife from the workstation, chef communicates with nodes through SSH and installs chef-client.

2. Chef client running on the remote node is configured with a list of chef servers.

3. Chef client fetches latest configuration files from the chef server and runs chef cookbooks to ensures the state of the node.

4. Cookbooks and scripts on Chef server can be edited through a knife client in a chef workstation.

13.2 Puppet

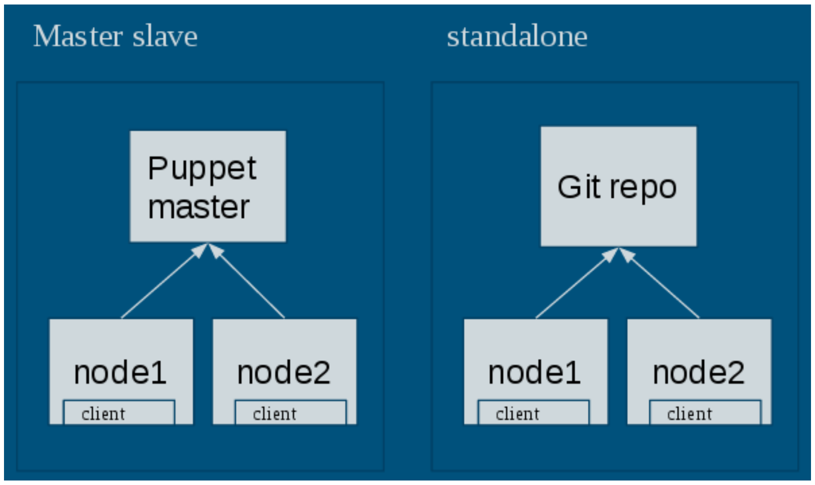


Figure 13-4 Puppet architectures

Puppet is a powerful configuration management tool developed in Ruby to deploy manage and maintain servers. It is used to maintain whole life cycle of a server machine which varies from the bootstrapping of machines with necessary configurations to updated server application and tears down the server. Puppet follows a master-slave architecture, where the slaves are servers managed by puppet. It also performs continuous checks on configurations of the servers and ensures state stability whenever a configuration drifts happen in the slaves. Each slave in puppet managed nodes is installed with a puppet agent which pulls the configurations known as manifests from the master periodically. Puppet agents can also run in a server-less architecture by polling configurations from the remote Git repository instead of the server serving the configuration files.

Puppet configurations are defined in Puppet domain specific language which uses IAC approach to abstract the low-level server configurations to from the user. Puppet is highly supported open source project due to its developer base.

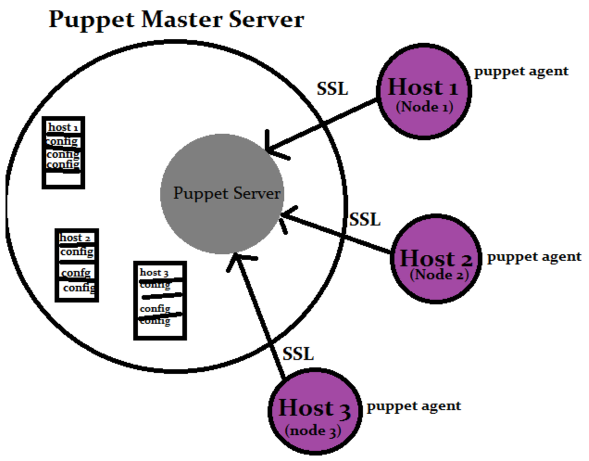


Figure 13-5 Puppet components [56]

Like Chef, Puppet also follows master-slave architecture. The following are components of puppet architecture:

Puppet master:

Puppet Master is a software that runs as a daemon on master server accepting period Pull requests from the clients known as puppet agents.

Puppet agent: Puppet agent is a daemon process that runs on all the hosts which are maintained by Puppet server. Puppet agent is responsible for fetching updated configurations scripts from the master server and run scripts if the host machine is differing expected state from the master server. All the communication happening between master and agent is done through an encrypted SSL channel. Puppet agent daemon is polled for new data usually with an admin configurable time interval ensuring updated files.

Puppet workflow:

Step 1: Whenever a connection is established between the master, and the client, Puppet master fetches the metadata about the client machine using a tool called as Facter, and decides the configuration to be applied to the node.

Step 2: Puppet master fetches all the configuration scripts to run on the puppet agent and complies it into a catalog file. This catalog file is transferred to puppet agent using SSL channel.

Step 3: Puppet agent daemon will run all the configurations mentioned in the catalog. Once the client node agent successfully attains the configurations, it reports back the status to the master server.

13.3 Ansible

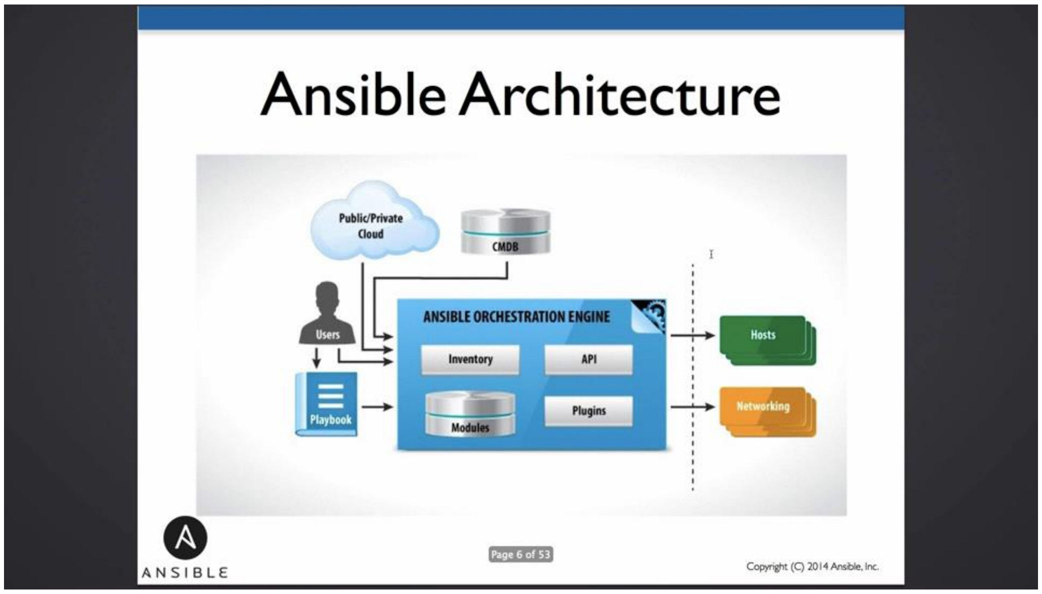


Figure 13-6 Ansible Architecture [76]

Ansible is a python based configuration management, deployment and orchestration tool. Ansible is designed in such a way to replace the everyday automation scripts with the help of YAML files known as playbooks. With a clear abstraction of the underlying operating system, Ansible playbooks run on any machine which has an SSH access and Python installed onto it. Having a generic YAML based syntax, Ansible has a very low learning curve for the end users. Like every other configuration management tool, it maintains the idempotency while running the Playbooks (Ansible specific files). According to [57], Idempotency is defined as a concept that change commands should only be applied when they need to be applied, and that it is better to describe the desired state of a system than the process of how to get to that state. With outstanding features packed into an agentless architecture Ansible sustains as consistent, reliable, and secure among the configuration management tools.

Ansible follows a Pull model based architecture. Though Ansible can be run in a Pull model, like the client based designs, it's more prominent in PUSH model. Moreover, it does not require any software to be installed on remote machines (target nodes which are configured) to make it run. As most of the Linux based devices are pre-built with python by default, it can manage all varieties of operating systems. If Python is not installed on the target nodes, with a little friction Ansible gives a provision to bootstrap the node before running configuration scripts on it. In the case of Windows based machine, Ansible needs Winrm to be configured which will be a burden to be taken while handling the windows based machines.

Ansible master node authenticates itself with the target nodes based on the SSH key pairs, or username/passwords. Further, it runs the scripts without administrator permissions unless it requires to carry out operations that need privilege acceleration. Another Major performance advantage of Ansible is while running scripts on the remote machines; no resources are consumed by them. With traits above, Ansible acts an ideal tool that can be used for secure and highly stable environments.

Components of Ansible:

Ansible Playbooks:

Playbooks are descriptive YAML tasks which are run in the set of hosts provided. Playbooks are divided into a set of plays, and each play is a group of tasks. Task can be defined as basic building block of Ansible playbook which can be an operation that needs to be performed on the local machine or remote machine. Task can be as simple as the installation of software, generating configurations, and restarting a service. Plays defined in playbooks can be run on a set of target nodes known as Inventory.

Inventory:

Inventory is a group of hostname or IP address and their metadata which can be maintained in a file of requested on the run by a script. There are two types of inventories:

1. Static Inventory:

Static inventories are the flat file which consist of host_ips/ DNS names and attributes of targets nodes managed by Ansible. Static inventories are widely maintained and need to be updated from time to time.

Example of a static inventory is as follows:

```
[webservers]
localhost           ansible_connection=local
foo.examplel.com    ansible_connection=ssh        ansible_user=testuser
bar.example.com     ansible_connection=ssh        ansible_user=testuser2

[DBservers]
localhost           ansible_connection=local
bar.example.com     ansible_connection=ssh        ansible_user=testuser2
```

The above is the inventory file that consists of two groups of servers, which are web servers and DB servers. One can specify metadata of the server which can be used by Ansible.

2. Ansible Dynamic Inventory:

Unlike static inventories, dynamic inventory is a script which runs on demand API calls to cloud providers and fetches the metadata of the remote hosts at the time of executing the playbooks. Dynamic inventories are very much useful while managing the cloud-based infrastructure where the metadata for the machines changes from promptly.

Ansible modules:

Ansible offers hundreds of modules for configuration management, system level, and cloud-based operations. Each module is an individual unit of operation which can be carried out on a remote node by referring it to respective parameters inside a playbook. To call a module, one has to call it by referring it inside Ansible Playbook with a task name and required parameters. Ansible modules can be written in any language with the condition that the language runtime needs to be present on the remote machine. Each module should be written in such a way that it allows the machine to attain the desired state and checks whether the task needs to be done in first place. For example, if a module is written to create a file on the remote machine, it will first check whether the file is created already or not and later creates the file if it's not. This helps in maintaining idempotency for multiple runs of the same module. Modules in Ansible can be classified into two categories namely custom modules and core modules. Core modules are packaged with Ansible installations. Custom modules are user defined scripts packaged with Ansible playbooks.

Ansible Roles:

Ansible also supports roles which act a reusable layout for a set of Ansible playbooks. Each role has groups of tasks, variable, and templates. The role is reusable playbook layout which can be referred inside a playbook as role. Whenever a playbook refers to a role, the Ansible assumes role defined and runs all the tasks which are defined inside a role. Ansible Galaxy is a community which has thousands of roles written by developers which can be imported on a command.

With the above components, Ansible also supports the Jinja templates to render the playbooks which enables conditional and repeatable execution of the tasks achieving complex automation with conditional statements and variables.

According to Ansible docs with the help of above components, Ansible is capable of following activities:

• Configuration management - for describing the desired state of a system called manually

or via provisioning tools.

• Application deployment - for pushing out hosted application software to one machine or a

series of machines.

• Orchestration - for coordinating a multi-machine process such as a rolling cluster upgrade.

• As-needed task execution - for performing tasks immediately that do not fit into the

previous models, such as batch server rebooting.

Feature comparision matrix :

	Puppet	Chef	Ansible
Agentless	No	No	Yes
Server Client	Yes	Yes	No
Modular	Yes	Yes	Yes
Mutual Auth	SSL	SSL	SSH
Model	Pull	Pull	Push
Target Node requirements	Puppet Client	Chef Client	Python2.7

Table 13-1 Feature comparison matrix among configuration management tools

The above table depicts the feature comparison among the tools mentioned.

While Puppet and Chef are Pull and Agent-based architecture, Ansible is the only tool that has agent less architecture. On a management perspective, having a server-client model, it is easier to manage updates and track fleets of machines easily using tools such as Puppet and Chef rather than Ansible. When coming to authentication aspect among both the tools SSL authentication prevails due to the easier distribution of SSL certificates than SSH keys. However, since target node requirements are just an SSH server and Python installation, which are prominent in all the Linux based distributions, Ansible is more feasible solution for simple provisioning of services. To design a simple orchestration service which requires one-time configuration setup, Ansible is more suitable due to agentless architecture. But for services which require continuous monitoring and state enforcement tools like Puppet are more appropriate.

Chapter 14

Orchestration Tools

14.1 OpenStack HEAT:

OpenStack is a community maintained collection of open source projects each of the projects has a particular functionality helping the users to build their private cloud. The following are considered as core components of OpenStack:

1. Nova compute [58]: Compute service to provision on-demand virtual machines.
2. Neutron Networking [59]: Networking service to create software defined networks among the virtual machines created.
3. Swift object storage [60]: Object storage service is providing users to upload objects like files with access specifiers.
4. Cinder Block Storage [61]: Block storage service is providing on-demand persistent volumes to virtual machines.
5. Keystone Identity [62]: An OpenStack service that provides API for multi-tenant authorization, client authentication, service discovery.
6. Glance Image [62]: OpenStack Image store service which allows users to upload and retrieve images of virtual machines.

Apart from the core services OpenStack has 13 more optional services addressing different problems of cloud computing [64] Among the optional services, Heat is prominent. It is used to orchestrate the OpenStack services using a declarative template format. It is a designed based on current service of AWS known as Cloud formation. It also adapts the AWS cloud formation syntax partially, providing compatible API calls. Like all the other services of OpenStack, Heat can also be accessible by REST API [65].

HOT (heat orchestration templates) templates are stored in readable file format and reused for orchestrating the multiple services. Each template defines the relationships among different OpenStack resources such as virtual machines, key pairs, and block store volumes. Further, Heat allows some advanced functionality to auto scale application by booting multiple instances on alerts of Ceilometer [70]. Operations performed by Heat can be customized by installing plugins in heat environment.

Example HEAT template [66]:

```
heat_template_version: 2013-05-23

description: Simple template to deploy a single compute instance

resources:
  my_instance:
    type: OS::Nova::Server
    properties:
      image: cirros-0.3.3-x86_64
      flavor: m1.small
      key_name: my_key
      networks:
        - network: private-net
```

From the above the example, we can observe a HEAT template consists of mainly three sections [67]:

1. Heat template version: It's a required parameter in a Heat template. It represents the version of template syntax being used. The current version of Heat template is 2017-02-24 [71]. Given a version, Heat validates per schema and supported features of the templates. Beginning Newton [68] release, Heat follows version 2013-05-23.

2. Description: It's an optional parameter describing the operations of the template, and it is used to provide a description of what the template does.

3. Resources: It is the critical section in the whole template because resources parameter defines all the OpenStack components resource types and their attributes. In above example, only one resource is identified with the name my instance of type OS:: Nova:: Server denoting a Nova compute instance where the properties attribute mentions the image, flavor, public key, and private network used by the instance. A template can have any number of resources.

Apart from the above, we can also specify runtime parameters, optional conditionals for each resource and outputs of HEAT template [69] for more complex deployments.

Once a template is defined with all the resources requires, it can be passed as input to HEAT, and it will take care of the orchestration of all the resources using all the underlying REST API interfaces of other components resulting in a full stack deployment.

14.2 AWS cloud formation:

Amazon web services provide a wide variety of services in Amazon public cloud. Among all the services, Cloud Formation service provides an easy way for management of AWS resources in stacks. Similar to Openstack, Cloud Formation service maintains the AWS infrastructure using template files known as Cloud Formation templates. Templates are written in DSL where the resources are declared in YAML/ JSON format. With the help of these templates, one can easily create a set of resources and their dependencies associated with them. One of the advantages of maintaining the services using Cloud Formation service is, regardless of the order in which the resources are defined, dependencies are resolved automatically. Further, AWS console provides an interactive cloud formation designer [72] to declare resources at ease.

Cloud Formation Template terminology:

AWS templates are highly analogous to Openstack templates in regards to structure. The following are the template attributes to be maintained in a Cloud Formation template1

As above figure, it contains five parts:

1. Description: A simple description of templates about its usage.
2. Parameters: Set runtime parameters for templates for deployment.
3. Resources: AWS resources type definition in accordance with CFN specification [73] and their relationships among them.
4. Outputs: Outputs are the values related to the resources to be visible on a successful provisioning of a stack
5. AWS template format version: A mandatory version of the template which is validated against.

A high-level view of CFN template:

```
{
    "Description" : "A text description for the template usage",
    "Parameters": {
        // A set of inputs used to customize the template per deployment
    },
    "Resources" : {
        // The set of AWS resources and relationships between them
    },
    "Outputs" : {
        // A set of values to be made visible to the stack creator
    },
    "AWSTemplateFormatVersion" : "2010-09-09"
}
```

Example template depicting provisioning of Keypair, and AWS instance:

```
{
    "Description": "Create an EC2 instance running the Amazon Linux 32 bit AMI.",
    "Parameters" : {
        "KeyPair" : {
            "Description": "The EC2 Key Pair to allow SSH access to the instance",
            "Type" : "String"
        }
    },
    "Resources" : {
        "Ec2Instance" : {
            "Type" : "AWS::EC2::Instance",
        "Properties" : {
            "KeyName" : { "Ref" : "KeyPair" },
            "ImageId" : "ami-3b355a52"
        }
      }
   },
   "Outputs" : {
      "InstanceId" : {
        "Description" : "The InstanceId of the newly created EC2 instance",
        "Value" : {
            "Ref" : "Ec2Instance"
        }
      }
   },
   "AWSTemplateFormatVersion" : "2010-09-09"
}
```

In order to provision a stack, the first step is to define a template. The template can be provided as a JSON or YAML formatted string. With the help of templates and other monitoring services such as Cloud Watch, one can create a highly resilient infrastructure. A cloud template can be reused multiple times as the logical names of the resources are different from the actual name of the stack. Thus, any end user can create, delete, and update deployment stack within few clicks using AWS Cloud Formation service.

14.3 Google cloud templates:

Like OpenStack and AWS, as a part of its deployment manager [14], Google Cloud provides infrastructure management service which automates management of resources through templates. Using this service, the end user can deploy variety services such as Google cloud storage, Google Cloud, and Google Cloud SQL. Google deployment manager helps in simplified deployments of Google cloud resources by reusable configuration templates. When compared to Amazon cloud formation and OpenStack HEAT, Google configuration templates are a more simplified version where the resources are just listed and provisioned in order without any syntax.

Google configuration templates:

A configuration is defined as a group of resources and their parameters. Configuration templates are nothing but a YAML representation of resources with their parameters. A resource is a representation of Google Cloud Platform resource. For example, a Compute Engine instance is a resource, in a configuration file, we can define a list of resources, which are later deployed on the cloud using Deployment Manager.

Each resource [74] has three attributes mainly.

1. name - A unique string to identify resources like virtual machines and cloud storage buckets and firewalls.
2. type – Essentially the type of service resource being provisioned. For example compute.v1.instance for virtual machine, compute.v1.disk for storage disk
3. properties – properties are the predefined parameters which are being passed to the resource type. For example, a virtual machine can have zone: attribute to specify the region in which the server needs to be booted.

Example for configuration template:

```
resources:
- name: the-first-vm
  type: compute.v1.instance
  properties:
    zone: us-central1-a
    machineType: https://www.googleapis.com/compute/v1/projects/myproject/zones/us-central1-
a/machineTypes/f1-micro
    disks:
    - deviceName: boot
      type: PERSISTENT
      boot: true
      autoDelete: true
      initializeParams:
        sourceImage: https://www.googleapis.com/compute/v1/projects/debian-cloud/global/images/debian-
7-wheezy-v20150423
        networkInterfaces:
        - network: https://www.googleapis.com/compute/v1/projects/myproject/global/networks/default
    accessConfigs:
    - name: External NAT
       type: ONE_TO_ONE_NAT
```

The above example configuration when deployed gives us a debian7 cloud compute engine instance in us-central1-a of flavor f1-micro.

Templates are like building blocks to create abstractions on sets of resources that are typically deployed together. These templates can be parameterized to enable them to be used over and over by changing input values to define what image to deploy, the zone in which to deploy or how many virtual machines to deploy. Templates enable parallel deployment of resources. Further, they can be controlled programmatically using languages such as Python. Using the deployment manager, one can also preview the entire deployment without actually deploying them like a dry run.

Drawbacks of Orchestration services:

Though the existing orchestration providers give the developers and cloud administrators facility to deploy complex architectures through templates, they have their drawbacks when accessed by API's. The following are the major prominent drawbacks:

1. Orchestration of Multiple accounts: At any instantiation, one can control only one cloud account based on their credentials of the cloud provider. However, in a real-time scenario, cloud administrators might have to manage multiple deployments across the multiple accounts.
2. Support for Multi-cloud: The current orchestration services such as Google Cloud templates and AWS cloud formation supports only their cloud resources, i.e., Google cloud templates can manage only google cloud resources and Amazon can maintain theirs.
3. Facilitating Post provision activities: Though Amazon has ops code to manage the provisioned resources, there is no standardized way to configure the provisioned resources through multiple configuration tools for other cloud providers.

Feature comparison matrix:

	Openstack Heat	AWS CFN	Google Cloud Templates
Learning	High	High	Low
Extendable	Yes (with plugins)	No	No
Customizable	Limited (with plugins)	No	No
Post Provisioning activities	Limited (with cloud-init)	Limited (with cloud-init)	Limited (with cloud-init)
Multi-cloud support	Yes (with plugins)	No	No
Support for other templates	Limited(supports AWS)	No	No

Table 14-1 Feature comparison matrix orchestration tools

The table 14-1 illustrates the feature set comparison among OpenStack Heat, AWS CFN, and Google cloud templates.

We can observe that OpenStack Heat, AWS CFN, Google compute templates have a high learning curve and have limited or no option for customization. Further, we can also derive that existing frameworks CFN and Google cloud templates do not have a multi-cloud resources support as they are in house tools. The post provisioning activities can be carried out through cloud-init module by passing shell scripts. However, it would be an additional burden to be undertaken because each image provisioned is to be configured with cloud-init and scripts need to be written per the machine specifications from time to time.

Chapter 15

Linchpin: Ansible Based Orchestration Tool

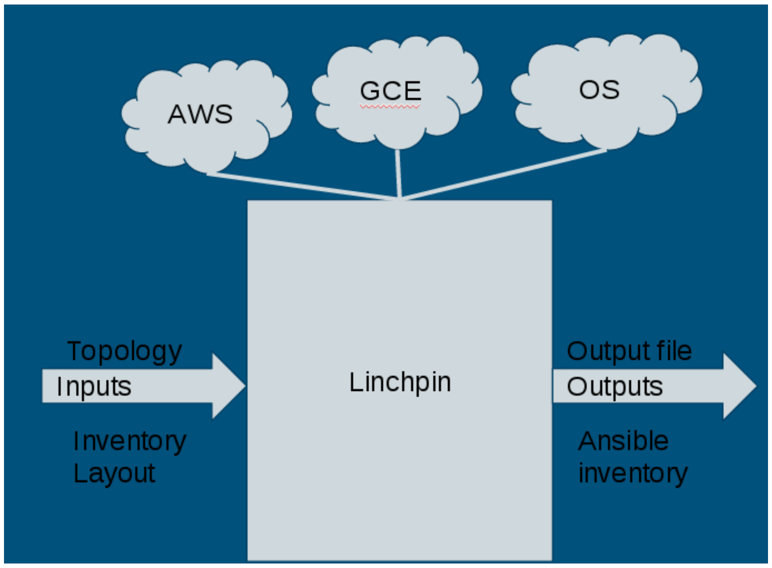


Figure 15-1 Linchpin as Black Box

Linchpin is a tool proposed in this thesis which tries to address the drawbacks of mentioned orchestration tools and the multi-cloud, multi-account provisioning issues. It is entirely written in Ansible which can act as both a configuration management and cloud resource provisioning tool. Though the most of the tool is written in Ansible DSL, it has been tweaked with many custom modules and Python scripts to enable cross-cloud orchestration. Linchpin gets its inspiration Google cloud configuration templates declares cloud resources in files called as topologies.

If we consider Linchpin as a black box, given a topology data, Linchpin breaks down the topologies into multiple RESTful requests and provision resources across the cloud providers based on the cloud type and dumps the details of resources into an output file. Further, it also generates the Ansible inventories per cloud provider based on the requirement.

15.1 Linchpin terminology:

The following are the few terminologies used in defined in Linchpin:

1. Topology: A topology helps us setting multiple cross-cloud deployments. Topology is a deployment file which defines set of resource groups and their metadata in resource group vars.

The structure of topology contains following attributes:

a) Topology name: Name of the topology

b) Resource groups: Resource group are a homogenous collection of resource definitions belonging to a specific cloud, identified by resource group type. Each resource definition constitutes to a resource type belonging to a cloud provider.

c) Resource Group Type (resource_group_type): predefined set of resource group type of different cloud providers, for eg: resource_group_type can be aws, openstack, and gcloud.

d) Resource definition (res_def): Resource definition are the individual unit resources of cloud providers. Each resource definition constitutes to cloud provider resource identified by a resource type (res_type).

e) Resource Type (res_type): res_type is generic label to identify the type of resources. For example, A Google cloud instance has a res_type of gcloud_gce.

f) Resource group variables (res_grp_vars): Resource group vars contain the meta data of the given resource group. These are essentially used for the resource types which need runtime parameters.

g) Associated credentials: Each resource has an attribute assoc_creds which refers to the credentials file to be used to connect to the cloud service maintained in Ansible Vault. One must setup Ansible Vault beforehand to use it.

The figure 15-2 depicts the structure of a Linchpin topology

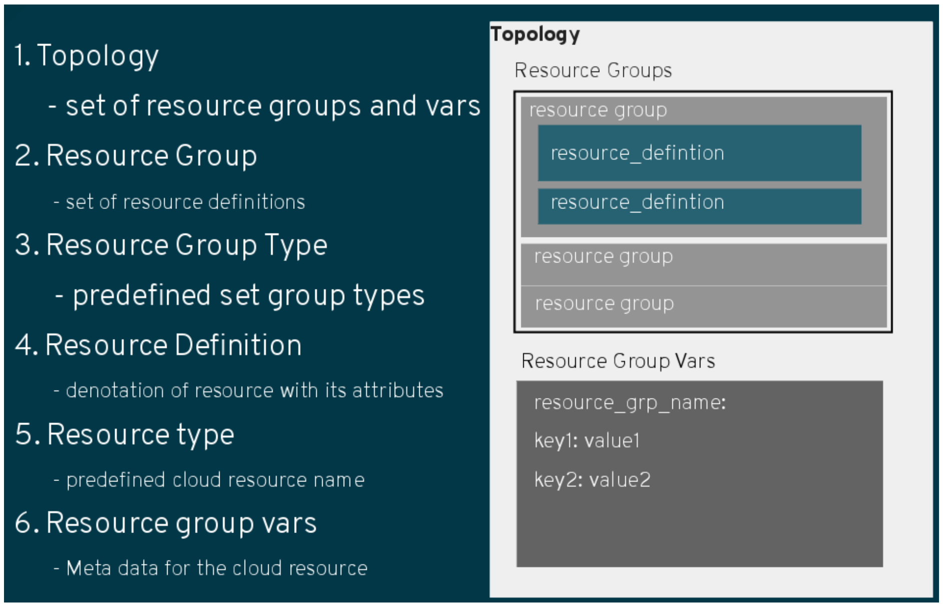


Figure 15-2 Topology structure

The following is an extensive example of all the resources currently supported by Linchpin.

Example Topology:

```
---
  topology_name: "ex_os_aws_gce_topo_full"
  resource_groups:
    -
      resource_group_name: "testgroup1"
      res_group_type: "aws"
      res_defs:
        -
          res_name: "ha_inst2"
          flavor: "t2.micro"
          res_type: "aws_ec2"
          region: "us-east-1"
          image: "ami-fce3c696"
          count: 2
          keypair: "sk_key"
        -
          res_name: "testbucket"
          res_type: "aws_s3"
          region: "us-west-2"
      assoc_creds: "aws _creds"
    -
      resource_group_name: "testgroup2"
      res_group_type: "openstack"
      res_defs:
```

```
      - res_name: "ano_inst_2"
        flavor: "m1.small"
        res_type: "os_server"
        image: "centos5"
        count: 2
        keypair: "testkey"
        networks:
          - "net-openstack"
    -
        res_name: "testresourcesme"
        flavor: "n1-standard-1"
        res_type: "gcloud_gce"
        region: "us-central1-a"
        image: "debian-8"
        count: 2
    assoc_creds: "gcloudsk"
```

The above topology once provisioned, creates three groups of resources among AWS, Openstack and Google cloud. “testgroup1” provisioned will have two AWS EC2 instances of t2.micro size using a ami image in us-east1 region and a s3 bucket named testbucket using aws_creds file from Ansible vault. “testgroup2” which is of type openstack uses openstack creds from Ansible vault and connects to openstack cloud to provision two instances with centos5 distribution and attaches them to network net-openstack. “testgroup3” with res_group_type gcloud provisions the resource definition of two n1-standard-1 instances in us-central1-a region with debain-8 as a distribution using gcloudsk credentials. It also generates output file which consists of data about provisioned resources.

Linchpin outputs:

Once the resources are being provisioned across the cloud each post provisioning output of the resource is being dumped to a file in YAML format.

A sample output of EC2 server provisioning looks as follows:

```
aws_ec2_res:
-   changed: true
    instance_ids:
    - i-0xxxxxxxxxxxxxxd
    - i-0xxxxxxxxxxxxxx5
    instances:
    -   ami_launch_index: '0'
        architecture: x86_64
        block_device_mapping:
            /dev/sda1:
```

```
            delete_on_termination: true
            status: attached
            volume_id: vol-xxxxxxxx
    dns_name: ec2-xx-xx-xx-xxx.compute-1.amazonaws.com
    ebs_optimized: false
    groups:
        sg-xxxxxxxx: default
    hypervisor: xen
    id: i-xxxxxxxxxxxxxxxxx
    image_id: ami-xxxxxxxx
    instance_type: t2.micro
    kernel: null
    key_name: xxxxxx
    launch_time: 'xxxx-xx-xxxxx:xx:xx.xxxx'
    placement: us-east-1a
    private_dns_name: ip-xxx-xx-xx-xx.ec2.internal
    private_ip: xxx.xx.xx.xx
    public_dns_name: ec2-xx-xx-xx-xxx.compute-1.amazonaws.com
    public_ip: xx.xx.xx.xxx
    ramdisk: null
    region: us-east-1
    root_device_name: /dev/sda1
    root_device_type: ebs
    state: running
    state_code: 16
    tags:
        Name: TestInstanceGroup1
        resource_group_name: testgroup1
        test_var1: test_var1 msg is grp1 hello
        test_var2: test_var2 msg is grp1 hello
        test_var3: test_var3 msg is grp1 hello
    tenancy: default
    virtualization_type: hvm
-   ami_launch_index: '1'
    architecture: x86_64
    block_device_mapping:
        /dev/sda1:
            delete_on_termination: true
            status: attached
            volume_id: vol-xxxxxxxx
    dns_name: ec2-xx-xxx-xxx-xxx.compute-1.amazonaws.com
    ebs_optimized: false
    groups:
        sg-xxxxxxxx: default
    hypervisor: xen
    id: i-xxxxxxxxxxxxxxxxx
    image_id: ami-xxxxxxxx
    instance_type: t2.micro
    kernel: null
    key_name: sk_key
    launch_time: 'xxxx-xx-xxxxx:xx:xx.xxxx'
    placement: us-east-1a
    private_dns_name: ip-xxx-xx-xx-xx.ec2.internal
    private_ip: xxx.xx.xx.xx
```

```
      public_dns_name: ec2-xx-xxx-xxx-xxx.compute-1.amazonaws.com
      public_ip: xx.xxx.xxx.xxx
      ramdisk: null
      region: us-east-1
      root_device_name: /dev/sda1
      root_device_type: ebs
      state: running
      state_code: 16
      tags:
        Name: TestInstanceGroup1
        resource_group_name: testgroup1
        test_var1: test_var1 msg is grp1 hello
        test_var2: test_var2 msg is grp1 hello
        test_var3: test_var3 msg is grp1 hello
      tenancy: default
      virtualization_type: hvm
  tagged_instances: []
```

Linchpin Layout:

Layouts are Linchpin specific files which generates the Ansible inventories from output files. Layouts contain the specification of how the outputs are generated into inventories. A layout consists of three parts:

1. Hosts: Host names with IP addresses

2. vars: Ansible specific variable to be passed to the inventory.

3. Host groups: Ansible inventory groups to which hosts belongs to.

A sample layout file looks as mentioned below

```
---
inventory_layout:
  vars:
    openshift_hostip: __IP__
  hosts:
    openshift-master:
      count: 1
      host_groups:
        - all
        - master
    openshift-node:
      count: 1
      host_groups:
        - nodes
  host_groups:
    nodes:
      vars:
        ansible_sudo: False
        ansible_ssh_user: root
```

Given a layout and topology output, Linchpin provides a generic inventory based on some instances present in the output. For example, if provided with two instance topology output, Linchpin generates the following inventory:

```
[nodes]
Openshift-master  openshift_hostname=192.168.2.3
[master]
Openshift-node openshift_hostname=10.2.3.5
[nodes: vars]
ansible_sudo: false
ansible_ssh_user: root
```

These inventories generated can be used for the post provisioning activities by the Ansible to do the software installations and configurations.

Tool Usage:

Since Linchpin is written in Ansible, it can only be accessed by the command line.

The following are the syntax and usage of the command line Linchpin.

Command to provision a Linchpin topology

```
$ ansible-playbook -vvv provision/site.yml -e "topology='/path/to/topology_file'" \
-e 'state=present'
```

Command to teardown a topology

```
ansible-playbook -vvv provision/site.yml -e "topology='/path/to/topology_file'" \
-e 'state=absent'
```

Chapter 16 Linchpin Framework

16.1 The architecture of Linchpin:

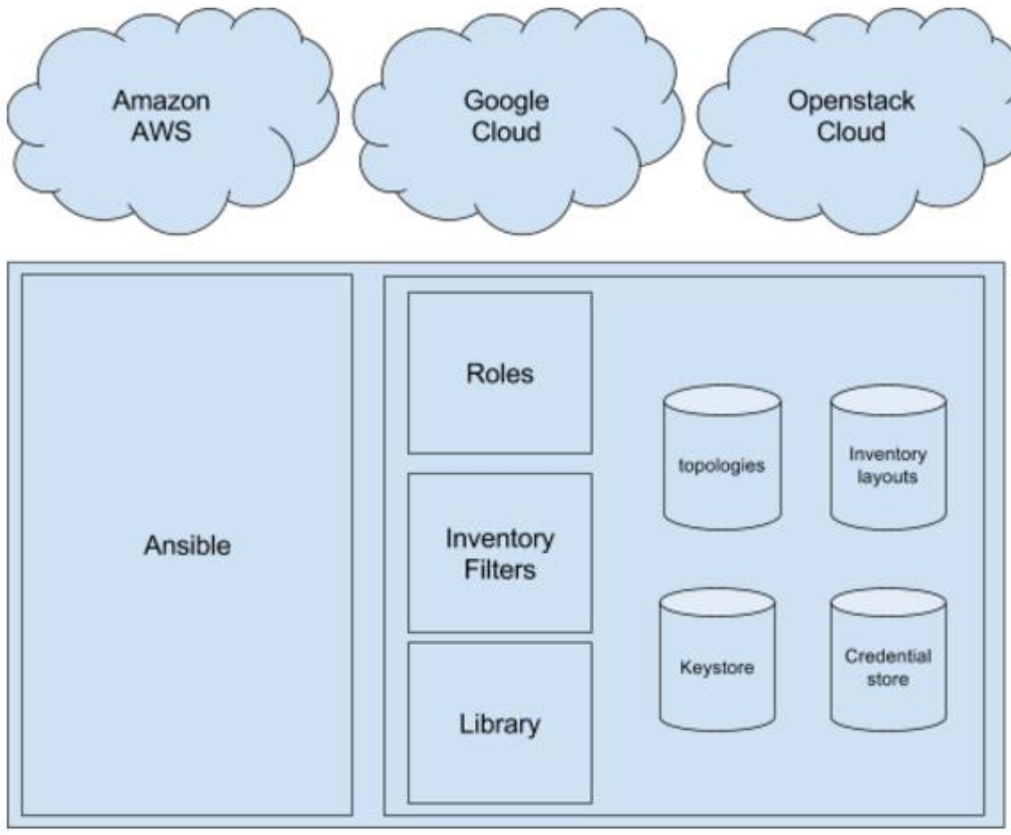


Figure 16-1 Linchpin Architecture

Linchpin primarily consists following components:

1. Ansible: It is the primary component of Linchpin whose main functionality is to parse the topologies and orchestrate the low-level RESTful calls to multiple cloud providers. Further, it is also responsible for storing the credentials in password encrypted vaults.

2. Roles:

Roles are the set of playbooks which are run by Ansible to do the appropriate tasks listed. There are three types of roles in Linchpin environment which are the assumed by Ansible.

a. common role:

Common roles parse the whole topology file being passed to Linchpin with a custom schema_check module. If the validation is successful, it lets Ansible assume respective cloud provider role for provisioning the resources.

b. provisioning role:

The provisioning role consists of tasks to create and delete cloud-specific resources. Each cloud provider has its role defined in Linchpin environment which uses custom cloud modules of Ansible to post requests to cloud providers.

1. aws: Ansible Role responsible for provision and teardown of aws_* type resources
2. openstack: Ansible Role responsible for provision and teardown of os_* type resources
3. gcloud: Ansible Role responsible for provision and teardown of gcloud_* type resources
4. rackspace: Ansible Role responsible for provision and teardown of rax_* type resources
5. libvirt: Ansible Role responsible for provision and teardown of libvirt_* type resources

c. inventory_gen role: Inventory role is responsible for parsing the whole output file of generated by Linchpin and generate Ansible inventory if the layout is specified.

3. Inventory Filters: Inventory filters is a package written in Python used by custom filter plugins to generate generic Ansible inventories regardless of the cloud provider.

4. library: library is collection of Ansible custom modules written.

As a part of Linchpin tool Ansible Module library consists of following:

a. async_status_custom: To enable asynchronous provisioning, each provisioning tasks in Linchpin is assigned with a task_id. Using the task id one can poll for the status of the completion of the task. “async_status_custom” module is responsible for checking the status of each task that is being generated.

b. schema_check: Each topology written constitutes to a schema. Schema_check modules ensure topology is always correct with respect to JSON schema file. If the topology does not validate against the specification schema_check module fails, the provision tasks in the initial stages until the topology is fixed.

c. Output writer module:

   Once provision is successful, each cloud provider dumps out the respective output to a file. Output writer module is responsible for filtering out tasks output and writes it into a topology output file which later can be used for generating the Inventory files.

5. Keystore: As a part of topology we can also create SSH keypairs on the cloud provider. Keystore is the default directory storing all the private SSH keys to the provisioned through topology. These keypairs can be used to connect to provisioned machines and run scripts on them.
6. Credential store: Directory storing all the secret keys and access keys for cloud service APIs to the provisioned machines on the cloud. To ensure the security, this directory is being password encrypted using Ansible vault.
7. Inventory Layouts: Directory containing layouts generating the Ansible inventory file which are used by inventory_gen role to generate inventories.

16.2 Features of Linchpin:

Having a novel architecture, Linchpin not only acts as a tool but also a common framework to unify the existing cloud providers. It is very modular in nature such that even if we would like to integrate a new cloud provider we just need to add one more Ansible role to do a set of operations for respective resources of the cloud. If a cloud provider has a RESTful API to interact with minimal changes one can easily integrate any cloud provider without any discrepancies. One of the interesting feature of Linchpin is, it supports the existing cloud template engines such as Openstack HEAT and AWS cloud formation. Without changing any of existing templates, one can seamlessly reuse them as resources inside a topology. Linchpin environment automatically parses them provision and teardown their respective stacks.

Each template can be created, validated, and provisioned and tore down using the respective commands.

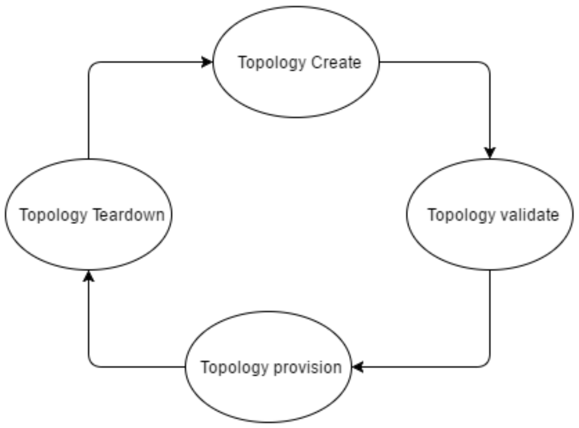


Figure 16-2 Topology Life cycle

16.3 Services Supported of Linchpin:

Currently, Linchpin supports eight services across three cloud providers

1. Amazon Web Services

- Elastic Compute Cloud (EC2)
- Simple storage service (S3)
- Cloud Formation

2. Openstack

- Servers
- Cinder
- Swift
- Heat

3. Google cloud

- Google compute engine

16.4 Scalability of Linchpin:

As Linchpin is implemented mostly through Ansible playbooks, the scalability and performance depend on the machine running Ansible. Usually a machine installed with Ansible can run 10 to 1000 threads in parallel in executing tasks of a playbook. Further, there is provision to delegate partial tasks to multiple machines which result in a highly scalable platform.

Feature comparision matrix among ochestration tools:

	Linchpin	Heat	CFN	GCloud Templates
Learning curve	Low	High	High	Low
Extendable	Yes (with Ansible Roles)	Yes (With plugins)	No	No
Customizability	Highly	High	No	No
Post-Provisioning Activites	Yes (with Ansible Playbooks)	Yes (with Cloud-init)	Yes (AWS opsworks)	No
Multicloud support	Yes	Yes (with Plugins)	No	No
Other template support	Yes	No	No	No
Multi Account support	Yes	No	No	No

Table 16-1 Feature comparison matrix of Linchpin with other orchestration tools

Comparing among the frameworks of HEAT, AWS CFN and Google Deployment manager Linchpin has the following advantages over others:

1. As its written in using Ansible, Linchpin can be highly extendable by using Roles and Modules whereas other frameworks do not support the same.
2. Linchpin acts as a Multi-cloud template by incorporating different cloud resources and their templates together as Linchpin resource types, unlike other services, it’s nearly impossible to include other templates into the same.
3. Having fixed definition of resource types and parameters, the learning curve of the tool is pretty low when compared to others where we need to learn their domain specific language.

4. Linchpin also unifies the vocabulary of multiple cloud parameters into common terms making it single syntax for multi-cloud resources

5. Though the other cloud services provide the post provisioning activities through cloud-init [5] scripts they are highly limited due wide variety distributions.

16.5 Linchpin Workflow : Provisioning topology with inventory generation:

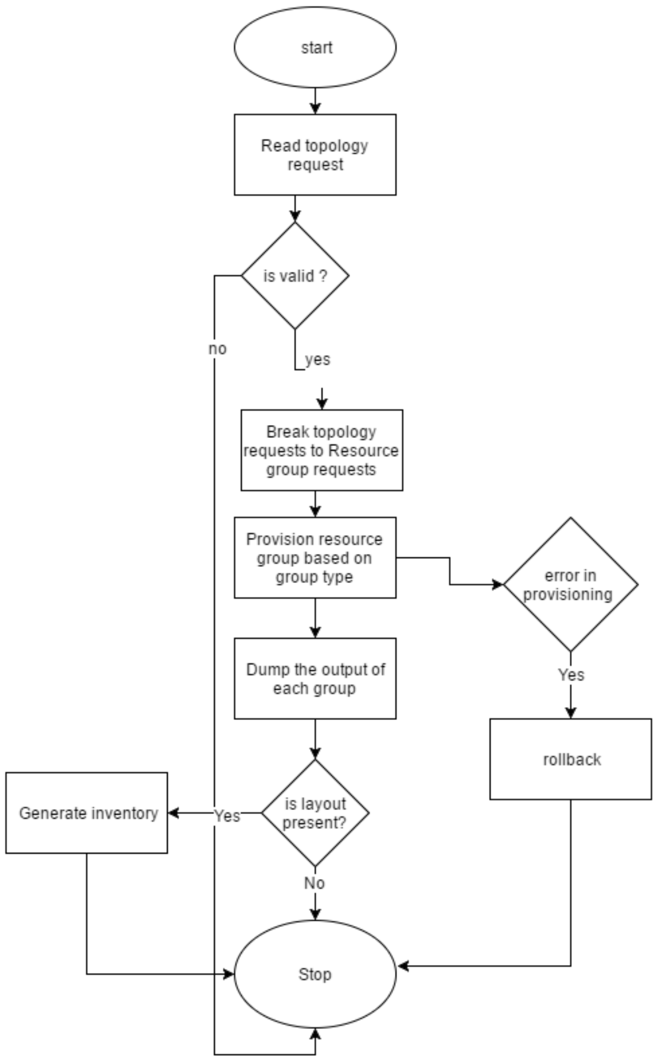

start
Read topology request
is valid ?
no
yes
Break topology requests to Resource group requests
Provision resource group based on group type
error in provisioning
Yes
Dump the output of each group
rollback
Generate inventory
Yes
is layout present?
No
Stop

Figure 16-16-3 Topology provisioning workflow

Workflow of Linchpin Topology Provisioning is as follows:

1. Given a topology Linchpin reads the topology request and loads it as a JSON inside Linchpin environment using common role.

2. Once the topology is loaded, it is validated against the predefined JSON schema. If the topology request is invalid, Linchpin fails and ends the provisioning task.

3. If topology request is valid each resource group is provisioned based on the resource group type if there is any error in provision rollback is performed.

4. If there is no error in provisioning each provisioned instance, details are dumped into an output file.

5. If the layout file is defined, Inventory is generated else it stops the provisioning activity.

16.6 Linchpin Workflow: Topology Teardown:

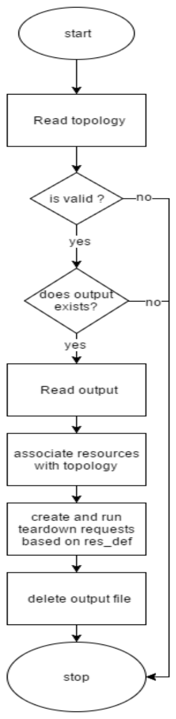


Figure 16-4 Topology Teardown workflow

Workflow of Linchpin Topology teardown:

1. Given a topology Linchpin reads the topology request and loads it as a JSON inside Linchpin environment using common role written.

2. Once the topology is loaded, it is validated against the predefined JSON schema. If the Topology Request is invalid, Linchpin fails and ends the provisioning task.

3. Linchpin searches for the topology output with the file name same as topology name. If the output is not found, the topology stops the teardown process.

4. If the output file is found, each task output is associated with resource definitions inside topology and teardown requests are initiated.

5. Once the requests are completed, the output file is deleted and teardown process is stopped.

Chapter 17

Linchpin Experiments and Performance

The experiments are performed by booting 1 to 5 instances in multiple environments through Linchpin. The same booting is carried out by in other frameworks with available images and flavors. The following are the machine configurations that are used for provisioning the instances:

AWS EC2 Machine configuration:

Operating system: Ubuntu 14.04

Flavor: t2.micro

Memory: 1GB

Storage: 8Gb

Google Cloud Machine configuration:

Operating system: Debian 8

Flavor: g1-small

Memory: 0.6GB

Storage: 8GB

Openstack Machine configuration:

Operating system: RHEL 6.5

Flavor: t1.micro

Memory: 1GB

Storage: 20GB

Host Machine Configuration:

Host type: Virtual machine

Distribution: Fedora 23

Memory: 4GB

Storage: 25GB

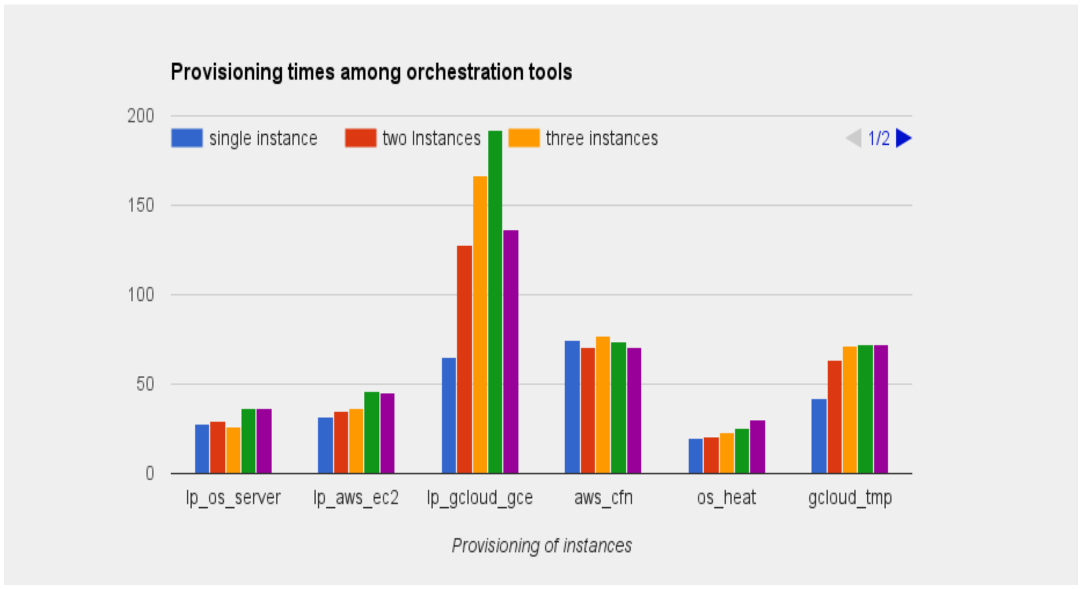
Feature comparison matrix:

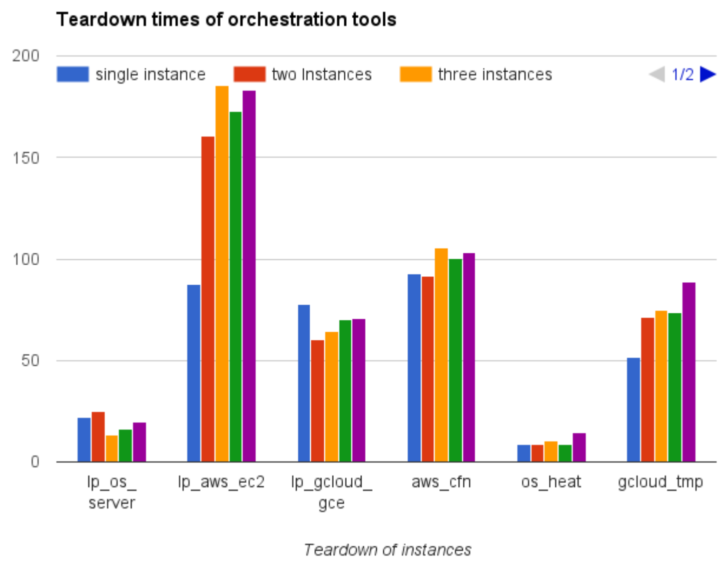
Provisioning	Single Instance	Two Instances	Three Instances	Four Instances	Five Instances
lp_os_server	29.92	29.36	26.30	36.50	36.51
lp_aws_ec2	31.39	34.70	36.14	45.96	44.84
lp_gcloud_gce	65.42	127.38	166.34	192.10	136.71
aws_cfn	19.55	20.92	22.92	25.07	30.46
os_heat	19.55	220.92	22.92	25.07	30.46
gcloud_tmp	41.82	63.22	71.82	72.36	72.33

Table 17-1 Provisioning times Linchpin vs other orchestration services

Teardown	Single Instance	Two Instances	Three Instances	Four Instances	Five Instances
lp_os_server	22.18	24.82	13.21	16.45	19.76
lp_aws_ec2	87.50	160.34	185.50	172.92	183.42
lp_gcloud_gce	77.42	60.42	64.31	70.18	70.60
aws_cfn	93.03	91.39	105.2523	100.46	103.141
os_heat	8.96	8.94	10.28	8.84	14.69
gcloud_tmp	51.38	71.10	74.73	73.59	88.47

Table 17-2 Teardown times using Linchpin vs other orchestration services





Observations:

The performance of Linchpin provisioning and teardown is compared against the other orchestration frameworks by provisioning a cluster of Virtual Machines from a single machine to five machines.

From tables 17-1, 17-2, we can observe the following:

1. Provisioning of aws_ec2 instances through Linchpin is almost 50 % efficient than provisioning instances through AWS cloud formation services.
2. Provisioning of Google Cloud templates is nearly 50% efficient than provisioning through Linchpin. This is because of the timeout caused by Apache Libcloud library that manages Google cloud in Ansible module.
3. Provisioning of OpenStack servers on a private cloud through Linchpin is comparable to the provisioning of instances through OpenStack HEAT.
4. In teardown, the Linchpin is more efficient than Google Cloud templates and less efficient than OpenStack HEAT and AWS Cloud Formation.

Chapter 18

Summary and Conclusion

The research has started with a goal to create extensible unified cloud provisioning system for enabling cross-cloud deployments. Multiple use cases of cross-cloud deployments, frameworks of existing cloud deployments and their drawbacks have been investigated in earlier chapters. Linchpin started as a proof of concept to make use of a configuration management tool to mimic the orchestration services. While developing, it evolved into a complex YAML based provisioning system with its domain specific language. Having a necessity for higher level abstraction for multiple cloud vendors' Linchpin framework come out to be a potential solution for a unified cloud deployment solution. With the highly evolving configuration management tools it has been proved that a framework can be put in place to provision heterogeneous cloud resources. A comparable performance in provision and teardown of Openstack, AWS, Google Cloud has been achieved when compared to their orchestration tools.

Chapter 19

Future Work

With the advent of many new services in the domain of IaaS, PaaS, and SaaS coming up on a day to day basis, a deeper investigation is required to search for potential services to be integrated. Now that, Ansible based framework in place, one can extend to provision services and deployments across unimplemented cloud providers too. Thus, providing support for multiple services helps us understand the interoperability issues and vendor lock-in problems of the cloud providers. A rigorous performance tuning needs to be done in future. Which helps us know more about the design traits of the configuration management tool. Further, the analysis needs to be made on the behavior of the cross-cloud application and to tune them for efficient management. We can also harness the feature of having cross-cloud deployments among the cloud service providers. As an open source project, there can be many frameworks arising based on the Linchpin. In short, Linchpin can be projected as a standalone service for users to deploy their cross-cloud topologies. Dependency relationships and associations need to be implemented in the resource group type of topologies to mimic the functionality of existing orchestration frameworks. Linchpin also acts as a proof of concept that a hybrid cloud deployment of resources is possible with a single endpoint. It could be highly enhanced with more network provisioning features which are currently not implemented.

Current thesis not only proves that configuration management tools can offer more than just configuration management, but also can compete with existing orchestration frameworks forming a unified solution to every service. For ease of use, a topology generator can be designed through a web-based interface to generate a Linchpin topology.

With the help of machine learning models and rest API of existing services, it is highly possible to have intelligent deployments of the existing topology.

References

[1] "Software » OpenStack open source cloud computing software," OpenStack. [Online].

Available: https://www.openstack.org/software/. Accessed: Dec. 5, 2016.

[2] "Amazon web services," in Wikipedia, Wikimedia Foundation, 2016.[Online]. Available: https://en.wikipedia.org/wiki/Amazon_Web_Services. Accessed: Dec. 5, 2016.

[3] "Heat orchestration template (HOT) guide — heat 8.0.0.0b2.dev55 documentation," 2012. [Online].

Available:

http://docs.openstack.org/developer/heat/template_guide/hot_guide.html#heat-orchestration-template-hot-guide. Accessed: Dec. 5, 2016.

[4] "AWS CloudFormation Templates," Amazon Web Services, 2016. [Online]. Available: https://aws.amazon.com/cloudformation/aws-cloudformation-templates/. Accessed: Dec. 5, 2016.

[5] "Documentation — Cloud-Init 0.7.8 documentation,". [Online]. Available: http://cloudinit.readthedocs.io/en/latest/. Accessed: Dec. 5, 2016.

[6] Robert Dukaric and Matjaz B. Juric. 2013. Towards a unified taxonomy and architecture of cloud frameworks. Future Gener. Comput. Syst. 29, 5 (July 2013), 1196-1210. DOI=http://dx.doi.org/10.1016/j.future.2012.09.006

[7] "Microsoft azure," in Wikipedia, Wikimedia Foundation, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Microsoft_Azure. Accessed: Dec. 5, 2016.

[8] "Google cloud platform," in Wikipedia, Wikimedia Foundation, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Google_Cloud_Platform. Accessed: Dec. 5, 2016.

[9] "Welcome to the heat documentation! — heat 8.0.0.0b2.dev55 documentation," 2012. [Online]. Available: http://docs.openstack.org/developer/heat/#heat-s-purpose-and-vision. Accessed: Dec. 5, 2016.

[10] "Puppet(software)," in Wikipedia, Wikimedia Foundation, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Puppet_(software). Accessed: Dec. 5, 2016.

[11] "Chef(software)," in Wikipedia, Wikimedia Foundation, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Chef_(software). Accessed: Dec. 5, 2016.

[12] A. R. Updates, C. Deployment, Z. Downtime, and R. Hat, "Ansible (software)," in Wikipedia, Wikimedia Foundation, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Ansible_(software). Accessed: Dec. 5, 2016.

[13] Distributed, Parallel, and Cluster Computing (cs.DC)
Journal reference:ServiceWave 2011, Second Optimising Cloud Services Workshop, arXiv:1203.0432 [cs.DC]

[14] "Google cloud deployment manager documentation," Google Cloud Platform. [Online]. Available: https://cloud.google.com/deployment-manager/docs/. Accessed: Dec. 5, 2016.

[15] E. Colman, "When to use SaaS, PaaS, and IaaS," ComputeNext, 2013. [Online]. Available: https://www.computenext.com/blog/when-to-use-saas-paas-and-iaas/. Accessed: Dec. 5, 2016.

[16] "Elastic compute cloud (EC2) cloud server & hosting – AWS," Amazon Web Services, 2016. [Online]. Available: https://aws.amazon.com/ec2/. Accessed: Dec. 5, 2016.

[17] "Google compute engine," in Wikipedia, Wikimedia Foundation, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Google_Compute_Engine. Accessed: Dec. 5, 2016.

[18] "Rackspace," in Rackspace. [Online]. Available: https://www.rackspace.com/en-us/cloud/servers. Accessed: Dec. 5, 2016.

[19] "AWS elastic beanstalk – deploy web applications," Amazon Web Services, 2016. [Online]. Available: https://aws.amazon.com/elasticbeanstalk/. Accessed: Dec. 5, 2016.

[20] "App engine - platform as a service | Google cloud platform," Google Cloud Platform. [Online]. Available: https://cloud.google.com/appengine/. Accessed: Dec. 5, 2016.

[21] "OpenShift: PaaS by red hat, built on Docker and Kubernetes,". [Online]. Available: https://www.openshift.com/. Accessed: Dec. 5, 2016.

[22] "Dropbox (service)," in Wikipedia, Wikimedia Foundation, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Dropbox_(service). Accessed: Dec. 5, 2016.

[23] "Box (company)," in Wikipedia, Wikimedia Foundation, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Box_(company). Accessed: Dec. 5, 2016.

[24] "Gmail," in Wikipedia, Wikimedia Foundation, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Gmail. Accessed: Dec. 5, 2016.

[25] "Office 365," in Wikipedia, Wikimedia Foundation, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Office_365. Accessed: Dec. 5, 2016.

[26] "VSphere documentation center,". [Online]. Available: https://pubs.vmware.com/vsphere-50/index.jsp?topic=%2Fcom.vmware.vsphere.vm_admin.doc_50%2FGUID-CEFF6D89-8C19-4143-8C26-4B6D6734D2CB.html. Accessed: Dec. 5, 2016.

[27] "VirtualBox," in Wikipedia, Wikimedia Foundation, 2016. [Online]. Available: https://en.wikipedia.org/wiki/VirtualBox. Accessed: Dec. 5, 2016.

[28] "Xen," in Wikipedia, Wikimedia Foundation, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Xen. Accessed: Dec. 5, 2016.

[29] "Hyper-V," in Wikipedia, Wikimedia Foundation, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Hyper-V. Accessed: Dec. 5, 2016.

[30] Datadog, "8 surprising facts about real Docker adoption," Datadog Instrastructure Monitoring. [Online]. Available: https://www.datadoghq.com/docker-adoption/. Accessed: Dec. 5, 2016.

[31] "Hypervisor," in Wikipedia, Wikimedia Foundation, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Hypervisor. Accessed: Dec. 5, 2016.

[32] S. J. Vaughan-Nichols, "Containers vs. Virtual machines: How to tell which is the right choice for your enterprise," ITworld, 2015. [Online]. Available: http://www.itworld.com/article/2915530/virtualization/containers-vs-virtual-machines-how-to-tell-which-is-the-right-choice-for-your-enterprise.html. Accessed: Dec. 5, 2016.

[33] Metz, "17 great gifts for the runner in your life," in Business, WIRED, 2015. [Online]. Available: https://www.wired.com/2015/07/1812997/. Accessed: Dec. 5, 2016.

[34] M. Rouse, "What is continuous integration (CI)? - definition from WhatIs.com," SearchSoftwareQuality, 2008. [Online]. Available: http://searchsoftwarequality.techtarget.com/definition/continuous-integration. Accessed: Dec. 5, 2016.

[35] P. M. Duvall, S. Matyas, and A. Glover, Continuous integration: Improving software quality and reducing risk, 2nd ed. United States: Addison-Wesley Educational Publishers, 2007.

[36] M. Rouse, "What is continuous delivery (CD)? - definition from WhatIs.com," SearchITOperations, 2016. [Online]. Available: http://searchitoperations.techtarget.com/definition/continuous-delivery-CD. Accessed: Dec. 5, 2016.

[37] T. Miller, "Continuous integration and delivery in the cloud: How RightScale does it," 2014. [Online]. Available: http://www.rightscale.com/blog/cloud-management-best-practices/continuous-integration-and-delivery-cloud-how-rightscale-does-it. Accessed: Dec. 5, 2016.

[38] S. Yegulalp, "Apache Libcloud provides single python API for all clouds," InfoWorld, 2016. [Online]. Available: http://www.infoworld.com/article/3088239/apis/apache-libcloud-provides-single-python-api-for-all-clouds.html. Accessed: Dec. 5, 2016.

[39] G. Reese, "The Dasein cloud API," 2009. [Online]. Available: http://broadcast.oreilly.com/2009/08/dasein-open-cloud-api.html. Accessed: Dec. 5, 2016.

[40] B. Butler, "And the cloud provider with the best uptime in 2015 is…," Network World, 2016. [Online]. Available:

http://www.networkworld.com/article/3020235/cloud-computing/and-the-cloud-provider-with-the-best-uptime-in-2015-is.html. Accessed: Dec. 5, 2016.

[41] M. Rouse, "What is CloudWatch ? - definition from WhatIs.com," SearchAWS, 2014. [Online]. Available: http://searchaws.techtarget.com/definition/CloudWatch. Accessed: Dec. 5, 2016.

[42] "Setting up OAuth 2.0 - API console help," 2016. [Online]. Available: https://support.google.com/googleapi/answer/6158849?hl=en. Accessed: Dec. 5, 2016.

[43] "IAM best practices," Amazon Web Services, 2016. [Online]. Available: https://aws.amazon.com/iam/. Accessed: Dec. 3, 2016.

[44] C. Bateman, "Multi-cloud approach can cut costs significantly - Datapipe Blog," Datapipe Blog, 2015. [Online]. Available: https://www.datapipe.com/blog/2015/08/17/multi-cloud-approach-can-cut-costs-significantly/. Accessed: Dec. 3, 2016.

[45] E. Banks, "Docker networking 101 – the defaults," in DevOps, Das Blinken Lichten, 2015. [Online]. Available: http://www.dasblinkenlichten.com/docker-networking-101/. Accessed: Dec. 3, 2016.

[46] "Service accounts," in Google cloud. [Online]. Available: https://cloud.google.com/compute/docs/access/service-accounts. Accessed: Dec. 5, 2016.

[47] A. Bloom, "The definition of ‘Cross platform cloud Applications’ - VMware cloud management," in Application Director, VMware Cloud Management, 2013. [Online]. Available: http://blogs.vmware.com/management/2013/01/the-definition-of-cross-platform-cloud-applications.html. Accessed: Dec. 3, 2016.

[48] P. de Leusse, B. Kwolek, and K. Zieliński, “Toward governance of cross-cloud application deployment,” in ServiceWave 2011, Second Optimising Cloud Services Workshop, 2011. [Online]. Available: http://arxiv.org/abs/1203.0432

[49] A. C. -MSFT, "Master slave architecture in Hadoop," Avkash Chauhan’s Blog, 2012. [Online]. Available: https://blogs.msdn.microsoft.com/avkashchauhan/2012/02/24/master-slave-architecture-in-hadoop/. Accessed: Dec. 5, 2016.

[50] M. Mainguy and V. my complete profile, "Push versus pull deployment models," 2011. [Online]. Available: http://mikemainguy.blogspot.com/2011/08/push-versus-pull-deployment-models.html. Accessed: Dec. 5, 2016.

[51] G. Gheorghiu and V. my complete profile, "Automated deployment systems: Push vs. Pull," 2010. [Online]. Available: http://agiletesting.blogspot.com/2010/03/automated-deployment-systems-push-vs.html. Accessed: Dec. 5, 2016.

[52] I. Bicking, "Configuration management: Push vs. Pull," 2010. [Online]. Available: http://www.ianbicking.org/blog/2010/03/configuration-management-push-vs-pull.html. Accessed: Dec. 5, 2016.

[53] M. Rouse, "What is infrastructure as code (IAC)? - definition from WhatIs.com," SearchITOperations, 2015. [Online]. Available: http://searchitoperations.techtarget.com/definition/Infrastructure-as-Code-IAC. Accessed: Dec. 5, 2016.

[54] M. Rouse, "What is multi-cloud strategy? - definition from WhatIs.com," SearchCloudApplications, 2016. [Online]. Available:

http://searchcloudapplications.techtarget.com/definition/multi-cloud-strategy. Accessed: Dec. 5, 2016.

[55] A. Maurya, "Openstack: Automated deployment {Chef, puppet, Heat}," Anil Maurya's Blog, 2014. [Online]. Available: https://anilmaurya.wordpress.com/2014/10/13/openstack-automated-deployment-chef-puppet-heat/. Accessed: Dec. 5, 2016.

[56] S. Pillai, "Puppet Tutorial: How does puppet work," in Slashroot, slashroot.in, 2012. [Online]. Available: http://www.slashroot.in/puppet-tutorial-how-does-puppet-work. Accessed: Dec. 5, 2016.

[57] RedHat, "Glossary — Ansible documentation," 2016. [Online]. Available: http://docs.ansible.com/ansible/glossary.html. Accessed: Dec. 5, 2016.

[58] "Software » OpenStack open source cloud computing software," OpenStack. [Online]. Available: https://www.openstack.org/software/releases/mitaka/components/nova. Accessed: Dec. 5, 2016.

[59] "Neutron,". [Online]. Available: https://wiki.openstack.org/wiki/Neutron. Accessed: Dec. 5, 2016

[60] "Swift,". [Online]. Available: https://wiki.openstack.org/wiki/Swift. Accessed: Dec. 5, 2016.

[61] "Cinder,". [Online]. Available: https://wiki.openstack.org/wiki/Cinder. Accessed: Dec. 5, 2016.

[62] "Welcome to glance's documentation! — glance 14.0.0.0b2.dev11 documentation," 2010. [Online]. Available: http://docs.openstack.org/developer/glance/. Accessed: Dec. 5, 2016.

[63] "Keystone, the OpenStack identity service — keystone 11.0.0.0b2.dev90 documentation," 2012. [Online]. Available: http://docs.openstack.org/developer/keystone/. Accessed: Dec. 5, 2016.

[64] "Software » OpenStack open source cloud computing software," OpenStack. [Online]. Available: https://www.openstack.org/software/project-navigator. Accessed: Dec. 5, 2016.

[65] "OpenStack Docs: Orchestration service APIs,". [Online]. Available: http://developer.openstack.org/api-ref/orchestration/index.html. Accessed: Dec. 5, 2016.

[66] "Rackspace developer center," 2014. [Online]. Available: https://developer.rackspace.com/blog/openstack-orchestration-in-depth-part-1-introduction-to-heat/. Accessed: Dec. 5, 2016.

[67] "Heat orchestration template (HOT) specification — heat 8.0.0.0b2.dev55 documentation," 2012. [Online]. Available: http://docs.openstack.org/developer/heat/template_guide/hot_spec.html. Accessed: Dec. 5, 2016.

[68] "Openstack Releases,". [Online]. Available: https://releases.openstack.org/newton/. Accessed: Dec. 5, 2016.

[69] "Heat orchestration template (HOT) specification — heat 8.0.0.0b2.dev55 documentation," 2012. [Online]. Available: http://docs.openstack.org/developer/heat/template_guide/hot_spec.html#outputs-section. Accessed: Dec. 5, 2016.

[70] "Telemetry,". [Online]. Available: https://wiki.openstack.org/wiki/Telemetry. Accessed: Dec. 5, 2016.

[71] "Heat orchestration template (HOT) specification — heat 8.0.0.0b2.dev55 documentation," 2012. [Online]. Available: http://docs.openstack.org/developer/heat/template_guide/hot_spec.html#ocata. Accessed: Dec. 5, 2016.

[72] "New – AWS CloudFormation designer + support for more services," 2015. [Online]. Available: https://aws.amazon.com/blogs/aws/new-aws-cloudformation-designer-support-for-more-services/. Accessed: Dec. 5, 2016.

[73] "AWS resource types reference," 2016. [Online]. Available: http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-template-resource-type-ref.html. Accessed: Dec. 5, 2016.

[74] "Deployment manager fundamentals," Google Cloud Platform, 2016. [Online]. Available: https://cloud.google.com/deployment-manager/fundamentals. Accessed: Dec. 5, 2016.

[75] Terry, "Terry: Chef," 2013. [Online]. Available: https://terry.im/wiki/terry/Chef.html. Accessed: Dec. 5, 2016.

[76] C. Patra, "Ansible and AWS: Cloud IT automation management," in Amazon Web Services, Cloud Academy Blog, 2015. [Online]. Available: http://cloudacademy.com/blog/ansible-aws/. Accessed: Dec. 5, 2016.

Biographical Information

Samvaran Kashyap Rallabandi joined the University of Texas at Arlington in Spring 2015. He received his B.Tech in Computer Science and Engineering from Jawaharlal Technological University in 2013.

He worked as a worked as Software Engineer for one year in Persistent Systems Ltd. In U.S he worked as a graduate teaching assistant at UTA and interned at RedHat Inc. His current research interests are in areas of Cloud computing, DevOps, Software Automation and Sofware design and development.