Multi-Cloud Automation with Ansible

Automate, orchestrate, and scale in a multi-cloud world

Pankaj Sabharwal

bpb

Multi-Cloud Automation with Ansible

Automate, orchestrate, and scale in a multi-cloud world

Pankaj Sabharwal

bpb

Multi-Cloud Automation with Ansible

Automate, orchestrate, and scale in a multi-cloud world

Pankaj Sabharwal

www.bpbonline.com

Copyright © 2024 BPB Online

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor BPB Online or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

BPB Online has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, BPB Online cannot guarantee the accuracy of this information.

First published: 2024

Published by BPB Online
WeWork
119 Marylebone Road
London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55517-746

www.bpbonline.com

Dedicated to

My wife ***Dr. Geetika Sabharwal***
My sons ***Ayaan*** *and* ***Anay***

My beloved Parents:
Naresh Sabharwal
Reeta Sabharwal

About the Author

Pankaj Sabharwal stands as an embodiment of deep technological prowess and visionary leadership. With an illustrious tenure of over 11 years at IBM, Pankaj currently holds the esteemed position of Principal Solution Architect, primarily catering to the Financial and Federal sectors. His adeptness is best reflected in his monumental achievements, having modernized and migrated a staggering 200+ legacy applications to renowned cloud platforms, including AWS and IBM. His expertise is not confined to just cloud platforms; he's also reshaped numerous legacy systems, steering them towards a modern era encapsulated by containers and microservices.

Hailing from the historic city of New Delhi, India, Pankaj's academic pursuits led him across shores. He made his journey to the United States in 2006, where he earned his Master's in Computer Science from the **New Jersey Institute of Technology (NJIT)**. This was a progressive leap from his robust foundation in engineering, which he acquired from the esteemed Pune University in India.

In Pankaj, one sees not just an executive architect but a luminary, who seamlessly melds his rich heritage with a forward-looking, global perspective on technology's transformative powers.

About the Reviewers

- **Anil Murmu** is a solution architect with over a decade of IT enterprise expertise. **Anil** specializes in AI and Machine Learning, automation, and Infrastructure as Code (IaC), leveraging Python and Ansible to design, implement and automate independently operated multifaceted cloud-based (AWS, Azure, and GCP) solutions for clients across the globe. He loves exploring new technologies and helps enable people through appropriate knowledge bases to match and support IT requirements.

  He is currently working in HCL Technologies Ltd. and is part of the Automation Implementation program, encouraging clients from all over the globe to build, implement, adopt and adapt AL and ML enriched independently operated automation systems within their environment.

- **Amit Bhanushali** is a highly accomplished software quality assurance professional with over 22 years of experience in the IT industry. He earned his Master's in Business Data Analytics from West Virginia University in 2017. Based in West Virginia, USA, Mr. Bhanushali is a Senior IEEE Member and has significantly contributed to software testing research and practice.

  His expertise spans automation testing, performance testing, DevOps, and CI/CD implementation. He has also led testing efforts in complex cloud

environments. In addition to testing, Mr. Bhanushali has authored several articles exploring cutting-edge topics like artificial intelligence and machine learning. His published research demonstrates his thought leadership and impact on software quality engineering.

Mr. Bhanushali’s accomplishments have been recognized through prestigious appointments. He serves as a technical reviewer for the Elsevier journal, Books, and has served as a judge for hackathons and the Globee Awards. His contributions were further honored in 2023 when he received the International Achievers’ Award. With his sustained record of excellence in software development, testing, and research, Mr. Bhanushali continues to be an influential leader in his field.

Acknowledgement

First and foremost, I want to express my profound gratitude to my parents, Naresh and Reeta Sabharwal. A special mention goes to my mother, whose constant prayers and relentless support have been my guiding light. Every word in this book is a testament to the values and strength she instilled in me.

To my wife, Dr. Geetika Sabharwal, who, amidst her bustling schedule, has been my pillar of strength. Her encouragement, faith, and belief in me have made this journey smoother. Even when her plate was already overflowing, she gave me the space and time I needed to immerse myself in this project, a sacrifice I deeply appreciate.

My sons, Ayaan and Anay, have been the heartbeats behind this book. Their innocent inquiries about the book’s progress, their unending love, and their cheerful encouragement gave me the motivation to push through even the most challenging phases of writing. Their excitement has been infectious and rejuvenating.

I also wish to extend my heartfelt gratitude BPB Publications. The professionalism, patience, and support provided by BPB have been instrumental in bringing this book to fruition. Collaborating with BPB has truly been a rewarding experience.

To all mentioned above and many others who have been part of this journey, either directly or indirectly, I thank you.

Every page of this book is imbued with the positive energies you have all shared with me.

Preface

In the dynamic landscape of IT, the need to ensure scalability, repeatability, and consistency across infrastructures has driven the surge in automation tools. Ansible, with its declarative nature and agent-less architecture, emerges as a favorite. This book, structured meticulously over eleven chapters, offers an in-depth dive into leveraging Ansible for modern IT needs.

The book commences by grounding the reader in Ansible's core principles, laying a foundation upon which more intricate subjects are built. Our exploration is not restricted to a single environment; instead, we delve into Ansible's prowess in a multi-cloud world. Be it AWS's expansive service offerings, GCP's data analytics capabilities, or Azure's enterprise solutions, Ansible seamlessly weaves these platforms together, ensuring interoperability without compromising on security or efficiency.

Infrastructure automation is at the heart of Ansible. Through dedicated chapters, we dissect how Ansible interacts with servers, network devices, and even storage solutions. This ensures that from the physical layer up to the virtual, everything remains orchestrated and in harmony.

However, Ansible's magic is not just confined to infrastructures. We expand our horizon into application and platform automation, allowing developers and operations teams alike to ensure that applications are consistently deployed, scaled, and managed. In synergy with platforms like OpenShift and Kubernetes, Ansible takes container

orchestration to new heights, ensuring microservices and applications run smoothly, irrespective of the underlying complexities.

As computing transcends beyond centralized data centers, our discussion ventures into the realm of Edge computing. Here, we unearth how Ansible, with its lightweight footprint, becomes invaluable in managing and automating tasks closer to data sources, be it IoT devices or regional servers.

For organizations striving for centralized control, role-based access, and a visual dashboard, our chapters on Ansible Tower illuminate its pivotal role in enterprise-scale automation. From job scheduling to real-time job status monitoring, Tower's capabilities are dissected and illustrated.

In the later sections, the book takes a futuristic turn. We delve into the realm of AI and Machine Learning, exploring Ansible's potential role in managing AI infrastructures, orchestrating ML workflows, and even its interplay in generating AI. As machine learning models and AI applications become mainstream, ensuring their consistent deployment and scalability becomes paramount – a challenge Ansible is poised to address.

This book is not just theoretical; it is a blend of concepts, hands-on examples, and real-world use cases. it is crafted for both the novice trying to understand automation's basics and the expert aiming to push Ansible's boundaries.

To every reader picking up this book, you are about to embark on a journey that intertwines automation, cloud, containers, edge computing, and the frontier of AI. it is a journey I'm excited to guide you through. Let us begin.

Chapter 1: Ansible in Multi-Cloud Environment – This chapter navigates the challenges of a multi-cloud landscape, showcasing Ansible's strengths in addressing

manual deployments, environment inconsistencies, and other complexities. We will delve into the myriad benefits that Red Hat Ansible offers, from increased operational efficiency to its cloud-agnostic capabilities, emphasizing its transformative role in the future of cloud automation.

Chapter 2: Ansible Setup Across OS and Cloud – This chapter takes a hands-on approach, guiding readers through the installation of Ansible across diverse environments—from major cloud platforms like AWS, Google Cloud, and Azure to diverse operating systems such as MacOS and Windows. Whether you are setting up on a VM, a Docker container, or a desktop, this chapter ensures you are adeptly equipped to get Ansible up and running seamlessly.

Chapter 3: Writing Tasks, Plays, and Playbooks – This chapter delves deep into the foundational elements of Ansible. From understanding basic concepts such as control nodes, managed nodes, and inventories to crafting intricate playbooks and plays, we elucidate the importance of each component. We explore the organization and configuration of playbooks the dynamism of Ansible's inventories, and offer hands-on guidance with a real-life example—crafting a playbook to deploy an NGINX server. By the chapter's end, readers will have a robust grasp of Ansible's structure and operational anatomy.

Chapter 4: Infrastructure Automation Using Red Hat Ansible - In this chapter, we embark on a journey of infrastructure automation using Ansible across the three major cloud giants: AWS, GCP, and Azure. We will showcase the power and adaptability of Ansible as we automate infrastructure provisioning and management, demonstrating its capabilities in diverse cloud ecosystems and ensuring readers are well-prepared to harness Ansible's full potential, regardless of their cloud platform of choice.

Chapter 5: Network Automation Using Ansible – This chapter delves into the world of network automation with Ansible. From gathering critical network data and viewing system configurations to ensuring the safety of your settings through backups, we will guide you step by step. We also touch upon specific configuration tasks, such as setting host names and adjusting system settings, underscoring Ansible's prowess in streamlining and bolstering network management tasks..

Chapter 6: App Automation Using Ansible – Chapter six zooms in on Ansible's capabilities in the realm of application automation. We take a practical approach, walking you through the deployment processes on major platforms: from AWS, GCP, and Azure to the container-centric world of RedHat OpenShift. By the end, you will appreciate the versatility and power of Ansible in seamlessly deploying and managing applications across diverse infrastructures, culminating in a comprehensive conclusion that ties together the chapter's key insights.

Chapter 7: Security Automation Using Red Hat Ansible – In this pivotal chapter, we delve deep into the realm of security automation with Ansible, emphasizing its transformative impact on security operations. Ansible offers unparalleled efficiency, allowing swift implementation of security protocols across diverse cloud environments. The chapter illustrates its prowess in crucial security tasks—from patch management to intrusion detection. We also spotlight how Ansible fosters synergies with leading security tools, including CyberArk and QRadar. In essence, Ansible not only streamlines but elevates security automation, arming organizations against evolving threats with agility and finesse.

Chapter 8: Red Hat Ansible Automation for Edge Computing – This chapter spotlights Ansible's role in the

emerging domain of Edge Computing. We begin by exploring the broader spectrum of enterprise automation, segueing into the potent capabilities of the Ansible Automation Platform and the revolutionary concept of the Automation Mesh. The heart of the chapter showcases diverse industry use cases, highlighting Ansible's transformative impact in sectors like transportation, retail, telecommunications, and health care, among others. By the chapter's end, readers will have a panoramic view of Ansible's expansive reach and relevance in the dynamic world of Edge Computing.

Chapter 9: Red Hat Ansible for Kubernetes and OpenShift Clusters – Chapter nine delves into the synergistic relationship between Ansible and the container orchestration giants, Kubernetes and OpenShift. From utilizing Ansible for Kubernetes Operators to exploring its dedicated modules for the platform, the chapter showcases the efficiency of deploying Kubernetes objects using Ansible. Furthermore, it details the intricacies of managing Kubernetes/OpenShift clusters, emphasizing Day2Ops operations. The chapter culminates with a deep dive into deploying a DevSecOps pipeline, epitomizing the combined power of Ansible, Kubernetes, and OpenShift in today's cloud-native landscape.

Chapter 10: Using Ansible Automation Platform in Multi-Cloud – In this chapter, we venture into the multifaceted world of the Ansible Automation Platform, especially in multi-cloud environments. We commence by breaking down the core components of AAP, with a deep dive into the Automation Controller and its architecture. The journey continues with a hands-on guide to installing AAP via the OpenShift Operator on an OpenShift 4 cluster. The integration of AAP with essential tools such as GIT repositories and **Red Hat Advance Cluster Management** (**RHACM**) is explored, setting the stage for real-world use

cases. Highlighting GitOps implementation and efficient policy management, the chapter reveals the holistic power and potential of AAP in a multi-cloud landscape.

Chapter 11: Red Hat Ansible for Deep Learning – This chapter illuminates Ansible's pivotal role in the realm of deep learning. Deep learning frameworks, such as TensorFlow, PyTorch, and Keras, come with intricate software dependencies, making them daunting to set up manually. Enter Ansible: a solution to automate these configurations seamlessly. Readers will grasp how Ansible's playbooks, with their modular and customizable nature, simplify and expedite the setup of deep learning environments. We further emphasize Ansible's strength in fostering reproducibility across different machines, a critical factor in large-scale research and collaborative ventures. In essence, Ansible emerges not just as a tool but as a game-changer, empowering deep learning enthusiasts to channel their efforts into model development and research without the overhead of setup hassles.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/255a8c

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Multi-Cloud-Automation-with-Ansible**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at :

business@bpbonline.com for more details.

At **www.bpbonline.com**, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Table of Contents

CHAPTER 1
Ansible in Multi-Cloud Environment

Introduction

In the fast-paced realm of today's IT world, the landscape has shifted dramatically towards a multi-cloud approach. Organizations are no longer tethered to a single cloud provider. Instead, they operate enterprise applications across multiple cloud environments to leverage the unique benefits each one offers, be it cost-effectiveness, advanced features, or regional availability.

Modern-day mission-critical enterprise applications have truly transformed how businesses operate. These applications are characterized by their complexity, distributed nature, and scalability. Such qualities make them powerful tools, capable of driving efficiency and innovation. Yet, with great power comes an inherent set of challenges. Their intricate architecture and wide distribution can often make them challenging to manage and maintain.

These challenges manifest in various ways. For instance, when setting up a network across different clouds, one must consider the nuances and specific protocols of each provider.

Security policies, which are paramount in ensuring the safety of data and applications, might differ vastly from one cloud environment to another, necessitating careful crafting and regular updates. Furthermore, managing compute engines across different cloud platforms requires a deep understanding of each environment's strengths and limitations. Moreover, when issues arise, the clock starts ticking. The ability to swiftly detect and rectify problems becomes a crucial competency, as prolonged downtimes or data breaches can have dire consequences for businesses.

This is where Ansible steps in as the hero of the story. Ansible is not just a tool; it is a solution designed to address the multifaceted challenges posed by a multi-cloud ecosystem. With its automation capabilities, Ansible streamlines processes, ensuring that networks are set up efficiently, security policies are uniformly applied, and compute engines are optimally managed. Most importantly, when issues emerge, Ansible facilitates rapid detection and rectification, minimizing potential damages.

As we delve deeper into this chapter, we will unravel how Ansible seamlessly integrates into the multi-cloud paradigm, acting as a linchpin that holds together the diverse and dynamic components of modern enterprise applications. Through real-world examples and expert insights, readers will gain a profound understanding of Ansible's pivotal role in navigating the challenges and opportunities of the multi-cloud era.

Structure

In this chapter, we will go through the following topics:

- Challenges in multi-cloud
  - Manual and error-prone deployments
  - Complex deployments

    - Environment inconsistencies
- Business value Red Hat Ansible brings
    - Faster business agility
    - Increased efficiency of IT operations
    - Same technology across the board
    - Simple and agentless
    - Step by step reporting
    - Cloud agnostic
- Why Red Hat Ansible is the future for cloud automation
    - Provisioning
    - Configuration management
    - Application deployment
    - Continuous delivery
    - Security automation
    - Cloud automation
    - Orchestration
    - End point protection

Objectives

In this chapter, we aim to delve deep into the multi-faceted challenges of a multi-cloud environment, such as the hurdles posed by manual and error-prone deployments, the intricacies of handling complex deployments, and grappling with environmental inconsistencies. As we navigate these challenges, we will also uncover the substantial business value that Red Hat Ansible brings to the table. Through our exploration, readers will gain insights into how Ansible fosters faster business agility, amplifies the efficiency of IT

operations, and offers the consistency of a single technology across diverse platforms. Its simplicity, combined with an agentless architecture, detailed step-by-step reporting, and cloud-agnostic features, truly accentuates its significance in the IT landscape. Finally, we will extrapolate why Ansible is not just a current solution but the future of cloud automation. We will touch upon its prowess in provisioning, configuration management, application deployment, and continuous delivery. Additionally, we will highlight its capabilities in ensuring security, orchestrating cloud processes, and safeguarding endpoint protection. By the end of this chapter, readers will be well-equipped with a comprehensive understanding of Ansible's role in shaping the future of multi-cloud operations.

Challenges in multi-cloud

Let us discuss the common problems that exist in managing enterprise applications in multi-cloud.

Manual and error-prone deployments

Deployments are done manually every other month and if everything goes as planned, it can take about eight hours to complete:

- Deployments are done by multiple global teams who follow the written instructions in cookbooks to perform different tasks.
- Different parts and bits of deployments are managed by different people based on his/her area of expertise using his/her custom scripts.
- Each deployment exercise is unique and brings its own complexity and challenges.

Complex deployments

As mentioned above, each deployment is unique. Some deployments become complex as you are required to follow a particular sequence in order to make deployment successful. A common example is database change, where you need to shut down the application before making a database update and before all that, you need to deploy a front-page, informing users that the site is under maintenance:

- If you deploy the database to change without shutting down the application, then the deployment will fail.
- Human interaction is also another factor which makes some deployments more complex. In some deployments, human interaction is required due to the lack of the right automation tools.
- Traditional manual script-based deployments procedures are not able to cope with applications like microservices, which are more distributed and scalable.

Environment inconsistencies

It is very rare that you will find an organization where non-prod environments are running with exact hardware and network configurations as prod environments. Normally, they will be in different network zones or will have different compute power assigned to them. In some cases, you will find inconsistencies in software as well due to reasons like different versions of software running or the way software was deployed and configured on each environment. These inconsistencies will create issues in prod, which can potentially lead to significant expenses and discomfort.

Business values Red Hat Ansible brings

Ansible can be the core of the solution you will build to solve all the problems described above in multi-cloud architecture and it is very human readable.

Faster business agility

There are a lot of tedious, manual and repeatable tasks which can be automated using Ansible, hence improving the productivity of teams. This way, businesses are able to meet and exceed their goals.

Increased efficiency of IT operations

Deployments and operational tasks which used to take multiple days to complete can be done in minutes with much higher success rates and less downtime. Since the same Ansible code can be used in all environments without change, it brings a lot more consistency between non-prod and prod environments and fewer errors.

Same technology across the board

Ansible can be used in deploying infrastructure, platform, and software. This means, you do not need a different software or need to learn new skills to manage each domain. Hence, with Ansible, you can install cloud infrastructure like **VPCs** and **EC2** on **AWS** and deploy any platform like **OpenShift** or **Kubernetes** and in the same fashion you can deploy any software like **Apache** or **Nginx** or even a database like **Oracle** or **Db2**.

Simple and agentless

Ansible is very human-readable, you do not need to have prior skills to learn Ansible. There is no special agent that needs to be deployed on each node, but Ansible just uses simple **OpenSSH** and **WinRM** to access the target node and deploy changes.

Step-by-step reporting

Ansible encourages administrator to name every step in their script, which helps in determining what each task is doing. It makes troubleshooting easier if any step fails, so administrator have a pinpoint view of where to look to fix the issue.

Cloud agnostic

Mostly, the same Ansible code can be used on any cloud provider of your choice. There are cloud-specific modules in Ansible for sure, but there is a very thin abstraction layer. Switching from one cloud provider to another can be done with very minimum effort.

Ansible is the future for cloud automation

When we talk about cloud automation, there are multiple use cases we are talking about. We are talking cloud provisioning, platform provisioning and management, configuration management, security and compliance management and much more. We are living in an age of microservices and cloud-native, where automation is not good to have but is a basic necessity. Ansible is a tool which can automate anything, as we have mentioned before.

In a traditional non-cloud environment, infrastructure provisioning is a separate exercise for configuration management. There is a dedicated team who owns infrastructure provisioning. But with Ansible, you can automate infrastructure provisioning too and since Ansible can be used for configuration management exercises, day one and day two operations can be performed in a more simplified way using a common tool.

The following are some common cloud-related use cases that Ansible can support easily and make the process much more

efficient.

Provisioning

Whether you are on traditional bare metal hardware or modern serverless or function as a service model, you need the underlying infrastructure and hence, it is the first step if you want to automate your application's operational life cycle. Ansible can be used to provision infrastructure on any cloud service provider, hypervisors, network devices and bare metals. Provisioning infrastructure can be easily followed by the next steps, which can be configuration management, or even setting up a new modern platform layer like Kubernetes or OpenShift Container Platform if you plan to run containerized workloads. The following *Figure 1.1*, shows major cloud service providers where Ansible is commonly used:

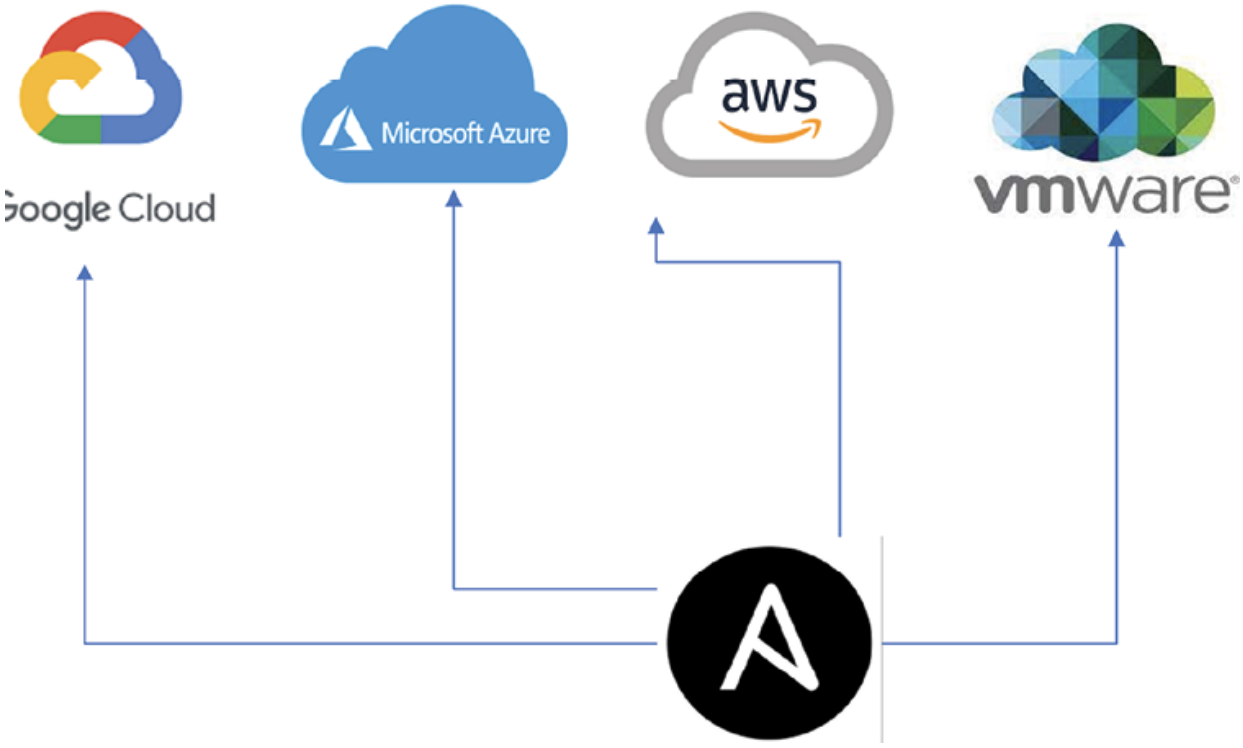

Figure 1.1: *Ansible is cloud agnostic*

Configuration management

Ansible configurations stand out for their simplicity and readability. This clarity ensures that even individuals unfamiliar with Ansible can readily interpret its code. Additionally, due to its straightforward nature, machines can parse Ansible code with ease, eliminating the need for any compilation. One of the major advantages of using Ansible for configuration management is its ability to deploy configurations to target environments, be it in the cloud or traditional on-premises systems, without the need for installing any agents on the target side. Ansible leverages OPENSSH to access the target environment and deploy the specified configurations. *Figure 1.2* illustrates how Ansible Playbook, positioned outside the target servers, manages configurations without the necessity of Ansible agents on the target server:

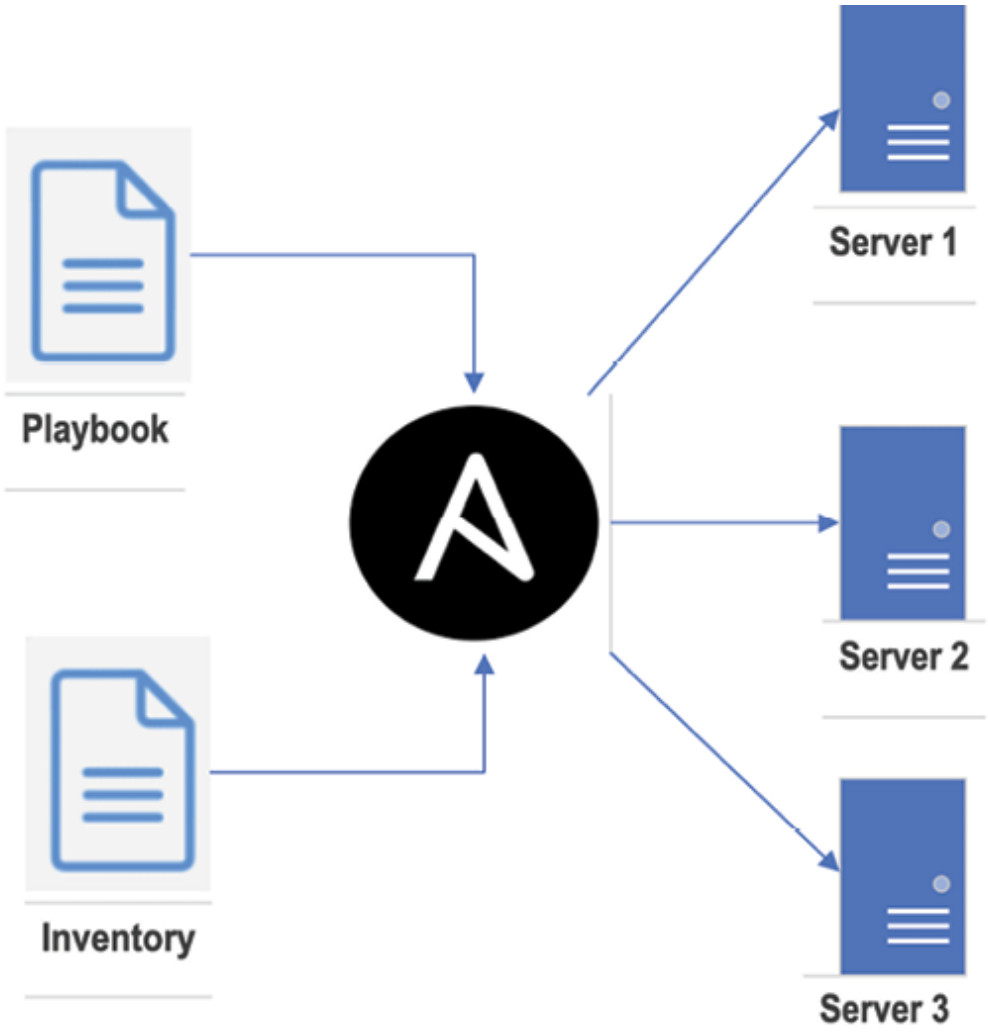

Figure 1.2: *Ansible deploying a patch simultaneously to 3 target nodes, in inventory*

Application deployment

With Ansible you can deploy multi-tier applications efficiently, consistently and securely. In Ansible, you can write the playbooks which is a collection of multiple tasks. These tasks are nothing but where you define the desired state of the target system. These playbooks are repeatable and reliable and hence make the application installations and upgrades easy and efficient. *Figure 1.3* shows a very basic structure of the Ansible **Playbook** and **Tasks** and **Roles** inside it:

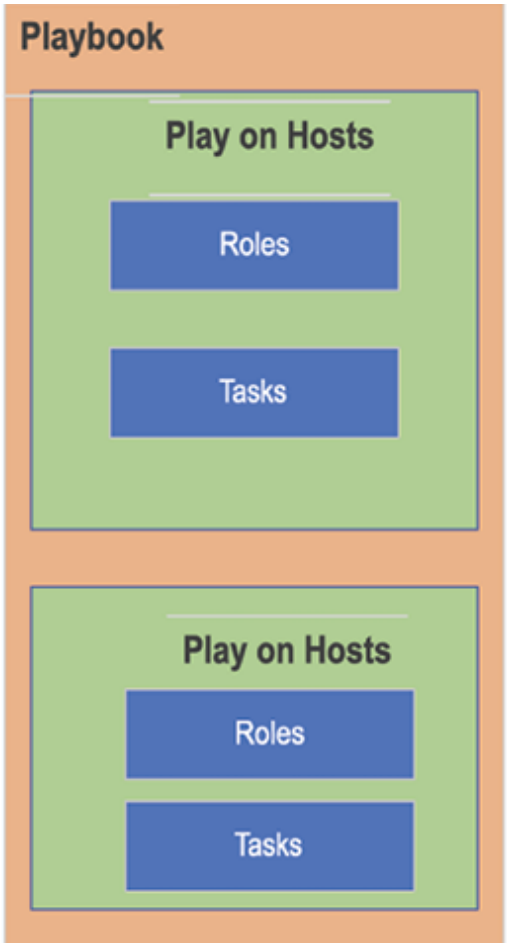

Figure 1.3: *How simple Ansible playbook looks like*

Continuous delivery

Continuous delivery (**CD**) is the right casing is simply frequently delivery updates to your applications without any impact or minimal impact on the end user. Ansible provides true multi-tier, multi-step orchestration. Ansible's push-based architecture allows very fine-grained control over operations, able to orchestrate the configuration of servers in batches, all while working with load balancers, monitoring systems, and cloud or web services. Slicing thousands of servers into manageable groups and updating them 100 at a time is incredibly simple, and can be done in a half page of automation content. In the following *Figure 1.4*, we can see

how easily Ansible can be integrated into the existing CI/CD pipeline:

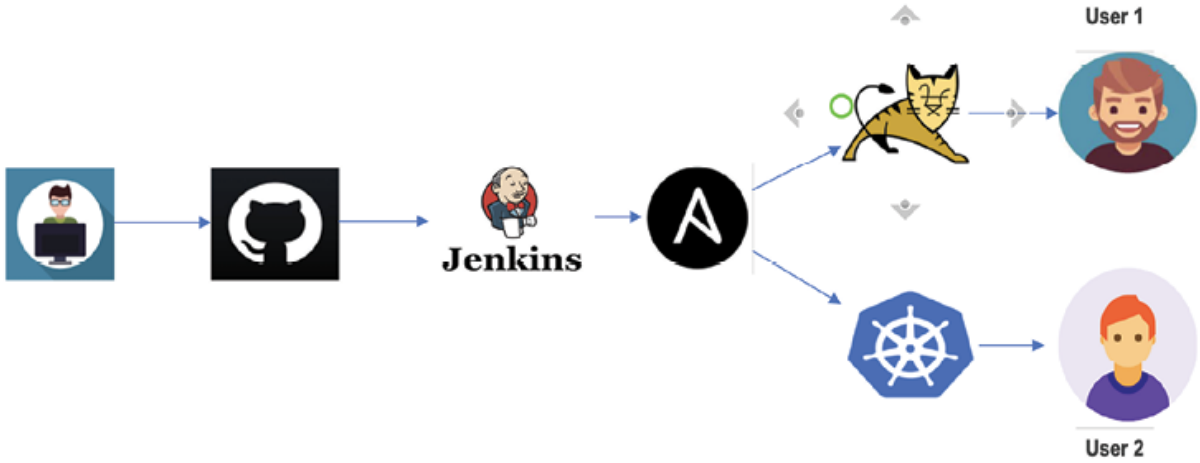

Figure 1.4: *Ansible code can be checked in to SCM just like any other application code*

Security automation

Ansible is a powerful tool for automating security tasks across multi-cloud environments. Its agentless nature and use of SSH for communication make it highly suitable for configuring systems in a secure manner, while its idempotent design ensures that desired system states are maintained over time.

One of Ansible's greatest strengths is its flexibility. This allows it to be easily integrated with various cloud providers (like AWS, Azure, GCP) using dedicated modules and collections. Through these, security configurations can be managed consistently across multiple environments, from setting up firewall rules and network configurations to managing access controls and permissions.

Further, Ansible can automate the deployment and configuration of security-focused applications like intrusion detection systems, log collectors, or security scanners. It

also has modules for managing SSL certificates, an important part of secure communications.

Another benefit is that Ansible allows for the implementation of security as code, where security policies can be codified in Ansible playbooks. This improves transparency, consistency, and repeatability while also allowing for version control and peer reviews of security changes.

Moreover, Ansible can be used for continuous security compliance checks, ensuring that infrastructure is compliant with security standards like CIS, NIST, or company-specific guidelines. By automating these checks, Ansible can help detect and rectify security drifts rapidly, which is crucial in a multi-cloud environment. *Figure 1.5* shows how **Jenkins** is deploying Ansible code for implementing a firewall change:

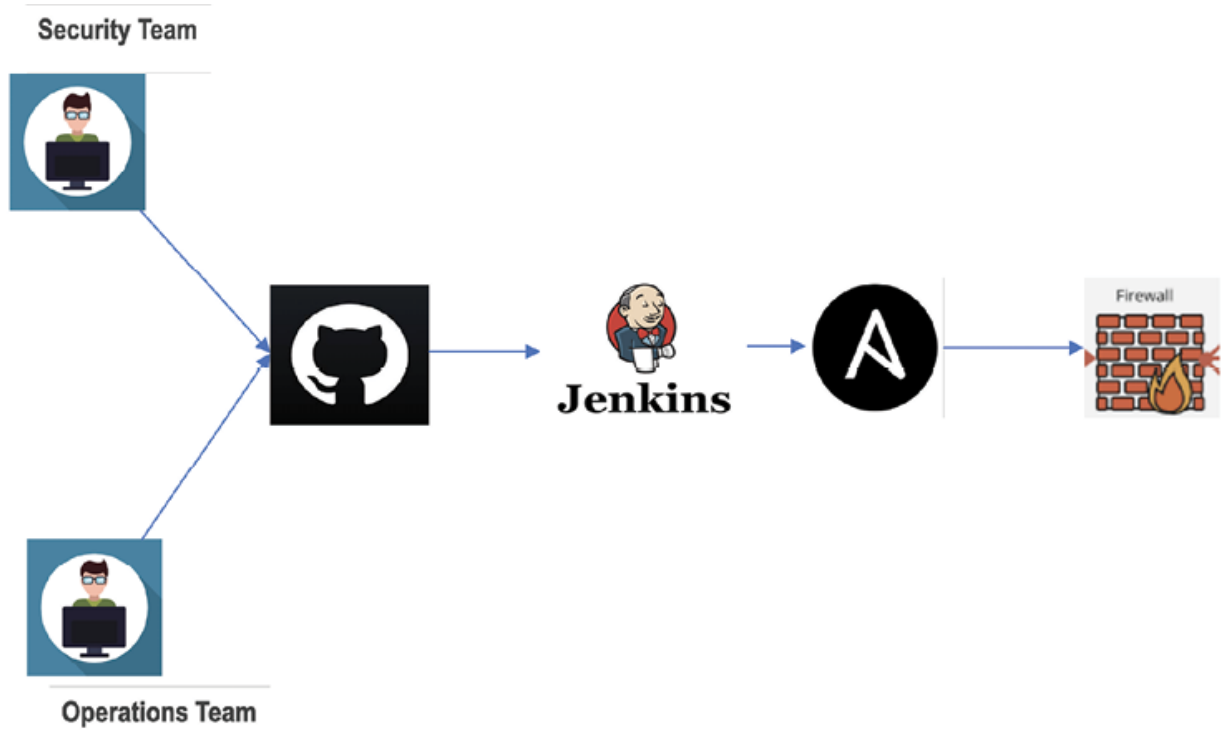

Figure 1.5: *DevOps team and Security team collaborating to deploy a security fix using Ansible*

Cloud automation

As we understand that, clouds are not just any servers but much more than that. Ansible provides multiple modules for all cloud service providers, like AWS, GCP, Azure, and IBM, to support tasks like operating system provisioning, infrastructure provisioning, setting up virtual private networks or VLANS, setting up access management, configuring load balancers, setting up a containerized platform like Kubernetes or OpenShift Container Platform, setting and defining auto-scaling policies and much more. We can see in *Figure 1.6*, that Ansible can deploy changes to different and all layers of your solution in the cloud:

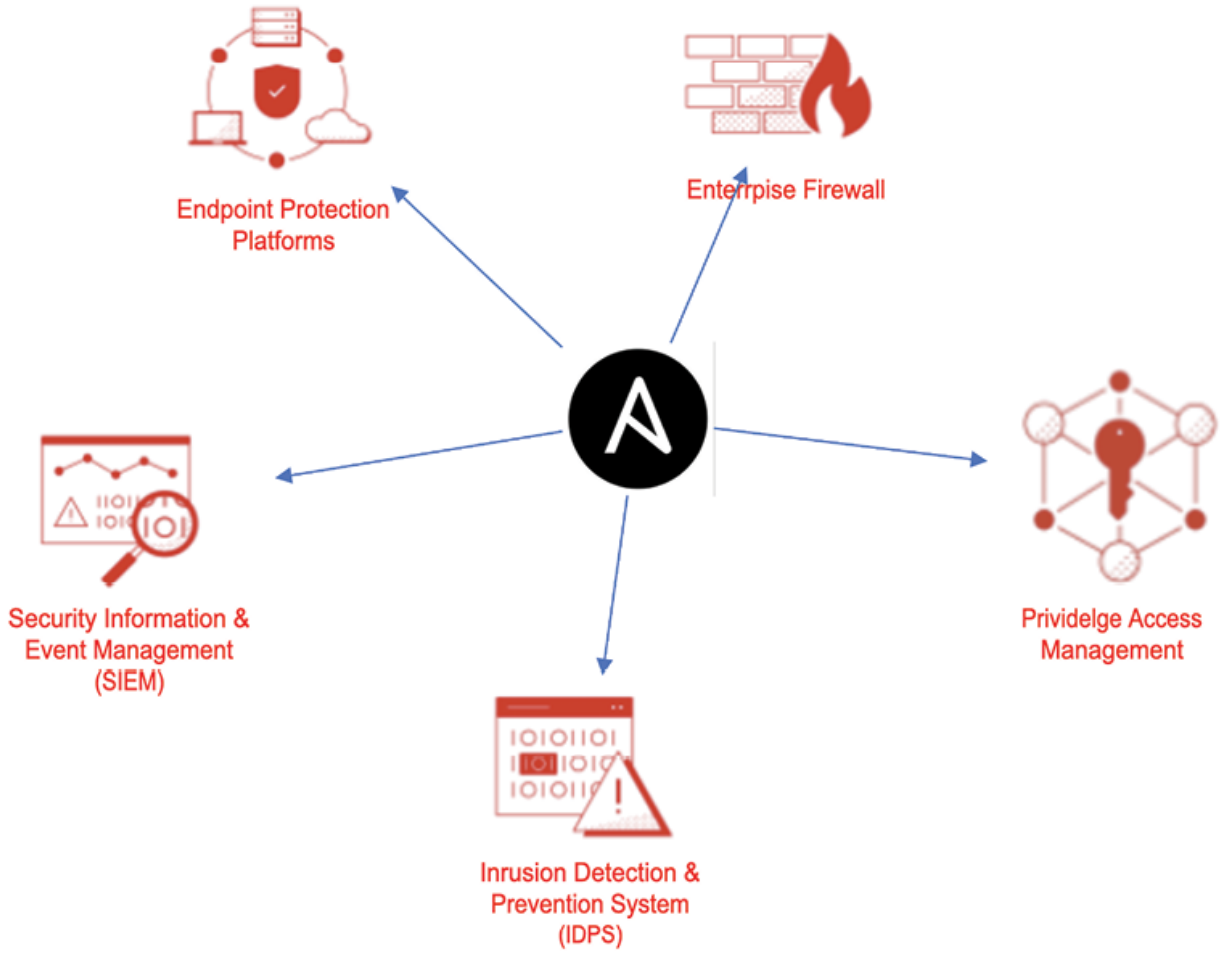

Figure 1.6: *Ansible can automate at all levels on any cloud provider platforms*

Orchestration

Deploying and managing multi-tier applications across multiple target systems poses significant challenges. An application typically encompasses several components such as frontend, backend, monitoring, network, storage, among others. To ensure optimal functionality, it is essential for these components to integrate seamlessly and in a specific sequence. Ansible simplifies this orchestration process. With Ansible, one can explicitly specify the sequence for deploying each component of the application stack and delineate the dependencies each component has on the others. Following yaml code shows a simple Ansible Playbook. This playbook deploys a **`Python Flask application`** as a **`backend service`** and uses **`httpd`** as the **`frontend server`**:

```
---
# Configuration for backend server
- hosts: backend
  become: yes  # Elevate privileges for the tasks
  tasks:
    - name: Install python3 and pip  # Install necessary Python packages
      yum:
        name:
          - python3
          - python3-pip
        state: present

    - name: Install Flask  # Install Flask using pip3
```

```
      pip:
        name: Flask
        executable: pip3

    - name: Copy flask app to server  # Transfer
Flask app to target server
      copy:
        src: /path/to/your/Flask/app.py
        dest: /app/app.py

    - name: Run Flask application  # Execute the
Flask app on target server
      shell: nohup python3 /app/app.py >
/app/flask.log 2>&1 &
      args:
        executable: /bin/bash
      become_user: root

# Configuration for frontend server
- hosts: frontend
  become: yes
  tasks:
    - name: Install httpd server  # Install httpd
server for hosting frontend
      yum:
```

```
        name: httpd
        state: present

    - name: Ensure httpd service is running  # Ensure httpd is active and running
      service:
        name: httpd
        state: started
        enabled: yes

    - name: Copy frontend files to server  # Transfer frontend assets to target server
      copy:
        src: /path/to/your/frontendFiles
        dest: /var/www/html

    - name: Open port 80 in firewall  # Allow traffic on port 80 for httpd server
      firewalld:
        port: 80/tcp
        permanent: yes
        state: enabled
        immediate: yes
        zone: public
...
```

Following is what this playbook does:

- On the **backend hosts**, it installs **Python** and **pip**, then installs **Flask** using **pip**.
- It then copies a **flask application** from your local machine to the **server**. You would need to replace **/path/to/your/flask/app.py** with the actual path of your **flask application** file.
- It runs the **flask application**.
- On the **frontend hosts**, it installs the **httpd** server, ensures it is running, and copies your **frontend files** from your local machine to the **httpd** document **root** on the server.
- It then opens **port 80** on the **frontend server**.

Endpoint protection

Endpoint protection (**EPP**) detects, investigates and remediates the malicious activities on the end point systems as these are the most exposed, hence the most vulnerable components of IT infrastructures. Ansible easily integrates with EPP tools and provides event-driven detection, quarantining and remediation. *Figure 1.7* shows different use cases of endpoint protection where Ansible can make the process more efficient:

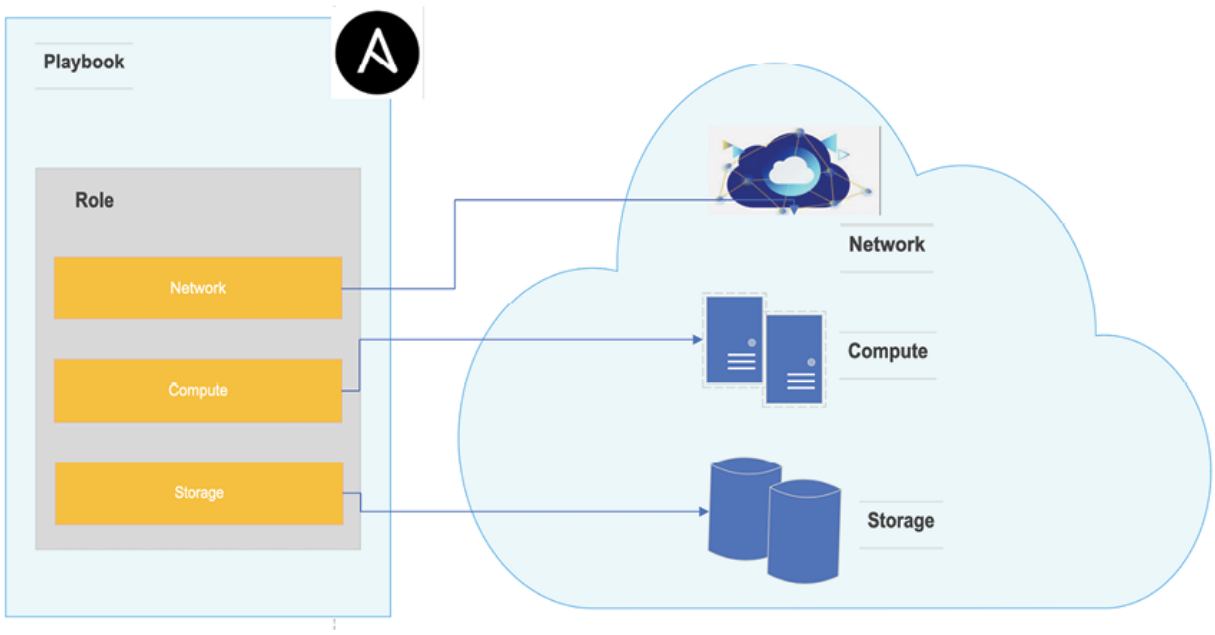

Figure 1.7: *Ansible's role in end point protection*

Conclusion

In conclusion, we have seen challenges faced by multi-cloud implementations and how easily and efficiently Ansible can be used to solve those challenges regardless of what cloud platform is or at what component the challenge is.

In the coming chapters, we will deep dive into each use case, understand the challenge in detail and understand how Ansible solves the challenge in the most effective way.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 2
Ansible Setup Across OS and Cloud

Introduction

Ansible's library of cloud support modules makes it easy to provision instances, networks, and complete cloud infrastructure on the cloud provider of your choice. The same simple Playbook language you use for application deployment and on-prem virtualization automation. It also provisions your infrastructure, and applies the correct configuration to it. Ansible ensures your cloud deployments work seamlessly across public, private, or hybrid cloud as easily as you can build a single system.

Structure

In this chapter, we will go through the following topics:

- Installing Ansible on AWS's EC2
  - Creating and setting up EC2 instance on AWS
  - Options for installing Ansible on AWS
    - Installing Ansible using amazon-linux-extra option
    - Installing Ansible using the EPEL repository option
    - Verification of Ansible install
- Installing Ansible on Google cloud provider
  - Creating VM instance on Google cloud provider
  - Installing Ansible on Google cloud provider using **pip** package manager option
  - Verification of Ansible install
- Installing Ansible on Microsoft Azure
  - Creating VM on Azure cloud using command line option
  - Installing Ansible on Azure cloud using shell script
  - Verification of Ansible install
- Installing Ansible on a Docker container
  - Creating Docker file
  - Creating Docker image using Docker file
  - Running a Docker container with Ansible
  - Verification of Ansible installation inside the container
- Installing Ansible on MacOS

    - Setting up Homebrew on your MacOS
    - Installing and setting up Ansible on MacOs using new
    - Verification of Ansible on MacOs
- Installing Ansible on Windows OS
    - Installing and setting up Ansible on Windows OS using Cygwin
    - Verification of Ansible on Windows OS
- Installing Ansible on other Linux distros
    - Installing Ansible on Fedora
    - Installing Ansible on Ubuntu
    - Installing Ansible on CentOS

Objectives

The objective of this chapter is to learn how Ansible can be installed and setup on various cloud providers like AWS, GCP, Azure. We will learn how to install Ansible on these cloud providers in different ways. Each cloud provider can have its own specific way to install Ansible. There can be a generic way to install Ansible on all cloud providers. So, we will explore these options and see how this can be accomplished manually and in a scripted way.

In addition to cloud providers, we will also see and learn how you can have containerized Ansible instance available in seconds. For this, we will build a Docker image with Ansible installed and will run a container out of it.

Installing Ansible on Amazon Web Services EC2

AWS, as we know, is one of the leading cloud service providers in the market today. In the next section, we will see how we can install Ansible on AWS compute instances.

Ansible can be used with AWS on any layer starting from provisioning infrastructure on AWS like creating VPCs, setting or managing platforms on AWS like setting EKS, or even installing or managing your own software applications on AWS. Let us see, how can we setup Ansible on AWS.

Creating EC2 instance

In this section, we will install Ansible on the AWS EC2 instance. First, we need to create the EC2 instance from the AWS management console. We will use Amazon Linux as the operating system on this EC2 instance. These steps are shown here, you need to have an AWS account for this and if you do not have an AWS account then there is an option to create a free account which is valid for 12 months. Learn more about it here **AWS Free Tier Account**.

1. Login to the AWS management console **https://us-east-1.console.aws.amazon.com/ec2/**.
2. On the EC2 dashboard you will see if you have any existing EC2 instances running.
3. Click **Launch Instances** in order to create a new EC2 instance for installing and setting up Ansible:

   Once you are in the **Launch an Instance** screen you can specify the **VM name**, **Operating System**, **Instance type** and other details. Here we are using Amazon

Linux as the operating system and **`t2.micro`** as the instance type. Once you are done, just click **Launch Instance,** and your instance will be ready in a couple of minutes.

Figure 2.1 shows a new instance **Ansible-master** we created for installing Ansible:

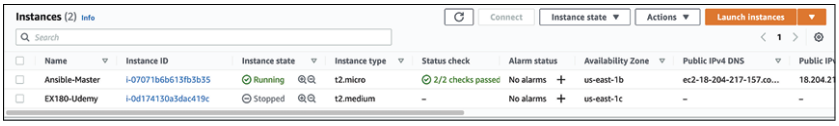

Figure 2.1: *AWS EC2 Dashboard showing the new instance*

SSH to EC2 instance:

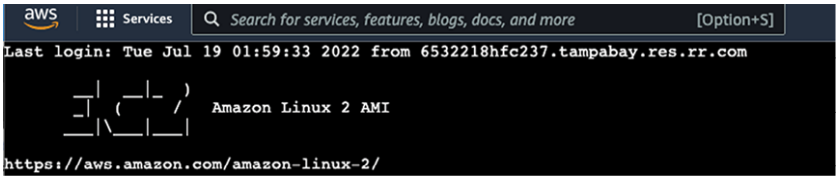

Figure 2.2: *AWS terminal once you ssh to EC2 InstanceScript to install Ansible using the amazon-linux-extras option*

On EC2 there are two options to setup Ansible:

1. Option 1 is using the Amazon **`Extra`** library code. Following is the code for installing Ansible using this option:

```
#!/bin/bash

# Update all packages
echo "Updating all packages..."
sudo yum update -y

# Install Ansible 2
echo "Installing Ansible 2..."
sudo amazon-linux-extras install ansible2 -y

# Display Ansible version
echo "Ansible version:"
ansible --version

echo "Installation and verification complete!"
```

   i. Save the code in the file **`ansible-amazon-extra-code.sh`**.
   ii. Make file executable by running **`chmod a+x ansible-amazon-extra-code.sh`**.
   iii. Execute the file by running **`./ansible-amazon-extra-code.sh`** and part of the output is shown here:

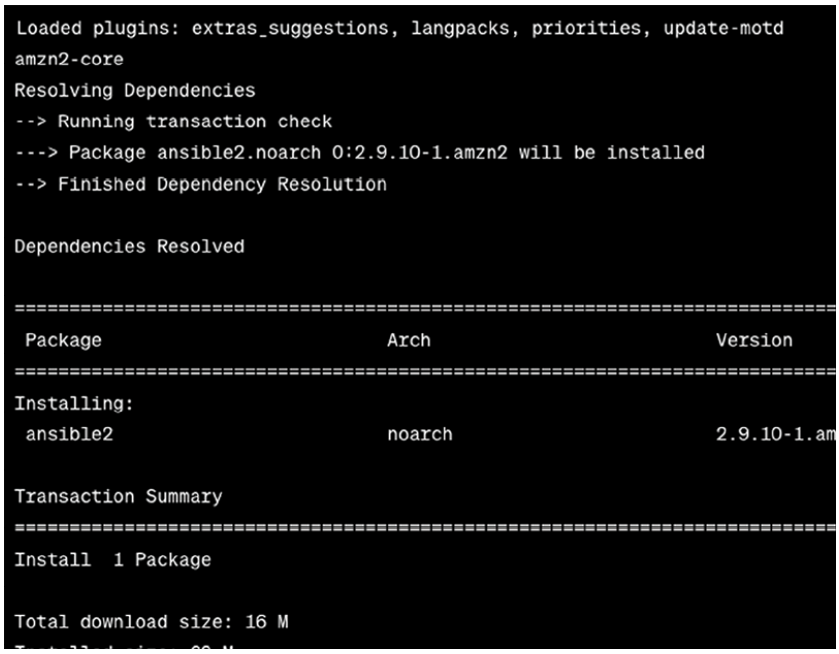

Figure 2.3: *Part of yum output showing linux packages getting installed*

iv. Once the previous command is completed you can run **`ansible –version`** to see if Ansible is installed or not. The version of installed Ansible is shown as follows:

```
ec2-user@ip-172-31-95-93 ~]$ ansible --version
ansible 2.9.23
  config file = /etc/ansible/ansible.cfg
  configured module search path = [u'/home/ec2-user/.ansible/plugins/mod
  ansible python module location = /usr/lib/python2.7/site-packages/ansi
  executable location = /usr/bin/ansible
  python version = 2.7.18 (default, May 25 2022, 14:30:51) [GCC 7.3.1 20
       ec2-user@ip-172-31-95-93 ~]$
```

v. Or you can run **`rpm -qa | grep ansible`** to verify the install and version **`ansible`**:

```
ec2-user@ip-172-31-95-93 ~]$ rpm -qa | grep ansible
ansible-2.9.23-1.amzn2.noarch
      ec2-user@ip-172-31-95-93 ~]$
```

2. Option 2 is to install Ansible using EPEL repositories: EPEL repository is an additional package repository which provides access to packages which are for commonly used softwares:

```
#!/bin/bash
sudo amazon-linux-extras install epel -y
sudo yum repolist
sudo yum-config-manager --enable epel
sudo amazon-linux-extras disable ansible2
```

```
sudo yum --enablerepo epel install ansible

      ansible --version
```

i. Similar to option 1 you can save this **`code in file ansible-epel-repo-code.sh`** and execute this file after making it executable.

ii. So, we have seen how to install Ansible on an AWS EC2 instance using Amazon **`Extra`** library and EPEL repositories options. In the next section, let us see how Ansible can be installed on Google cloud provider VM instance.

Installing Ansible on Google cloud provider VM instance

Just like AWS, you can create a Google Cloud Platform account easily, in case you do not have an existing account. There is an option to create a free account valid for 90 days too. Learn more about the GCP free account here **https://cloud.google.com/free/docs/gcp-free-tier**. On GCP too just like AWS one can use Ansible at any layer. The most common examples are using Ansible for provisioning instances on GCP and setting up autoscaling, setting up custom networks on GCP, and setting and managing storage on GCP.

Let us see how to install Ansible on GCP:

1. On the GCP console, you need to go to your project and select **VM Instances,** as shown in *Figure 2.4*:

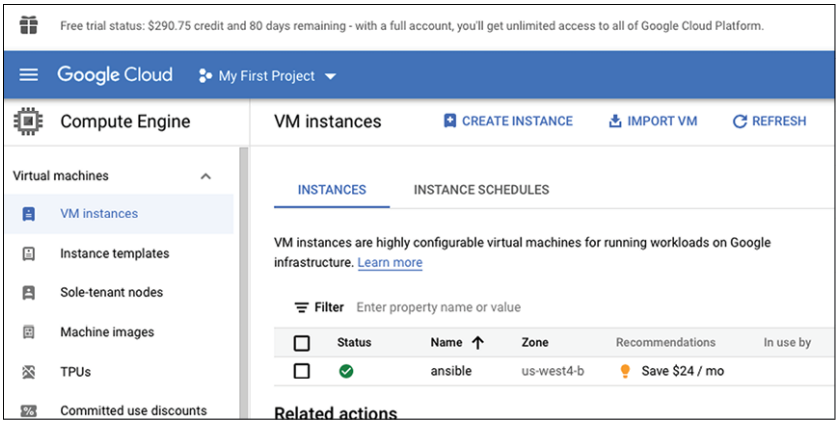

Figure 2.4: *GCP Dashboard showing VM instances*

2. Once you are there you click **CREATE INSTANCE** to create VM for Ansible Control Node as shown in *Figure 2.5*. You can see that we have created the VM called **ansible** using a free tier account. We have installed Red Hat Enterprise Linux as the operating system.
3. **SSH** to VM using any one of the options, as shown in *Figure 2.5*. We are using the option **Open in browser window** option here:

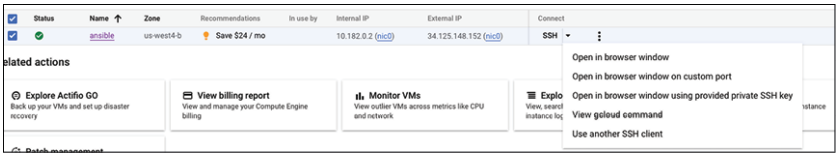

Figure 2.5: *GCP Dashboard showing SSH options*

4. Install **`pip`** using **`yum`**

So, here we can see show how we can install Ansible on a Linux VM using the **pip** option. **pip** is a cross-platform package manager for installing and managing python packages. So first we install **pip** using **yum**. The command is **yum install pip -y** and once you run you will see the **Package** shown as follows, getting installed on your VM:

```
Package                          Arch         Version               Repositc

================================================================================

Installing:

pipelight-selinux                noarch       0.1.0-2.el7           epel

Transaction Summary

================================================================================
```

5. Install **python3** using **yum**. Once the **pip** is installed, we will install **python3** using **yum**. The command **is yum install python3 -y** and you will see packages getting installed on your system, *Below we can see the output showing* the packages:

```
================================================================================

Package                  Arch       Version                Repository

================================================================================

Installing:

python3                  x86_64     3.6.8-18.el7           updates

Installing for dependencies:

libtirpc                 x86_64     0.2.4-0.16.el7         base

python3-libs             x86_64     3.6.8-18.el7           updates

python3-pip              noarch     9.0.3-8.el7            base

python3-setuptools       noarch     39.2.0-10.el7          base

Transaction Summary

================================================================================
```

6. Install Ansible and run command **python3 -m pip install –user ansible** as below to install Ansible:

```
root@ansible ~]# python3 -m pip install --user ansible

Collecting ansible

  Using cached ansible-4.10.0.tar.gz (36.8 MB)

Collecting ansible-core==2.11.7

  Using cached ansible-core-2.11.7.tar.gz (7.1 MB)

  Preparing metadata (setup.py) ... done

Requirement already satisfied: jinja2 in /usr/local/lib/python3.6/site-pac

Requirement already satisfied: PyYAML in /usr/local/lib/python3.6/site-pac

Collecting cryptography

  Downloading cryptography-37.0.1-cp36-abi3-manylinux2014_x86_64.whl (4.2
```

```
    4.2 MB 7.7 MB/s
Collecting packaging
  Using cached packaging-21.3-py3-none-any.whl (40 KB)
Collecting resolvelib<0.6.0,>=0.5.3
  Downloading resolvelib-0.5.4-py3-none-any.whl (12 KB)
...
```

7. Once the install finishes you can verify the Ansible installation and verify the installed version. Once Ansible is installed then verify the Ansible version using the **`ansible –version`**, as shown below:

```
[root@ansible ~]# ansible --version
ansible 2.9.27
  config file = /etc/ansible/ansible.cfg
  configured module search path = [u'/root/.ansible/plugins/modules', u'/u
  ansible python module location = /usr/lib/python2.7/site-packages/ansibl
  executable location = /bin/ansible
  python version = 2.7.5 (default, Nov 16 2020, 22:23:17) [GCC 4.8.5 20150
[root@ansible ~]#
```

Here, in this section, we have installed Ansible on a Linux virtual machine on the Google Cloud Platform. Here, if you notice, we have performed all manual steps to show how Ansible can be installed on Linux OS regardless of any cloud provider. These steps are pretty standard and are applicable to any cloud provider.

In the next section, we will install Ansible on the Microsoft Azure cloud.

Installing Ansible on Microsoft Azure VM

In this section, we will talk about installing Ansible on Microsoft Azure VM.

Creating Microsoft Azure VM

Just like AWS and GCP you need account for Azure cloud too. There are options to have a free tier account on Azure for 30 days. To learn more about free account check **https://azure.microsoft.com/en-us/free/**. There are out of the box Ansible modules for Azure like virtual machines, virtual networks, resource groups, storage, templated deployments and much more. Let us see the steps to install and setup Ansible on Azure.

1. Once you have an Azure portal login, you can access the section **Virtual machines** under **Dashboard**.
2. In order to create a VM on Azure, you will need to make sure you have Azure CLI installed on your system. You can create the VM from portal by clicking the **Create** option under **Virtual machines** as shown in *Figure 2.6*:

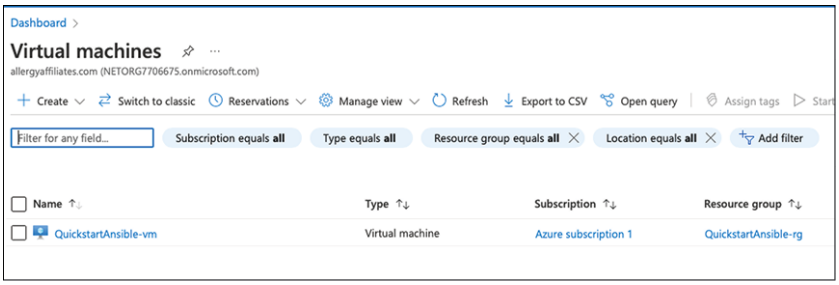

Figure 2.6: *Virtual machines on Azure portal*

3. There is also the option to **Create/Add Virtual machine** using the **Command Line** option and that is the option we are using here for Azure cloud. If you do not have Azure CLI installed, you can directly go to **https://portal.azure.com/#cloudshell/** and login with your credentials.

   Once you are logged in, you will shell the terminal as shown in *Figure 2.7*:

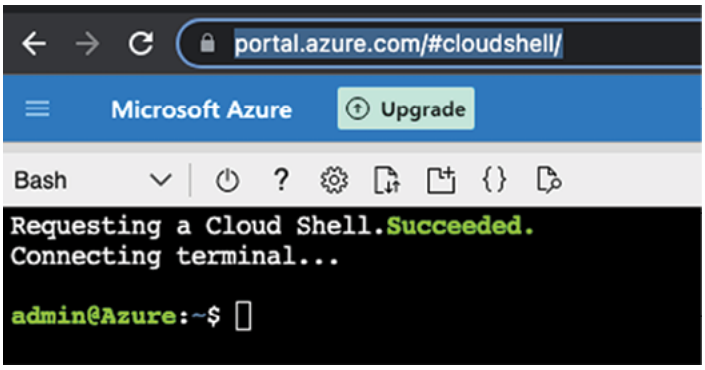

Figure 2.7: *Azure shell*

4. Once you are logged in, you need to extract the **`subscriptionID`** for your account. Run **`az account get-access-token`** from the **`cloudshell`**. This is shown as follows:

```
{
  "accessToken": "eyJ0eXAiOiJKV1QiLC...REDACTED...",
  "expiresOn": "2022-07-23T08:39:45.1100000+00:00",
  "subscription": "22b7e1a6-06e1-4a93-8f0a-192b34edc9e8",
  "tenant": "1d61eea8-8702-d4f1-8a0d-a6cbf080671",
  "tokenType": "Bearer"
}
```

5. Create service principle after having **`subscriptionID`**. Run command **`az ad sp create-for-rbac --name ansible --role Contributor --scopes /subscriptions/<subscription_id>`**, this is shown as follows:

```
admin@azure:~$ az ad app create-for-rbac --name ansible --role Contributor
Found an existing application instance: (id) cfda5f42-53d2-45de-98e3c929b2
Creating 'Contributor' role assignment under scope '/subscriptions/22b7e1a
{
  "appId": "REDACTED",
```

```
  "displayName": "ansible",
  "password": "REDACTED",
  "tenant": "REDACTED"
}
      admin@azure:~$
```

6. Create resource group using **`az group create --name QuickstartAnsible-rg --location eastus`**. Here you specify the location as per your choice where you want to run the VM. This is shown below:

```
admin@azure:~$ az group create --name QuickstartAnsible-rg --location east
{
  "id": "/subscriptions/2eb7efa6-06f8-49a3-8f0a-192b34edc9e8/resourceGroup
  "location": "eastus",
  "managedBy": null,
  "name": "QuickstartAnsible-rg",
  "properties": {
    "provisioningState": "Succeeded"
  },
  "tags": null,
  "type": "Microsoft.Resources/resourceGroups"
}
```

7. Get the public IP for the VM using **`az vm show -d -g QuickstartAnsible-rg -n QuickstartAnsible-vm --query publicIps -o tsv`**, as follows:

```
admin@azure:~$ az vm show -d -g QuickstartAnsible-rg -n QuickstartAnsible-
20.231.54.165
            admin@azure:~$
```

8. SSH to the VM using the **`ssh`** command as shown:

```
admin@azure:~$ ssh azureuser@20.231.54.165
The authenticity of host '20.231.54.165 (20.231.54.165)' can't be establis
ECDSA key fingerprint is SHA256:26OpINEFkRNsDfk9tfpvIlhpjT1B5Ebk08wZKojyUm
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '20.231.54.165' (ECDSA) to the list of known ho
azureuser@20.231.54.165's password:
[azureuser@QuickstartAnsible-vm ~]$ hostname
QuickstartAnsible-vm
[azureuser@QuickstartAnsible-vm ~]$
```

9. Create the file **`AzureAnsibleInstall.sh`**:

```
[azureuser@QuickstartAnsible-vm ~]$ touch AzureAnsibleInstall.sh
[azureuser@QuickstartAnsible-vm ~]$ vi AzureAnsibleInstall.sh
```

10. Add steps to the file **`AzureAnsibleInstall.sh`** as shown below. Basically, we are installing **`pip`** and then using **`pip`** to install **Ansible 2.9**:

```
#!/bin/bash

# Update all packages that have available updates.
sudo yum update -y

# Install Python 3 and pip.
sudo yum install -y python3-pip

# Upgrade pip3.
sudo pip3 install --upgrade pip

# Install Ansible.
pip3 install "ansible==2.9.17"

# Install Ansible azure_rm module for interacting with Azure.
pip3 install ansible[azure]
```

11. Save the file **`AzureAnsibleInstall.sh`** and **`chmod a+x`** to make an executable before running it, shown as follows:

```
Loaded plugins: langpacks
No packages marked for update
Package python3-pip-9.0.3-8.el7.noarch already installed and latest versic
Nothing to do
WARNING: pip is being invoked by an old script wrapper. This will fail in
Please see https://github.com/pypa/pip/issues/5599 for advice on fixing th
To avoid this problem you can invoke Python with '-m pip' instead of runni
Requirement already satisfied: pip in /usr/local/lib/python3.6/site-packag
WARNING: Running pip as the 'root' user can result in broken permissions a
Collecting ansible==2.9.17
Downloading ansible-2.9.17.tar.gz (14.3 MB)
...
```

12. Verify the Ansible install and version using **`ansible –version`** as shown :

```
[azureuser@QuickstartAnsible-vm ~]$ ansible --version
ansible 2.9.17
```

```
  config file = None
  configured module search path = ['/home/azureuser/.ansible/plugins/modul
  ansible python module location = /home/azureuser/.local/lib/python3.6/si
  executable location = /home/azureuser/.local/bin/ansible
  python version = 3.6.8 (default, Nov 16 2020, 16:55:22) [GCC 4.8.5 2015(
[azureuser@QuickstartAnsible-vm ~]$
```

Installing Ansible on Docker container

In this section, we will talk about installing Ansible on a Docker container. Follow the steps to have containerized Ansible in seconds.

1. Create the directory in your VM and in that location add **`Dockerfile`** with the following content:

```
FROM centos:7

ENV LANG en_US.UTF-8
ENV LC_ALL en_US.UTF-8

RUN yum check-update; \
    yum install -y gcc libffi-devel python3 epel-release; \
    yum install -y python3-pip; \
    yum install -y wget; \
    yum clean all

RUN pip3 install --upgrade pip; \
    pip3 install --upgrade virtualenv; \
    pip3 install pywinrm[kerberos]; \
    pip3 install pywinrm; \
    pip3 install jmspath; \
    pip3 install requests; \
    python3 -m pip install ansible; \
    ansible-galaxy collection install azure.azcollection; \
    pip3 install -r ~/.ansible/collections/ansible_collections/azure/azcol
```

2. Build the image using the **`Dockerfile`** created in the previous step. For this run, **`docker build . -t ansible`**, shown as follows:

```
pankajs-iMac-2:develop pankajsabharwal$ docker build . -t ansible
[+] Building 581.6s (7/7) FINISHED
```

Next, you can verify the new image using **`docker images`**, as shown :

```
REPOSITORY          TAG                 IMAGE ID            CREATED
ansible             latest              9ad9dc97e015        5 minutes ago
```

3. Once you have verified the image successfully, you can run the Docker container using **`docker run -it ansible`** as shown below. Next, you can also verify the Ansible version using **`ansible –version`** command **`o`** container:

```
[root@22897d45f229 /]# ansible --version
ansible [core 2.11.12]
  config file = None
  configured module search path = ['/root/.ansible/plugins/modules', '/usr
  ansible python module location = /usr/local/lib/python3.6/site-packages/
  ansible collection location = /root/.ansible/collections:/usr/share/ansi
  executable location = /usr/local/bin/ansible
  python version = 3.6.8 (default, Nov 16 2020, 16:55:22) [GCC 4.8.5 20150
  jinja version = 3.0.3
  libyaml = True
[root@22897d45f229 /]#
```

Installing Ansible on MacOS

We have seen above how we can install Ansible on Linux operating systems on different cloud providers. MacOS is a very common OS among the developer community and MacOS is not getting available as a compute instance on different cloud providers like AWS. So, this gives reasons to see how we can use Ansible on MacOS:

1. For installing Ansible on MacOS simplest way is to use brew installed. For this you need Homebrew installed on your Mac. If you do not have Homebrew installed on your mac, then you can install it using the command **`/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)`**.
    a. Once you have brew installed, you can just run **`brew install ansible`** on your MacOS to install ansible:

```
Installing dependencies for ansible: ca-certificates, openssl@1.1, sql
Reinstalling CA certificate bundle from keychain, this may take a whil
...
Installing ansible dependency: openssl@1.1
Pouring openssl@1.1--1.1.1q.mojave.bottle.tar.gz
...
Installing ansible dependency: sqlite
Pouring sqlite--3.39.2.mojave.bottle.tar.gz
...
Installing ansible dependency: xz
```

```
Pouring xz--5.2.5.mojave.bottle.tar.gz
...
Installing CA certificates: 95% done...
Installing ansible--2.9.15: python3.10
Pouring python@3.10--3.10.5.mojave.bottle.tar.gz
...
/usr/local/Cellar/python@3.10/3.10.5/bin/python3 -m pip install --no-d
Installing ansible...
Pouring ansible--2.9.15.mojave.bottle.tar.gz
...
'brew cleanup' has not been run in the last 30 days, running now...
/usr/local/Cellar/ansible/2.9.15: 22,738 files, 341.9MB
```

b. Once the ansible is installed, you can verify the version using the **`ansible –version`** command. You can see the version in the output as shown:

```
ansible [core 2.13.1]
  config file = None
  configured module search path = ['/Users/pankajsabharwal/.ansible/pl
  ansible python module location = /usr/local/Cellar/ansible@3.0/3.0.0
  ansible collection location = /Users/pankajsabharwal/.ansible/collec
  executable location = /usr/local/bin/ansible
  python version = 3.10.5 (main, Jun 23 2022, 17:51:25) [Clang 13.1.6
  jinja version = 3.1.2
  libyaml = True
  ansible-lint = not available
```

Installing Ansible on Windows OS

Windows after Linux and MacOS is another OS where ansible is used commonly. Windows is used both as a local OS among developers and also is available as an OS on major cloud providers. There are a couple of ways you can install Ansible on windows but the most straight forward way you can get going is via the Cygwin route. Follow the steps to setup Cygwin and Ansible:

1. If you do not have Cygwin installed then go here **https://cygwin.com/setup-x86_64.exe** and download the installer for Cygwin.
2. Double click the installer executable just like any other **`exe`** file. And you will see the **Setup Program** as shown in *Figure 2.8*:

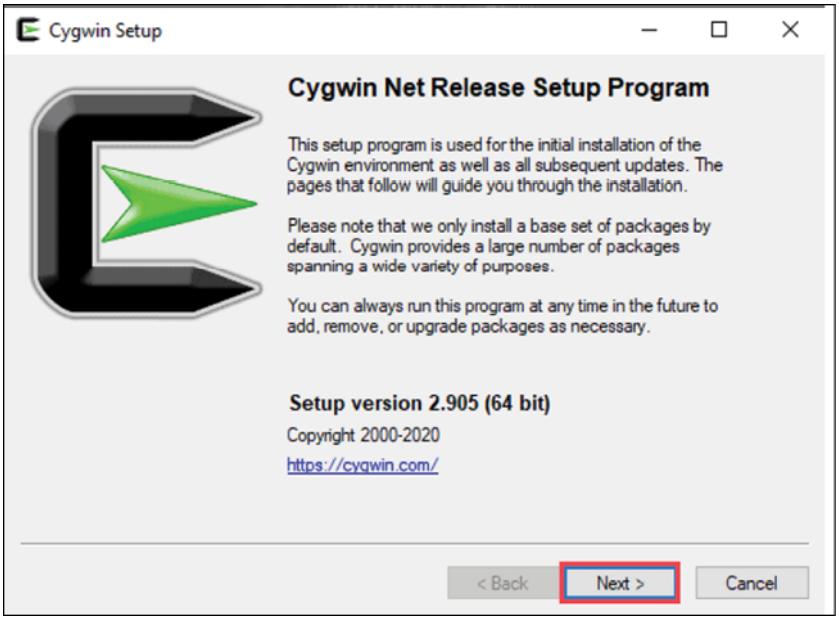

Figure 2.8: *Setup program GUI for Ansible*

3. Click **Next** and select the option to **Install from Internet** as shown in *Figure 2.9*:

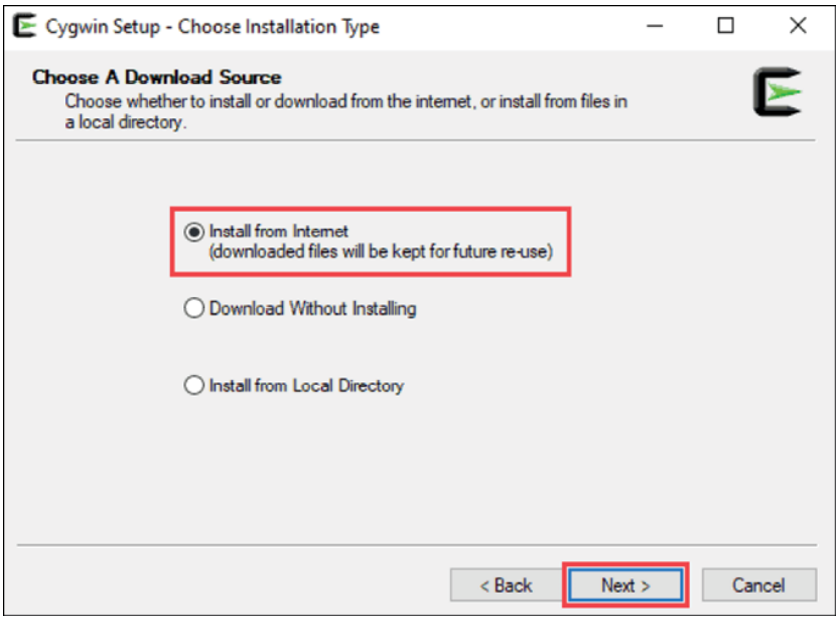

Figure 2.9: *Cygwin program asking for installation type*

4. Click **Next** and select the **installation directory** as shown in *Figure 2.10*:

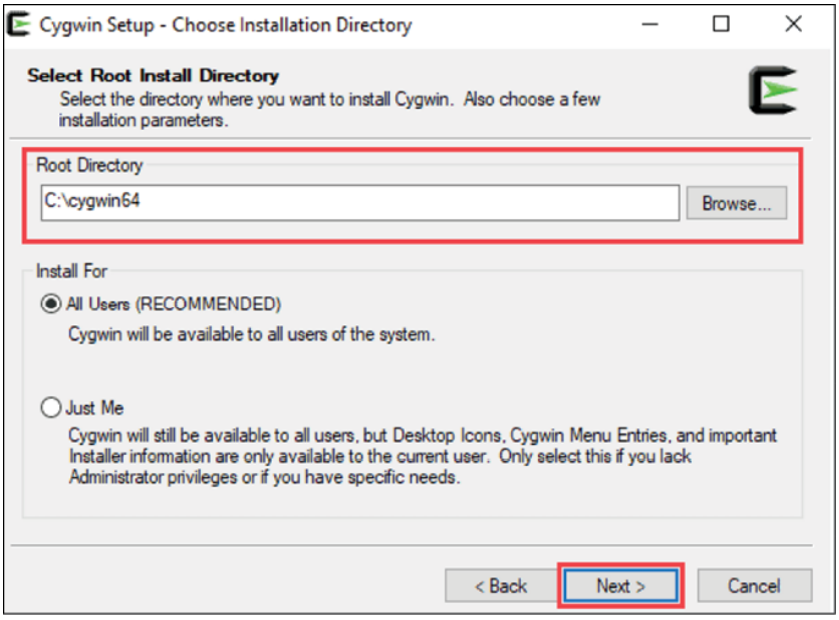

Figure 2.10: *Cygwin program install location*

5. Click **Next** and mention the package install location. Please refer to the following figure:

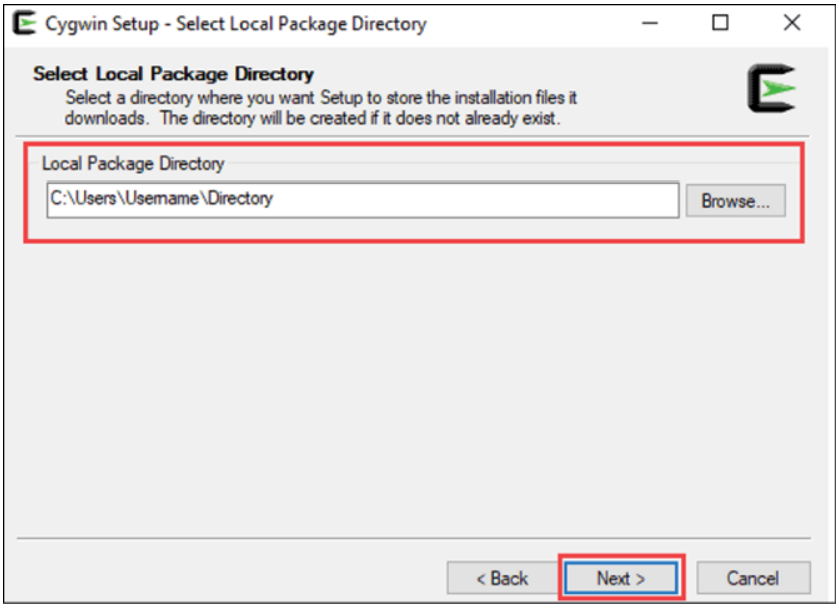

Figure 2.11: *Cygwin program asking local package install directory*

6. Click **Next** and specify your internet connection type as shown in *Figure 2.12*:

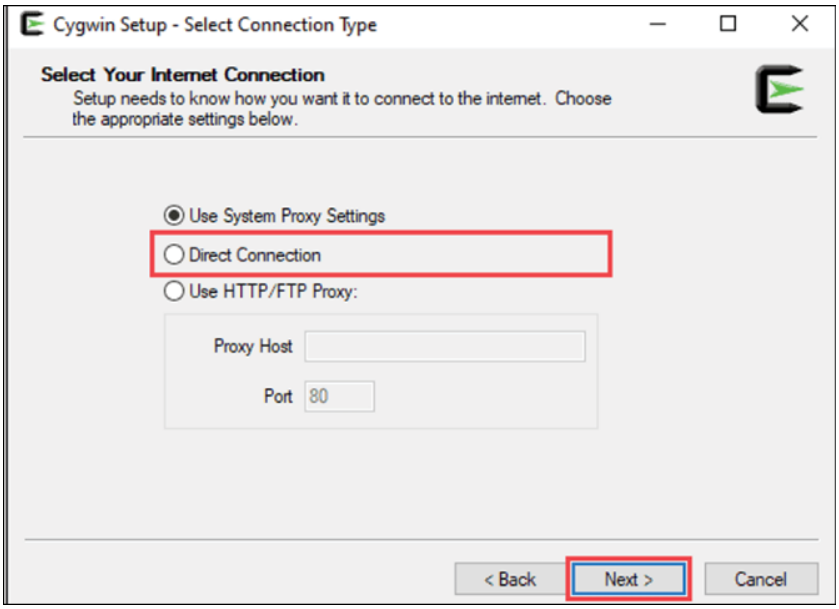

Figure 2.12: *Cygwin program asking for internet connection type*

7. Click **Next** and you will see different mirror options for downloading the packages, as shown in *Figure 2.13*:

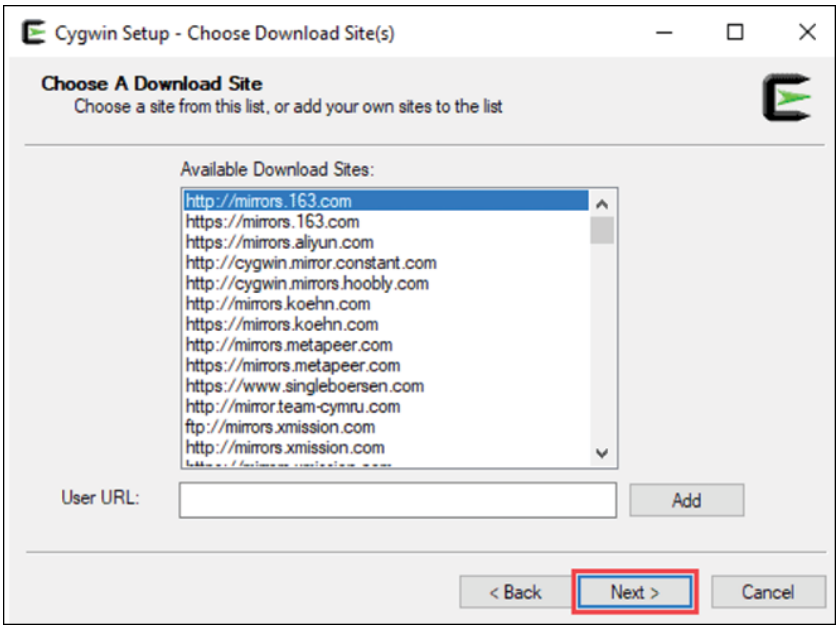

Figure 2.13: *Cygwin program asking for mirror location for downloading the packages*

8. Click **Next** and specify the packages you are interested in as shown in the following figure:

Note: Package ansible-doc is optional here.

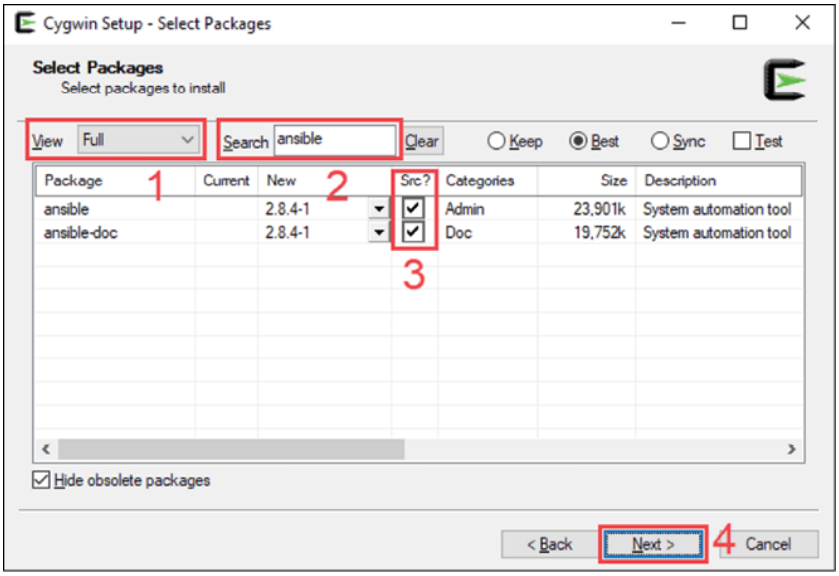

Figure 2.14: *Cygwin program asks for packages you want to install*

9. Click **Next,** and in the next screen, click **Finish** so that you have the **Desktop** icon for Cygwin and Cygwin to **Start Menu** as shown in the following figure:

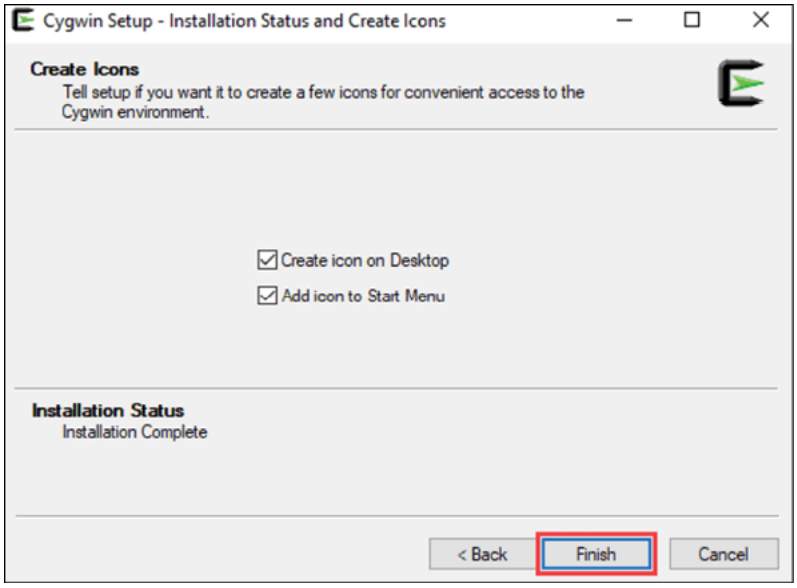

Figure 2.15: *Cygwin program asking to create a Desktop icon*

10. Verify the Ansible version on Windows OS as shown:

```
$ ansible --version
ansible 2.8.4
  config file = /etc/ansible/ansible.cfg
  configured module search path = ['/home/scorpio_ckumar/.ansible/plugins/
  ansible python module location = /usr/lib/python3.7/site-packages/ansibl
  executable location = /usr/bin/ansible
```

```
python version = 3.7.4 (default, Jul 21 2019, 14:43:25) [GCC 7.4.0]
```

Installing Ansible in other Linux distros

Basically, the steps are the same for all distros as we have seen for Red Hat Enterprise Linux in earlier sections.

We will quickly go through the steps for other Linux distros as follows:

1. **Fedora**

   With Fedora you just need to use **`dnf`** instead of **`yum`**. So Ansible can be installed using the command **`sudo dnf install ansible`**.

2. **Ubuntu**

   With Ubuntu instead of **`yum`** or **`dnf`** we will use **`apt`**. So, the command to install Ansible will be **`sudo apt install ansible`**.

3. **CentOS**

   CentOS is like Red Hat Enterprise Linux. So, the command will be the same which is **`sudo yum install ansible`**.

Conclusion

In this chapter, we have observed, how easy it is to install and setup Ansible on different cloud providers and operating systems and even how easily we can run ansible in Docker container. The installation and setup of Ansible do not need any technical expertise and you are up and ready to go with Ansible in very less time unlike other config management tools. Since Ansible is agentless in nature it would not take any compute resources on your machine or container.

In the next chapter, we will talk more in detail about Ansible and how we can write our Ansible Playbooks.

Join our book’s Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 3
Writing Tasks, Plays, and Playbooks

Introduction

Any action which needs to be repeated should be automated. In this chapter, we will cover Ansible Playbooks and we will see how playbooks can be written in a simple way to execute repeatable tasks on multiple hosts for running simple to very complex deployments. Since this is a step towards automation hence it removes the chances of any human error. In all, Ansible Playbooks can be used for configuration management, orchestrating manual steps in the desired sequence on multiple target hosts and launching child tasks regardless if those are synchronous or asynchronous.

The coverage will also include the grouping of Ansible tasks to create an Ansible play and how these plays can be grouped together to create an Ansible Playbook.

Structure

In this chapter, we will learn the following topics:

- Basic Ansible concepts
  - Control node
  - Managed node
  - Inventory
  - Playbooks
    - Plays
    - Roles
    - Tasks
    - Handlers
  - Modules
  - Plugins
  - Collections
  - AAP
- General structure of playbook
  - Directory layout
  - Configuration breakdown
- Dynamic inventories with Ansible

    - Steps to setup dynamic inventory for AWS EC2 instances
- Real-life example of the Ansible Playbook
    - Creating a playbook to install the NGINX server
    - Running the playbook
    - Anatomy of playbook

Objectives

The objective of this chapter is to cover the basic concepts of Ansible so that one can understand the basic structure of the playbook, how the playbook can be executed, and where the playbook can be executed. We will use a real-life example playbook to facilitate the learning.

Basic Ansible concepts

Before deep diving into Ansible, it is important to understand some basic concepts as these are common to all Ansible use cases. The details will not be covered extensively here, but the topics will be addressed sufficiently to understand all the content covered in this book.

Control node

The Control node is the machine where Ansible is installed and from which automation tasks are orchestrated. It is responsible for running Ansible commands and playbooks, communicating with target nodes (where tasks will be executed). Although traditionally this node was Linux-based, with the introduction of tools like the **Windows Subsystem for Linux** (**WSL**), it can also be hosted on a Windows system. Essential command-line tools such as `ansible-playbook`, `ansible-vault`, and ansible are executed from the Control node.

Managed nodes

These are the hosts, where you target your Ansible Playbooks to deploy the changes. These can be any server, network device, database server and so on. Normally, we do not install Ansible on these host systems.

Inventory

Inventory is a list of target nodes or managed nodes where the `ansible-playbook` will deploy changes. In inventory, we can group managed nodes based on different criteria. For example, we can group all WebServer nodes together so if we want the Ansible Playbook to deploy the change on all WebServers then we just point to the group WebServers instead of mentioning each host individually.

Playbook

An Ansible Playbook is a structured document that defines one or more plays, designed to automate tasks on managed nodes specified in the inventory. Each play is made up of metadata and a series of tasks. The metadata can include variables, host selections, and information about which user should execute the play. Tasks within a play describe the actual automation steps, such as installing a package or starting a service. Essentially, a playbook provides a scriptable framework that describes the desired state or actions to be performed on target nodes.

Following *Figure 3.1* shows **Playbook** execution to a list of servers in the inventory file:

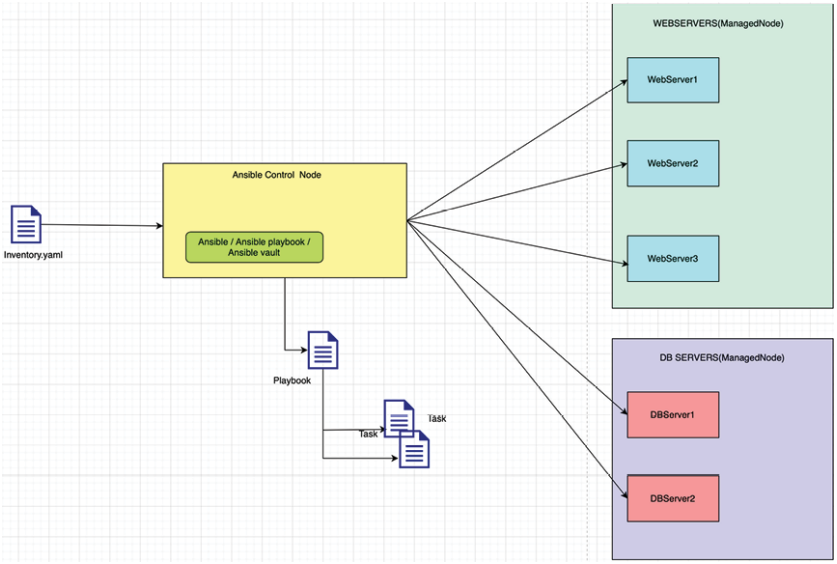

Figure 3.1: *Playbook execution to servers listed in the inventory file*

Modules

At the heart of Ansible's power and flexibility are its modules. Ansible modules are essentially the building blocks used for performing specific tasks in a playbook. Think of them as the tools in Ansible's automation toolbox:

- **Pre-defined scripts**: Modules are essentially pre-defined scripts that can be executed on target managed nodes. These scripts encapsulate specific functionalities and abstract the underlying complexities. For instance, rather than manually crafting commands to install a package, you can use the `apt` or `yum` module (depending on your OS) to handle that task.
- **Arguments and customization**: Each module accepts specific arguments which allow you to tailor its behavior. For example, with the user module, you can define the name of the user, whether the user should exist or not, the user's password, and more.
- **JSON output**: After a module is executed, it provides output in JSON format. This standardized output format allows Ansible to be system agnostic, meaning it can work with various systems and tools that understand JSON.
- **Reusable and idempotent**: One of the significant benefits of modules is that they are reusable. The same module can be used across different playbooks and roles. Moreover, they are often designed to be idempotent, which means running the module multiple times will not have any unintended side effects. If a system is already in the desired state, the module will recognize it and refrain from making any changes.
- **Extensive library and community contribution**: Ansible comes with a massive library of built-in modules catering to various needs - from system administration tasks, like managing users and installing packages, to more advanced operations, like interacting with cloud platforms or databases. The open-source nature of Ansible has also led the community to contribute a plethora of custom modules, expanding its capabilities even further.
- **Invoking modules**: While you can run modules directly using the Ansible command-line tool for **ad-hoc** tasks, the real power comes when you incorporate them into

playbooks. In a playbook, each task invokes a specific module, and you can sequence these tasks to achieve complex automation workflows.

In summary, Ansible modules are the workhorse of Ansible automation, providing users with a wide range of capabilities without requiring them to get into the nitty-gritty details of every task.

Plugins

Ansible plugins are pieces of code which augment the core functionality of Ansible and make it more flexible. Example: `action` plugins are available which can be used to test and validate the data in play and playbooks. Similarly, there are `var` plugins available which enable the injection of additional custom data during playbook runtime.

Collections

Ansible collections are formats by which you make Ansible content like playbooks, roles, plays and so on, to other Ansible users across the ansible community. These collections are extracted from Ansible galaxy in a very similar way to how we import the role.

Ansible Automation Platform

Ansible Automation Platform (AAP) provides an enterprise grade framework and platform for building IT automation at scale from hybrid cloud to **Egde**. Using AAP, IT teams, from network to security to developers, can collaborate with each other and share, manage and operate automation in a secure way. It has multiple components like the Ansible execution environment, Ansible controller, and Ansible mesh which we will go through in detail in a later chapter of the book.

General structure of Ansible Playbook

The general structure of an Ansible Playbook involves a directory layout understanding and a comprehensive grasp of the playbook's structure. This encompasses topics such as understanding the directory layout in Ansible and gaining insights into the overall structure of a playbook.

Understanding directory layout

In order to understand the structure of the Ansible Playbook, it is important and useful to understand the directory layout of Ansible. We have an idea of what files and directories are being referred to in the Ansible Playbook. So, we will cover the directory layout quickly as follows:

Following are the components which make Ansible work:

- **Inventory files:** Inventory is a list of machines or hosts you want Ansible to manage. By default, it is located at `/etc/ansible/hosts` but it is customizable.
- **Group vars:** In Ansible, `group_vars` are used to apply variables to multiple hosts at once.
- **Host vars:** These are in the same directory as `group_vars`. It contains data models that apply to individual hosts or devices specified in inventory files.
- **Playbooks:** These are blueprints of automation tasks. Playbooks can call other playbooks or other roles to execute the tasks in the desired sequence.
- **Roles:** These are sets of tasks to manage a host or multiple hosts.

- **Task:** These are the smallest units of action we can create to automate using a playbook.
- **Templates:** These are files that contain all configuration parameters, but values are uploaded dynamically using different variables.

Structure of Ansible Playbook

So now we understand the Ansible directory structure. We are in a position to look at what the general Ansible Playbook looks like.

So, let us look at the simple playbook below for installing the Apache webserver:

```
---
- name: Playbook
  hosts: webservers
  become: yes
  become_user: root
  tasks:
    - name: ensure apache is at the latest version
      yum:
        name: httpd
        state: latest

    - name: ensure apache is running
      service:
        name: httpd
        state: started
```

So, what we can tell about the playbook?

- The name of the play is **`Playbook`**.
- It will install Apache on all hosts/machines defined under the group **`webservers`** in the inventory file.
- We need to run some tasks as **`root`**.
- The first task is to run **`yum`** for installing package **`httpd`**. We will install the **`latest`** version of **`apache`**.
- The second task is to use the **`service`** module and define the service for **`httpd`** to ensure it is running after the host's reboot.
- All the above is accomplished within a single play.

Now let us look at the following playbook which has multiple plays:

```
---
# Play1 - WebServer related tasks
```

```
- name: Play Web - Create apache directories and username in web servers
  hosts: webservers
  become: yes
  become_user: root
  tasks:
    - name: create username apacheadm
      user:
        name: apacheadm
        group: users,admin
        shell: /bin/bash
        home: /home/weblogic

    - name: install httpd
      yum:
        name: httpd
        state: installed

# Play2 - Application Server related tasks
- name: Play app - Create tomcat directories and username in app servers
  hosts: appservers
  become: yes
  become_user: root
  tasks:
    - name: Create a username for tomcat
      user:
        name: tomcatadm
        group: users
        shell: /bin/bash
        home: /home/tomcat

    - name: create a directory for apache tomcat
      file:
        path: /opt/oracle
        owner: tomcatadm
        group: users
```

```
    state: directory
```

We can tell the following about this playbook:

- This playbook contains 2 plays. The first play is **`Play Web`** and the second play is **`Play App`**.
- This playbook will run tasks on all hosts or machines defined under the group **`webserver`** in the inventory file.
- There are 2 tasks under the first play. First task is to create a user **`apacheadm`**. The second task is to use **`yum`** module to install the **`httpd`** package.
- There are 2 tasks under the second play. The first task is to create user **`tomcatadm`** and the second task will create the **`directory`** before defining **`directory`** permissions and ownerships.

Dynamic inventories with Ansible

When you are using a cloud service provider you do not have static IPs hence maintaining IPs in ansible inventory files is not useful. In these cases, we can make use of the **`dynamic`** inventory option. We will understand this by using AWS as our cloud service provider.

The following section will show how we can setup AWS dynamic inventory for Ansible use.

Ensure we have **`python`** and **`pip`** installed on our Ansible control machine. The following code shows **`python`** and **`pip`** installed with their respective versions:

```
[ec2-user@ip-172-31-87-164 tmp]$ python3 --version

Python 3.6.8

[ec2-user@ip-172-31-87-164 tmp]$ pip3 --version

pip 9.0.3 from /usr/lib/python3.6/site-packages (Python 3.6)

[ec2-user@ip-172-31-87-164 tmp]$
```

Since we are on **RHEL8** host hence **`python`** is installed as **`python3`** and **`pip`** is installed as **`pip3`**. If your system does not have these binaries installed, then you can install easily using **`sudo yum install -y python3`** and **`sudo yum install -y python3-pip`**:

1. **Install boto3 library:**

   Ansible uses the **`boto`** library to make API calls to AWS. You can use **`pip`** to install the **`boto`** library using the command **`sudo pip3 install boto`**. In our case, **`boto`** is already installed hence we got the following output, as shown:

   ```
   [ec2-user@ip-172-31-87-164 tmp]$ sudo pip3 install boto

   WARNING: Running pip install with root privileges is generally not a good

   Requirement already satisfied: boto in /usr/local/lib/python3.6/site-packa

   [ec2-user@ip-172-31-87-164 tmp]$
   ```

2. **Create inventory directory and file:**

   a. Create the **`directory`** for inventory using **`sudo mkdir -p /opt/ansible/inventory && cd /opt/ansible/inventory/`**.

   b. Create the **`aws_ec2.yaml`** file using **`touch aws_ec2.yaml`**. Copy the content below in this file. You will replace the values with your AWS access key and AWS secret key, as:

Note: Do not store this file in public repositories or expose it on the internet.

```
---
plugin: aws_ec2
aws_access_key: <YOUR-AWS-ACCESS-KEY-HERE>
aws_secret_key: <YOUR-AWS-SECRET-KEY-HERE>
keyed_groups:
  - key: tags
    prefix: tag
```

3. **Setup ansible.cfg file:**

   By default `ansible.cfg` is located at `/etc/ansible/ansible.cfg`. Add line `enable_plugins = aws_ec2` in the `[inventory]` section so your file should look like as shown:

   ```
   [inventory]
   # enable inventory plugins, default: 'host_list', 'script', 'auto', 'yaml'
   #enable_plugins = host_list, virtualbox, yaml, constructed
   enable_plugins = aws_ec2
   ```

Once you have done the previously mentioned steps you can test the dynamic inventory configuration by listing `ec2` instances using the command `ansible-inventory -i /opt/ansible/inventory/aws_ec2.yaml –list`.

Real-life example of Ansible Playbook

In this section, we will see how to setup `NGINX WebServer` on our RHEL 8 hosts.

Creating a playbook for NGINX

Basically, our playbook will contain the following components. We will cover each component separately:

- Playbook file
- Template for `NGINX HTML` page
- Configuration file for `NGINX`

Following is our playbook for `NGINX` and we are calling it `nginx.yaml`. Refer to the following section:

```
- name: Configure NGINX WebServer
  hosts: localhost
  become: True
  become_user: root
  tasks:
    - name: install nginx
```

```
      yum: name=nginx update_cache=yes

    - name: copy nginx config file
      copy: src=files/nginx.conf dest=/etc/nginx/sites-available/default

    - name: enable configuration
      file:
        dest: /etc/nginx/sites-enabled/default
        src: /etc/nginx/sites-available/default
        state: link

    - name: copy index.html
      template: src=templates/index.html.j2
dest=/usr/share/nginx/html/index.html
      mode: 0644

    - name: restart nginx
      service: name=nginx state=restarted
```

We have **nginx.conf** file which we will use as shown below:

```
server {
    listen 80 default_server;
    listen [::]:80 default_server ipv6only=on;

    root /usr/share/nginx/html;
    index index.html index.htm;

    server_name localhost;

    location / {
        try_files $uri $uri/ =404;
    }
}
```

The **HTML** page template we will deploy is shown as follows:

```
<html>
    <head>
        <title>Welcome to ansible</title>
    </head>
```

```
    <body>
        <h1>nginx, configured by Ansible</h1>
        <p>If you can see this, Ansible successfully installed nginx.</p>
        <p>Running on {{ inventory_hostname }}</p>
    </body>
</html>
```

So our **directory** structure will look like, as shown below. We are using the **tree** command to see this:

```
.
├── files
│   └── nginx.conf
├── hosts
├── nginx.yaml
└── templates
    └── index.html.j2
```

Running playbook for NGINX

Playbook **nginx.yaml** is run using **sudo ansible-playbook nginx.yaml** command. You can run the playbook in check mode to see if all tasks will run successfully or no using **sudo ansible-playbook nginx.yaml --check** command but that is optional and we will cover this option in debugging sections in later chapters of this book. The output of the Ansible run is shown, as follows:

```
[ec2-user@ip-172-31-87-164 ansible]$ sudo ansible-playbook nginx.yaml

[WARNING]: provided hosts list is empty, only localhost is available

[WARNING]: No inventory was parsed, only implicit localhost is available

[WARNING]: provided hosts list is empty, only localhost is available. Note
that the implicit localhost does not match 'all'

/usr/lib/python3.6/site-packages/requests/__init__.py:91:
RequestsDependencyWarning: urllib3 (1.26.12) or chardet (3.0.4) doesn't
match a supported version!

  RequestsDependencyWarning)


PLAY [Configure NGINX WebServer]
***********************************************


TASK [Gathering Facts]
*************************************************************

ok: [localhost]
```

```
TASK [Install nginx]
*********************************************************

ok: [localhost]


TASK [Copy nginx config file]
***********************************************

ok: [localhost]


TASK [Enable configuration]
***********************************************

ok: [localhost]


TASK [Copy index.html] ***********************************************

ok: [localhost]


TASK [Restart nginx] *************************************************

changed: [localhost]


PLAY RECAP
*********************************************************************


localhost                  : ok=6    changed=1    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0
```

Anatomy of playbook for NGINX

Now, let us take a deeper look at the previous **`NGINX playbook`** to see what components make this playbook.

As we know the Ansible Playbook is nothing but a set of different individual plays. In this playbook, we see that it consists of only one play, which installs and configures **`NGINX webserver`**. Let us look at more about this playbook as follows:

- **Hosts:** Host section in the Ansible Playbook specifies what hosts the **`playbook`** has to run on. Here we see it runs only on **`localhost`**.
- **Tasks:** Tasks are individual steps under one Ansible play. Here we have 5 tasks and let us discuss them:
  - First task is using the **`yum`** module to install **`nginx`** package.
  - Second task is using **`copy`** module to copy the **`nginx config`** file to target location.
  - Third task is creating a symbolic link between two different directories.
  - Fourth task copies the **`index.html`** page which is served by the **`webserver root`**.
  - Fifth task restarts the **`nginx webserver`** to enable all the deployed changes.
- **Optional play settings:**
  - **Name:** Name of the play.

    - **Become:** If you need to run a task or more using some other user like `root`.
    - **Vars:** Specifies lists of variables and their respective values. Here in this playbook, we have not used any variables.
- **Modules:** Modules are nothing but standard out of the box reusable, standalone scripts that can be used by Ansible API or other Ansible programs in accomplishing automation tasks. So, in this playbook we have used various modules for each task.
    - **Yum:** `Yum` is used to manage packages using the `yum` package manager. Here, we installed and updated the `Nginx` package using the `yum` module.
    - **Copy:** `Copy` module copies the files from the local to the remote target system. Since we are running the Ansible Playbook only on `localhost`, a copy of a file like `nginx.conf` will happen from the files directory under Ansible to the `webserver` director on the same system.
    - **File:** `File` module is used to create and delete file. We can also use this module in the creation of soft links and hard links. In our `playbook`, we have used this module for the creation of soft link or symbolic link.
    - **Template:** A template is a file which contains configuration parameters. But in some cases, the values for each variable have to be fed dynamically and that is where once can use variables. The template module also, by default copies the file to the target server and uses **Jinja2** to render the values dynamically from variables like Ansible facts. In our case we have used an `index.html` file as a template.
    - Service this module controls the services on the remote system. Here in our playbook, we are using a service module to define and restart the service for `nginx`.

Conclusion

In conclusion, we have covered a wide range of topics related to Ansible tasks, plays, and playbooks, providing you with a comprehensive understanding of this powerful automation tool. You now possess the knowledge and skills necessary to effectively utilize Ansible in your configuration management and automation endeavors.

Throughout this journey, you have learned how to structure Ansible Playbooks, organize tasks, and create plays that orchestrate actions on target systems. By following best practices and leveraging Ansible's declarative language, you can write concise and efficient playbooks tailored to your specific needs. With this foundation in place, the next chapters will take you even further into the realm of Ansible automation, with a deep dive into infrastructure automation. This exciting exploration will equip you with the insights and techniques required to automate complex infrastructure setups, streamline deployments, and manage a scalable and resilient IT environment. By understanding how to leverage Ansible's vast array of modules, plugins, and inventory management, you will unlock the potential to automate repetitive tasks, reduce human error, and enhance overall operational efficiency. Whether it is provisioning virtual machines, configuring network devices, or deploying applications, Ansible empowers you to automate these processes with ease.

In the upcoming chapters on *Infrastructure Automation using Red Hat Ansible*, we will explore advanced topics such as managing dynamic inventories, handling variable data, working with roles, and incorporating Ansible into a continuous integration and deployment pipeline. These insights will enable you to tackle real-world scenarios and deliver robust and scalable automation solutions.

In summary, your journey into Ansible has equipped you with the fundamental knowledge and practical skills necessary to harness the full potential of this remarkable automation framework. As we venture deeper into infrastructure automation, prepare to broaden your expertise and transform the way you manage and orchestrate your infrastructure, revolutionizing your approach to IT operations.

Join our book’s Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 4
Infrastructure Automation Using Red Hat Ansible

Introduction

Infrastructure automation with Ansible is relevant in different cloud providers because it allows you to manage your infrastructure in a consistent, repeatable, and scalable way, across multiple cloud providers and regions.

For example, if you are using multiple cloud providers such as **Amazon Web Services** (**AWS**), **Azure**, and **Google Cloud Platform** (**GCP**), Ansible can help you manage your infrastructure in a consistent way, regardless of the underlying infrastructure. Ansible has built-in modules for working with different cloud providers, such as the `ec2` module for AWS, the `Azure` module for Azure, and the `GCP` module for GCP.

With Ansible, one can write playbooks and roles that define your infrastructure as code, and then use those playbooks and roles to provision and configure infrastructure resources across multiple cloud providers. This can help you reduce the time and effort required to manage your infrastructure, and ensure that your infrastructure is consistent and compliant across different cloud providers.

In addition, Ansible allows you to leverage the APIs and services provided by different cloud providers, in order to automate tasks such as provisioning instances, configuring networks, setting up load balancers, and managing security groups. This can help you achieve better agility, flexibility, and cost-effectiveness when working with different cloud providers.

Some of the benefits of infrastructure automation with Ansible include:

- **Consistency:** Ansible allows you to define your infrastructure as code, which means that you can ensure consistency across different cloud providers and regions. By defining your infrastructure as code, you can ensure that it is provisioned, configured, and managed in a consistent and repeatable way.
- **Scalability:** Ansible allows you to easily scale your infrastructure up or down as needed, by automating the process of adding or removing resources. This can help you reduce costs and improve agility.
- **Efficiency:** Ansible can help you automate routine and repetitive tasks, freeing up your team to focus on higher-level tasks. By automating infrastructure provisioning, management, and configuration, you can reduce the time and effort required to manage your infrastructure.
- **Flexibility:** Ansible supports a wide range of cloud providers, including AWS, Azure, GCP, and others, as well as on-premises infrastructure. This flexibility allows you to use Ansible to manage your infrastructure, regardless of the underlying infrastructure.

- **Security:** Ansible can help you ensure that your infrastructure is secure by allowing you to automate the configuration of security policies, access controls, and other security-related settings.
- **Compliance:** Ansible can help you ensure that your infrastructure is compliant with regulatory requirements and industry standards by allowing you to automate the configuration of compliance-related settings.

Overall, infrastructure automation with Ansible can help you streamline your IT operations, reduce the risk of errors and downtime, and free up your team to focus on higher-level tasks.

Structure

In this chapter, we will go over the following topics:

- Infrastructure automation using Ansible on AWS
- Infrastructure automation using Ansible on Azure
- Infrastructure automation using Ansible on GCP

Objectives

The objective of this chapter is to provide readers with a comprehensive understanding of infrastructure automation using Red Hat Ansible. We aim to equip readers with practical knowledge and skills to provision and manage infrastructure on major cloud platforms, namely AWS, GCP, and Microsoft Azure.

In the context of AWS, we will delve into the intricacies of managing resources such as **Elastic Compute Cloud** (**EC2**), **Simple Storage Service** (**S3**), and **Virtual Private Cloud** (**VPC**). We will explore how to utilize Ansible for automating tasks related to these services and provide sample codes and use cases for further illustration.

For Azure, our focus will shift towards network management. We will highlight key Ansible modules like `azure_rm_subnet` and `azure_rm_securitygroup`, which are instrumental in managing virtual networks. We will also cover other pertinent Azure resources to provide a holistic view of infrastructure automation on Azure. Regarding GCP, we will spotlight modules such as `gcp_compute_subnetwork` and discuss how they facilitate managing GCP's infrastructure resources. Throughout this chapter, we aim to foster a clear understanding of how Red Hat Ansible can be utilized for infrastructure automation across these three major cloud platforms. By the end of this chapter, readers should be able to deploy, manage, and automate their cloud resources effciently using Ansible.

Infrastructure automation using Ansible on AWS

We can do almost any configuration automation of almost any infrastructure component on AWS, using Ansible. There are various modules available in Ansible for compute, networking, and storage components.

Following are some very common examples of provisioning, configuring and managing infrastructure components using Ansible:

- **Launching EC2 instances:** Ansible can be used to automate the process of launching `EC2 instances` on AWS, including specifying the instance type, AMI, key pair, security groups, and other configuration settings.
- **Configuring VPCs:** Ansible can be used to automate the configuration of VPCs on AWS, including creating subnets, routing tables, internet gateways, and other related resources.

- **Managing security groups:** Ansible can be used to automate the management of security groups on AWS, including creating, modifying, and deleting security groups, as well as adding and removing rules.
- **Configuring load balancers:** Ansible can be used to automate the configuration of load balancers on AWS, including creating and configuring application load balancers and network load balancers.
- **Managing auto scaling groups:** Ansible can be used to automate the management of auto-scaling groups on AWS, including creating, modifying, and deleting auto-scaling groups, as well as defining scaling policies.
- **Managing RDS instances:** Ansible can be used to automate the management of RDS instances on AWS, including launching RDS instances, modifying configuration settings, and managing backups.
- **Managing S3 buckets:** Ansible can be used to automate the management of S3 buckets on AWS, including creating, modifying, and deleting buckets, as well as managing permissions and access policies.

ec2 module

The `ec2` module is a core module in Ansible that allows you to manage Amazon Web Services Elastic Compute Cloud instances. It provides a set of tasks for launching, starting, stopping, and terminating `EC2 instances`, as well as for managing EC2 security groups, volumes, and snapshots.

Following are some of the key features of the `ec2` module:

- **Launching EC2 instances:** The `ec2` module provides a task to launch `EC2 instances` with a variety of configuration options, such as instance type, security groups, key pair, AMI, subnet, and more.
- **Managing EC2 instances:** The `ec2` module allows you to manage the state of `EC2 instances`, such as starting, stopping, and terminating instances.
- **Managing EC2 security groups:** The `ec2` module allows you to manage the state of EC2 security groups, such as creating and deleting security groups, adding and removing rules from security groups, and managing security group membership.
- **Managing EC2 volumes and snapshots:** The `ec2` module allows you to manage the state of EC2 volumes and snapshots, such as creating and deleting volumes and snapshots, attaching and detaching volumes from instances, and creating and restoring volumes from snapshots.
- **Tagging EC2 resources:** The `ec2` module allows you to tag `EC2 instances`, volumes, snapshots, and security groups with custom key-value pairs for better organization and management.

Note: To utilize the ec2 module in your Ansible Playbook, ensure the following prerequisites are met on the Ansible controller:

- **Python version:** Ensure that you have Python version 3.6 or higher.
- **Boto libraries:**
  - Install `boto3` with version 1.22.0 or above. This library provides the **Python SDK** for AWS.
  - While `botocore` is required with a version of 1.25.0 or higher, it will be installed by default when you install `boto3`.

- **Amazon.aws collection:** Install the `amazon.aws` collection on the controller using the following command:

  ```
  ansible-galaxy collection install amazon.aws
  ```

- **AWS credentials:** You will need to supply AWS credentials. This can be done via environment variables or configuration files.
- **Task variables:** Ensure you have all the necessary variables for your EC2 tasks ready, such as the region, AMI ID, instance type, and others.
- **Dynamic inventory:** Dynamic inventory in Ansible is used to integrate with external data sources, allowing Ansible to fetch a list of hosts from those sources at runtime rather than relying on a static predefined list. This is especially useful in cloud environments like AWS, where instances might be frequently provisioned or terminated.

  For AWS, Ansible offers a dynamic inventory script/plugin to fetch `EC2 instance` details. Here is how you can set it up:

  - First, ensure you have the `amazon.aws` collection installed:

    ```
    ansible-galaxy collection install amazon.aws
    ```

  - **Dynamic inventory script**:

    Use the EC2 dynamic inventory plugin provided by Ansible. To do this, you will need a configuration file (let us call it `ec2.ini`).

    A sample `ec2.ini` configuration could look like this:

    ```
    [ec2]

    regions = us-west-1

    instance_states = running
    ```

- **Dynamic inventory YAML:**

  You also need a `.yml` file (let us call it `ec2.yml`) that tells Ansible to use the EC2 plugin.

  ```
  plugin: amazon.aws.ec2
  ```

- **AWS credentials:**

  The dynamic inventory script will use your AWS credentials to fetch the list of `EC2 instances`. The credentials can be set in various ways:

  - **Environment variables:** Export `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, and `AWS_SESSION_TOKEN` (if you are using a temporary session).
  - **AWS credentials file:** Typically located at `~/.aws/` credentials.
  - **IAM role:** If you are running Ansible from an `EC2 instance`, you can attach an IAM role to the instance with the necessary permissions.

- How to use the dynamic inventory with Ansible:
  - When running an Ansible command or playbook, you specify the dynamic inventory file (`ec2.yml`) with the `-i` flag.

    ```
    ansible-inventory -i ec2.yml --graph
    ```

    The above command will display a graph of your `EC2 instances` fetched via the dynamic inventory.

  - When running playbooks:

`ansible-playbook -i ec2.yml your-playbook.yml`

By satisfying the above prerequisites, you can seamlessly integrate and run tasks involving the **`ec2`** module in your Ansible playbooks.

ec2 module in EC2 instance

Now let us see how Ansible's **`ec2`** module can be used in provisioning, configuring and managing the life cycle of **`EC2 instance`**. Refer to the **`playbook`** as follows:

```
---
- name: Launch an EC2 instance
  hosts: localhost
  gather_facts: False
  tasks:
    - name: Include AWS variables
      include_vars: vars/aws.yml

    - name: Launch EC2 instance
      amazon.aws.ec2:
        aws_access_key: "{{ aws_access_key }}"
        aws_secret_key: "{{ aws_secret_key }}"
        region: "{{ region }}"
        image: "{{ ami_id }}"
        instance_type: "{{ instance_type }}"
        key_name: "{{ key_pair }}"
        group: "{{ security_group }}"
        count: "{{ instance_count }}"
        tags: "{{ tags }}"
      register: ec2

    - name: Add new instance to host group
      add_host:
        hostname: "{{ item.public_ip }}"
        groupname: launched
      loop: "{{ ec2.instances }}"
      when: ec2.instances is defined

    - name: Wait for SSH to come up
      ansible.builtin.wait_for:
```

```
        host: "{{ item.public_dns_name }}"
        port: 22
        delay: 60
        timeout: 320
        state: started
      loop: "{{ ec2.instances }}"
      when: ec2.instances is defined

    - name: Print new EC2 instance details
      debug:
        msg:
          - "EC2 instance ID: {{ item.id }}"
          - "Public IP: {{ item.public_ip }}"
      loop: "{{ ec2.instances }}"
      when: ec2.instances is defined
```

This `playbook`:

- **Includes AWS variables:** Loads the AWS-related variables from `vars/aws.yml`.
- **Launches the EC2 instance:** Uses the `amazon.aws.ec2` module to launch the desired `EC2 instance`.
- **Adds the new instance to a host group:** Registers the newly launched instance to a host group called launched.
- **Waits for SSH:** After launching the instance, it waits for `SSH` to come up, ensuring the instance is fully ready.
- **Prints the new EC2 instance details:** Finally, it prints out the details of the newly launched instance.

Note: Before running this playbook, ensure that the amazon.aws collection is installed (ansible-galaxy collection install amazon.aws). Additionally, always ensure that any sensitive information in vars/aws.yml is kept secure and not exposed in version control or other public spaces.

Following is how the `var/aws.yml` variable file should look like, which this `playbook` refers:

```
# vars/aws.yml

# AWS Access and Secret keys, it's recommended to use IAM roles or
environment variables instead of hardcoding these.
aws_access_key: "YOUR_AWS_ACCESS_KEY"
aws_secret_key: "YOUR_AWS_SECRET_KEY"

# Region where you want to launch the EC2 instance
region: "us-west-1"
```

```
# The Amazon Machine Image (AMI) ID
ami_id: "ami-xxxxxxxxxxxxxx"

# The type of instance to launch
instance_type: "t2.micro"

# Name of the security group
security_group: "my-security-group"

# Key pair name
key_pair: "my-key-pair"

# EC2 instance count
instance_count: 1

# Any other tags or metadata
tags:
  Name: "MyAnsibleManagedInstance"
```

Now let us configure this **`EC2 instance`** and deploy a **`Web Server`**. Let us look at the following **`playbook`** for this and call it **`configure_ec2.yml`**:

```
---
- name: Configure EC2 Instances
  hosts: launched
  become: yes
  tasks:
    - name: Install Apache Web Server
      yum:
        name: httpd
        state: present
      when: ansible_os_family == "RedHat"

    - name: Copy index.html to Web Server
      copy:
        src: ./index.html
        dest: /var/www/html/index.html

    - name: Start Apache Service
```

```
      service:
        name: httpd
        state: started
```

This **playbook** assumes a **RedHat**-based distribution (like **CentOS** or **RHEL**). If your **EC2 instances** are running another OS, you will need to adjust the package manager (like **apt** for **Ubuntu**/**Debian**) and potentially the service name or paths.

Sample **index.html** we can use is as follows:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Welcome to Our Web Server!</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            background-color: #f9f9f9;
            color: #333;
            text-align: center;
            padding-top: 50px;
        }
    </style>
</head>
<body>
    <h1>Welcome to Our AWS-hosted Web Server!</h1>

    <p>This page is served by Apache from an EC2 instance configured by Ansible.</p>
</body>
</html>
```

Save this HTML content in a file named **index.html** in the same directory as your Ansible **playbook**.

To run the **playbook**, use:

```
ansible-playbook -i ec2.yml configure_ec2.yml
```

This assumes you are using the **ec2.yml** dynamic inventory file mentioned in the previous section. Once the **playbook** completes, accessing the public IP of your **EC2 instances** in a browser should display the HTML page we have provided.

Now let us see how we can recycle the **EC2 instances** based on their age. Let us look at the following playbook **manage_lifecycle.yml**:

```
---
- name: Manage Lifecycle of EC2 Instances
  hosts: localhost
  gather_facts: False
  vars_files:
    - vars/aws.yml
  tasks:

    - name: Fetch all EC2 instances in the region
      amazon.aws.ec2_instance_info:
        region: "{{ aws_region }}"
      register: ec2_instances

    - name: Stop EC2 instances older than 7 days
      amazon.aws.ec2:
        instance_ids: "{{ item.id }}"
        state: stopped
      loop: "{{ ec2_instances.instances }}"
      when:

        - (ansible_date_time.epoch | int) - (item.launch_time | to_datetime
| date2unix) > 604800

        - (ansible_date_time.epoch | int) - (item.launch_time | to_datetime
| date2unix) <= 1209600

    - name: Terminate EC2 instances older than 14 days
      amazon.aws.ec2:
        instance_ids: "{{ item.id }}"
        state: terminated
      loop: "{{ ec2_instances.instances }}"
      when: (ansible_date_time.epoch | int) - (item.launch_time |
to_datetime | date2unix) > 1209600
```

This **playbook** will fetch all the **EC2 instances** in the **region** specified in the **vars/aws.yml** file. It will then stop instances that are **older than 7 days** but **less than 14 days** old. Lastly, it will terminate instances that are **older than 14 days**.

Below is how `vars/aws.yml` should look like:

```
aws_region: us-west-1

aws_access_key: YOUR_AWS_ACCESS_KEY

aws_secret_key: YOUR_AWS_SECRET_KEY
```

ec2_vpc_net module

Let us now move on to the topic of VPC. Amazon VPC provides a flexible and scalable network infrastructure for your applications and services in the AWS cloud. An `ec2_vpc_net` is a module in Ansible that is used to create and manage **Virtual Private Cloud** (**VPC**) networks on AWS using Ansible. The `ec2_vpc_net` module provides a simple and flexible way to define VPC network resources as code. With this module, you can create and manage VPCs, subnets, internet gateways, route tables, network ACLs, and other related resources in a declarative way.

Some of the key features of the `ec2_vpc_net` module include:

- **Flexible network configuration:** The `ec2_vpc_net` module allows you to define your VPC network resources using a variety of parameters, such as CIDR blocks, subnet configurations, and routing rules.
- **Idempotent resource creation:** The module is designed to be idempotent, meaning that it will only create new resources if they do not already exist. This can help you ensure that your infrastructure is consistent and repeatable.
- **Resource tagging:** The `ec2_vpc_net` module supports resource tagging, which allows you to add metadata to your VPC network resources for better organization and management.
- **Support for multiple AWS accounts and regions:** The module can be used to create and manage VPC network resources across multiple AWS accounts and regions.

Other modules

Similar to the `ec2_vpc_net` module, there are other important modules that Ansible has, which can be used to provision your complete network infrastructure. Let us go over these modules briefly now. Just like before ensure you have the `amazon.aws` collection installed:

```
ansible-galaxy collection install amazon.aws
```

The `ec2_vpc_subnet`, `ec2_vpc_igw`, and `ec2_vpc_route_table` modules are all part of the `ec2_vpc` module family in Ansible, which provides a set of modules for creating and managing Amazon VPC resources using Ansible. They are explained as follows:

- `ec2_vpc_subnet`: This module is used to create, modify, and delete `VPC` subnets in Amazon Web Services. It allows you to specify the subnet's `CIDR` block, availability zone, `VPC ID`, and other optional attributes, such as tags and route table association.
- `ec2_vpc_igw`: This module is used to create, modify, and delete internet gateways in Amazon `VPC`. An internet gateway is a horizontally scaled, redundant, and highly available `VPC` component that allows communication between instances in your `VPC` and the internet. This module allows you to associate the internet gateway with a `VPC` and add tags to it.
- `ec2_vpc_route_table`: This module is used to create, modify, and delete `VPC` route tables in Amazon Web Services. A route table is a set of rules that determine where network traffic is directed within a `VPC`. This module allows you to specify the `VPC ID`, associate

the route table with one or more subnets, and add or remove routes from the route table.

Using these modules, you can automate the provisioning and management of **VPC** resources in AWS using Ansible, making it easier to create and manage complex networking configurations in a repeatable and consistent manner.

The following **playbook** creates a **VPC**, two **public** subnets, an **Internet Gateway** (**IGW**), a **Route Table**, and then associates the **Route Table** with the **public** subnets:

```
---
- name: Create AWS infrastructure
  hosts: localhost
  gather_facts: False
  vars:
    region: YOUR-REGION
    vpc_cidr: YOUR-VPC-CIDR
    public_subnet1_cidr: YOUR-SUBNET1-CIDR
    public_subnet2_cidr: YOUR-SUBNET2-CIDR

  tasks:
    - name: Create a VPC
      ec2_vpc_net:
        name: MyVPC
        state: present
        cidr_block: "{{ vpc_cidr }}"
        region: "{{ region }}"
      register: vpc

    - name: Create public subnet 1
      ec2_vpc_subnet:
        state: present
        vpc_id: "{{ vpc.vpc.id }}"
        cidr: "{{ public_subnet1_cidr }}"
        map_public: yes
        region: "{{ region }}"
      register: subnet1

    - name: Create public subnet 2
      ec2_vpc_subnet:
```

```
    state: present
    vpc_id: "{{ vpc.vpc.id }}"
    cidr: "{{ public_subnet2_cidr }}"
    map_public: yes
    region: "{{ region }}"
  register: subnet2

- name: Create IGW and attach to VPC
  ec2_vpc_igw:
    vpc_id: "{{ vpc.vpc.id }}"
    state: present
    region: "{{ region }}"
  register: igw

- name: Create a Route Table and associate subnets
  amazon.aws.ec2_vpc_route_table:
    vpc_id: "{{ vpc.vpc.id }}"
    region: "{{ region }}"
    tags:
      Name: Public
    routes:
      - dest: 0.0.0.0/0
        gateway_id: "{{ igw.gateway_id }}"
    subnets:
      - "{{ subnet1.subnet.id }}"
      - "{{ subnet2.subnet.id }}"
  register: public_route_table
```

Following is more information about this **`playbook`**:

- **Metadata and variables:**
  - The **`playbook`** is named **Create AWS infrastructure**.
  - It is intended to run locally (**`hosts: localhost`**). This means that the Ansible commands will be executed from the machine where the **`playbook`** is run, typically the control node.
  - **`gather_facts`**: False tells Ansible not to gather system facts.
  - **Four variables are declared:**
    - **`region`**: **`AWS`** region where resources will be created.

    - `vpc_cidr`: The `CIDR` block for the new `VPC`.
    - `public_subnet1_cidr`: The `CIDR` block for the first `public subnet`.
    - `public_subnet2_cidr`: The `CIDR` block for the second `public subnet`.

- **Create a VPC:**

  A new **Virtual Private Cloud** (**VPC**) named `MyVPC` will be created in the specified region and `CIDR` block. The ID and details of the created `VPC` are registered to the `vpc` variable for later use.

- **Create public subnet 1:**
    - A new `public` subnet is created within the earlier defined `VPC`.
    - The `map_public` option set to yes ensures instances launched in this `subnet` can automatically assign a `public IP`.
    - The details of this `subnet` are stored in the `subnet1` variable.
- **Create public subnet 2:**
    - Another `public subnet` is created in the `VPC`, similar to the first one but using a different `CIDR` block.
    - The details of this `subnet` are stored in the `subnet2` variable.
- **Create IGW and attach to VPC:**
    - An **Internet Gateway** (**IGW**) is created.
    - This `IGW` is then attached to the `VPC`, enabling communication between instances inside the `VPC` and the internet.
    - The details of the created `IGW` are stored in the `igw` variable.
- **Create a route table and associate subnets:**
    - A new `Route Table` is created for the `VPC`.
    - A route is added to this table to forward all traffic (`0.0.0.0/0`) to the earlier created `IGW`. This essentially provides internet access to the subnets associated with this route table.
    - This route table is then directly associated with both the created public subnets using the subnets attribute.

This `playbook` efficiently sets up a foundational `AWS` network architecture consisting of a `VPC`, two public subnets within the `VPC`, an Internet Gateway, and a route table that routes traffic from the subnets to the Internet via the `IGW`.

Storage on AWS

Let us see what Ansible provides us for managing storage on AWS. Let us talk about S3 here. **Amazon Simple Storage Service** (**Amazon S3**) is an object storage service provided by AWS. It offers highly durable, secure, and scalable storage for objects, such as files and media, in the cloud.

Following are some `S3` modules which Ansible provides:

- `amazon.aws.aws_s3`: This module is used to create, delete, or modify `S3` buckets. It allows you to specify the bucket name, region, access control settings, and other `bucket` attributes.
- `aws_s3_bucket_info`: This module is used to retrieve information about an `S3 bucket`, such as its creation date, region, and size.

- **s3_bucket_notification**: This module is used to configure **bucket** notification for an **S3 bucket**. You can configure notification for object creations, deletions, and updates, and specify the SNS topic or **Lambda** function to send notification to.
- **s3_bucket_object**: This module is used to manage **S3** objects within a **bucket**, such as uploading or downloading files, copying or moving objects, and deleting objects. It allows you to specify the object name, bucket name, file path, and other object attributes.
- **s3_sync**: This module is used to synchronize files or directories between a local system and an **S3 bucket**. It allows you to specify the source and destination paths, and supports various synchronization options such as delete, exclude, and include.

Using these modules, you can automate the management of **S3** buckets and objects in **AWS** using Ansible, making it easier to create and manage complex storage configurations in a repeatable and consistent manner.

Let us now create a **playbook** that uses some of these **S3** modules. Following, we have a **playbook** showing how we can create an **S3 bucket** with versioning and private ACL, upload a sample file to the **bucket**, download the same file from the **bucket**, copy the **object** within the **bucket**, **delete** the **object** from the **bucket**, and finally **delete** the **S3 bucket** itself:

```
---

---

- name: Manage AWS S3 bucket

  hosts: localhost

  gather_facts: False

  vars:

    region: YOUR-REGION

    bucket: BUCKET-NAME

    object: sample-file.txt

    local_file: LOCAL-FILE-PATH


  tasks:

  - name: Ensure bucket exists and has versioning enabled

    amazon.aws.s3_bucket:

      name: "{{ bucket }}"

      state: present

      versioning: yes

      region: "{{ region }}"


  - name: Upload file to the bucket

    amazon.aws.aws_s3:

      bucket: "{{ bucket }}"
```

```
      mode: put
      src: "{{ local_file }}"
      dest: "s3://{{ bucket }}/{{ object }}"
      region: "{{ region }}"

  - name: Get object details (using s3 ls command as an example, as direct
object info isn't directly provided in amazon.aws modules)
    command: "aws s3 ls s3://{{ bucket }}/{{ object }}"
    register: object_info

  - name: Print object details
    ansible.builtin.debug:
      var: object_info

  - name: Delete object from the bucket
    amazon.aws.aws_s3:
      bucket: "{{ bucket }}"
      object: "{{ object }}"
      mode: delete
      region: "{{ region }}"

  - name: Ensure bucket is absent
    amazon.aws.s3_bucket:
      name: "{{ bucket }}"
      state: absent
      region: "{{ region }}"
```

Please replace **`YOUR-REGION`**, **`BUCKET-NAME`**, and **`LOCAL-FILE-PATH`** with your actual **`AWS region`**, desired bucket name, and the local path of the file you want to upload.

This **`playbook`** creates an **`S3 bucket`** with versioning enabled, uploads a file to the bucket, gets the details of the uploaded **`object`**, deletes the **`object`**, and finally deletes the **`bucket`**. Please note that the **`s3_bucket_object`** module is only used for retrieving information about the **`object`** and deleting the **`object`**. Also, you should make sure that the **`bucket`** is empty before deleting it.

Infrastructure automation using Ansible on Azure

Let us go over the reasons why we can have infrastructure automation on Azure using Ansible:

- **Azure modules:** Ansible provides a set of Azure-specific modules that allow users to manage Azure resources such as virtual machines, networks, storage accounts, and

more.

- **Azure resource manager templates:** Ansible can be used to deploy and manage ARM templates, which are Azure's native infrastructure-as-code solutions. This makes it easy to automate the deployment of complex Azure environments.
- **Azure credentials:** Ansible allows users to store Azure credentials securely in a file, making it easy to manage access to Azure resources across multiple environments and teams.
- **Dynamic inventory:** Ansible can automatically discover and manage Azure resources, allowing users to dynamically provision and de-provision resources as needed.
- **Azure compliance:** Ansible provides a set of compliance playbooks that allow users to automate the enforcement of security and compliance policies on Azure resources.
- **Azure automation:** Ansible can be integrated with Azure automation, which is a cloud-based automation platform that provides additional features such as job scheduling, inventory management, and more.

Ansible provides a set of modules that are specifically designed for working with Azure resources. These modules allow you to automate the provisioning, configuration, and management of Azure resources using Ansible Playbooks. Here are some of the key Azure modules that Ansible provides:

- `azure_rm_virtualmachine`: This module is used to create, start, stop, restart, and delete Azure virtual machines. It can also be used to manage the configuration of virtual machines, including the operating system, network settings, and applications to be installed.
- `azure_rm_resourcegroup`: This module is used to create, update, and delete Azure resource groups. Resource groups are used to organize and manage Azure resources, and this module can be used to automate the creation of resource groups as part of a larger provisioning process.
- `azure_rm_storageaccount`: This module is used to create, update, and delete Azure storage accounts. Storage accounts are used to store data in Azure, and this module can be used to automate the creation of storage accounts as part of a larger provisioning process.
- `azure_rm_networkinterface`: This module is used to create, update, and delete Azure network interfaces. Network interfaces are used to connect virtual machines to virtual networks and this module can be used to automate the creation of network interfaces as part of a larger provisioning process.
- `azure_rm_sqlserver`: This module is used to create, update, and delete Azure SQL servers. SQL servers are used to host SQL databases in Azure, and this module can be used to automate the creation of SQL servers as part of a larger provisioning process.

Now let us use some of these modules and prepare a `playbook` as follows:

Setup an Azure virtual network named `myVirtualNetwork` with a `subnet` named `mySubnet`. The `VLAN` prefix is set to `10.0.1.0/24` and no outbound traffc is allowed from the `subnet`. Here is a breakdown of what the `playbook` is doing:

1. The `playbook` starts by authenticating with Azure using the `azure_rm_auth` module.
2. We then use the `azure_rm_virtualnetwork_info` module to get information about the virtual network and the `azure_rm_subnet_info` module to get information about the `subnet`.

3. The **azure_rm_subnet** module is used to configure the **subnet** with the specified **VLAN** prefix and network security group. The when condition is used to only run this task if **vlan_no_outbound** is set to true.
4. The **azure_rm_securitygroup_rule** module is used to update the network security group associated with the **subnet** to deny all outbound traffic. The when condition is used to only run this task if **vlan_no_outbound** is set to true.
5. Finally, the **azure_rm_networkinterface** module is used to apply the changes to the virtual network. In this **playbook**, we are thus configuring an Azure virtual network named **myVirtualNetwork** with a subnet named **mySubnet**. The **VLAN** prefix is set to **10.0.1.0/24** and no outbound traffic is allowed from the **subnet**. Refer to the following:

```
---

- name: Configure Azure virtual network hosts: localhost

connection: local gather_facts: no


vars:

rg_name: myResourceGroup vnet_name: myVirtualNetwork subnet_name: mySubnet


tasks:

name: Authenticate with Azure azure_rm_auth:

subscription_id: "<your subscription ID>" client_id: "<your client ID>"

secret: "<your client secret>" tenant: "<your tenant ID>"


name: Get virtual network information azure_rm_virtualnetwork_info:
resource_group: "{{ rg_name }}"

name: "{{ vnet_name }}" register: vnet_info


name: Get subnet information azure_rm_subnet_info:


resource_group: "{{ rg_name }}" virtual_network_name: "{{ vnet_name }}"

name: "{{ subnet_name }}" register: subnet_info


name: Configure subnet azure_rm_subnet:

resource_group: "{{ rg_name }}" virtual_network_name: "{{ vnet_name }}"

name: "{{ subnet_name }}" address_prefix: "{{ vlan_prefix }}"

network_security_group: "{{ subnet_info.subnet.network_security_group.id

}}"

when: vlan_no_outbound


name: Update network security group azure_rm_securitygroup_rule:
```

```
    resource_group: "{{ rg_name }}"
    network_security_group_name: "{{ subnet_info.subnet.network_security_group
    name: "Deny all outbound traffic" direction: "Outbound"
    access: "Deny" priority: 1000 protocol: "*"
    destination_address_prefix: "0.0.0.0/0" destination_port_range: "*"
    when: vlan_no_outbound

    name: Apply changes azure_rm_networkinterface:
    resource_group: "{{ rg_name }}" virtual_network_name: "{{ vnet_name }}" su
    name: "{{ item.name }}" public_ip_name: "{{ item.public_ip_name }}"
    security_group_name: "{{ item.security_group_name }}"
    with_items: "{{ vnet_info.virtualnetwork.properties.ip_configurations }}"
    loop_control:
    label: "{{ item.name }}"
```

Now let us provision a virtual machine on this **VLAN**. We will also deploy a MySQL server on this VLAN. Everything will be done using the Ansible Playbook. Following are the details about what each module is doing in this **playbook**:

- **azure_rm_auth**: Authenticates with Azure using the specified credentials.
- **azure_rm_virtualmachine**: Creates a new Azure virtual machine with the specified settings, including the name, size, admin username and password, location, network interface names and network interfaces.
- **azure_rm_virtualmachine_info**: Retrieves information about the specified Azure virtual machine and waits for it to come up.
- **win_chocolatey**: Installs the specified software package (in this case, SQL Server Express) on the Windows virtual machine.
- **azure_rm_sqldatabase**: Creates a new SQL Server database on the specified Azure SQL server instance with the specified settings, including the name, login credentials, and server name.

Following is how the playbook will look like:

```
---
name: Provision Azure VM and Deploy SQL Server Database hosts: localhost
connection: local gather_facts: no

vars:
rg_name: myResourceGroup vnet_name: myVirtualNetwork subnet_name: mySubnet
vlan_prefix: "10.0.1.0/24" vlan_no_outbound: true vm_name: myVM
vm_size: Standard_DS2_v2 admin_username: myadminuser admin_password:
myadminpassword sql_username: myuser
```

```
sql_password: mypassword sql_server_name: mySqlServer sql_db_name:
myDatabase


tasks:

name: Authenticate with Azure azure_rm_auth:

subscription_id: "<your subscription ID>" client_id: "<your client ID>"

secret: "<your client secret>" tenant: "<your tenant ID>"


name: Create Azure Virtual Machine azure_rm_virtualmachine:

resource_group: "{{ rg_name }}" name: "{{ vm_name }}"

vm_size: "{{ vm_size }}" admin_username: "{{ admin_username }}"
admin_password: "{{ admin_password }}"

location: "{{ vnet_info.virtualnetwork.location }}" network_interface_names:
["{{ vm_name }}-nic"] network_interfaces:

- name: "{{ vm_name }}-nic" virtual_network: "{{ vnet_name }}" subnet_name:
"{{ subnet_name }}"


name: Wait for the virtual machine to come up azure_rm_virtualmachine_info:

resource_group: "{{ rg_name }}" name: "{{ vm_name }}"

wait_for_running: yes wait_for_ssh_timeout: 300


name: Install SQL Server on the virtual machine win_chocolatey:

name: sql-server-express state: present

become: true


name: Create SQL Server Database azure_rm_sqldatabase:

resource_group: "{{ rg_name }}"

server_name: "{{ sql_server_name }}.database.windows.net" name: "{{
sql_db_name }}"

login_user: "{{ sql_username }}" login_password: "{{ sql_password }}"
```

Infrastructure automation using Ansible on Google compute cloud

For GCP, let us do something different. As we saw in AWS and Azure, we can similarly provision traditional infrastructure in GSP too, but here let us see how to provision a Kubernetes cluster using **Google Kubernetes Engine** (**GKE**).

Let us first go over the various modules:

- **`gcp_compute_network`**: This module is used to create a VPC network in Google Cloud. In this **`playbook`**, we use this module to create a network for our **`GKE cluster`**.
- **`gcp_compute_subnetwork`**: This module is used to create a subnet within a VPC network in Google Cloud. In this **`playbook`**, we use this module to create a subnet for our **`GKE`**

cluster.

- **gcp_container_cluster**: This module is used to create a **GKE cluster** in Google Cloud. This module allows you to specify various options for your cluster, such as the number of nodes, machine types, and disk sizes. In this **playbook**, we use this module to create a **GKE cluster** with a single node pool.
- **gcp_container_cluster_auth**: This module is used to retrieve the credentials for a **GKE cluster**. This module is necessary because the **k8s** module needs access to the Kubernetes API server to deploy resources.
- **k8s**: This module is used to deploy Kubernetes resources to a cluster. In this **playbook**, we use this module to deploy a sample application to our **GKE cluster**. The **src** option specifies the **YAML** file that contains the Kubernetes resources, and the namespace and name options specify the namespace and name of the resources to be deployed.

By using these modules in combination, we can provision infrastructure for a **GKE cluster** and deploy resources to it, all in a single Ansible Playbook, as shown:

```
---

name: Provision GKE cluster and deploy sample app hosts: localhost

gather_facts: no


vars:

project_id: my-project region: us-central1 zone: us-central1-a

cluster_name: my-cluster network_name: my-network subnet_name: my-subnet
node_pool_name: my-node-pool node_pool_size: 3 machine_type: n1-standard-1
disk_size: 100 app_namespace: my-app app_name: my-app


tasks:

name: Create network gcp_compute_network:

name: "{{ network_name }}" project: "{{ project_id }}"

auto_create_subnetworks: no


name: Create subnet gcp_compute_subnetwork:

name: "{{ subnet_name }}"

region: "{{ region }}" network: "{{ network_name }}"

ip_cidr_range: 10.0.0.0/24

project: "{{ project_id }}"


name: Create GKE cluster

gcp_container_cluster:

name: "{{ cluster_name }}"

zone: "{{ zone }}"
```

```
network: projects/{{ project_id }}/global/networks/{{ network_name }}
subnetwork: projects/{{ project_id }}/regions/{{ region }}/subnetworks/{{

subnet_name }} node_pools:

- name: "{{ node_pool_name }}" initial_node_count: "{{ node_pool_size }}"

autoscaling: enabled: yes

max_node_count: 5

config:

machine_type: "{{ machine_type }}" disk_size_gb: "{{ disk_size }}"
project_id: "{{ project_id }}" auth_kind: serviceaccount

service_account_file: ~/path/to/credentials.json state: present

name: Get cluster credentials gcp_container_cluster_auth:

name: "{{ cluster_name }}" project_id: "{{ project_id }}" auth_kind:
serviceaccount

service_account_file: ~/path/to/credentials.json


name: Deploy sample app k8s:

state: present src: ~/path/to/app.yaml

namespace: "{{ app_namespace }}" name: "{{ app_name }}"
```

Now, the **GKE cluster** is set up. The next step can be how to manage the distribution of traffic on the cluster. There are multiple modules in Ansible for **GCP** for setting up a **load balancer**. Following are some modules in that area:

- **gcp_compute_address**: This module is used to reserve a static external IP address for the **load balancer**. The **name** parameter specifies the name of the IP address to reserve, and the **region** parameter specifies the region where the IP address should be reserved.
- **gcp_compute_hflp_health_check**: This module is used to create an **HTTP** health check for the **load balancer**. The **name** parameter specifies the name of the health check to create, and the **port** parameter specifies the port to use for the health check.
- **gcp_compute_backend_service**: This module is used to create a backend service for the **load balancer**. The **name** parameter specifies the name of the backend service to create, and the **port_name** parameter specifies the name of the port to use for the backend service.
- **gcp_compute_url_map**: This module is used to create a **URL map** for the **load balancer**. The **name** parameter specifies the name of the **URL map** to create, and the **default_service** parameter specifies the default backend service to use for requests that do not match any of the path rules.
- **gcp_compute_target_hflp_proxy**: This module is used to create a target **HTTP proxy** for the **load balancer**. The name parameter specifies the name of the target **HTTP proxy** to create, and the **url_map** parameter specifies the URL map to use for the target **HTTP proxy**.
- **gcp_compute_global_forwarding_rule**: This module is used to create a global forwarding rule for the **load balancer**. The name parameter specifies the name of the forwarding rule to create, and the **ip_address** parameter specifies the external IP address to use for the forwarding rule. The **port_range** parameter specifies the port range to use for the forwarding rule.

- **gcp_compute_firewall_rule**: This module is used to create a firewall rule to allow incoming traffic to the **load balancer**. The name parameter specifies the name of the firewall rule to create, and the network parameter specifies the name of the network where the firewall rule should be created. The **target_tags** parameter specifies the tags to use for the firewall rule, and the allow parameter specifies the rules to allow incoming traffic.

These modules work together to create and configure a **load balancer** on a **GKE cluster**.

Let us use some of the modules discussed above and create a **load balancer** of the **GKE cluster** that we set up above. In this **playbook**, we have added new variables for the **load balancer** and target proxy name. The steps for creating a forwarding rule, target **HTTP proxy**, **URL map**, **service**, and **deployment** remain the same as before.

After creating the service with **NodePort** type, the **load balancer** can be set up using the **gcp_compute_forwarding_rule** module. The target parameter is set to the name of the target **HTTP proxy** created in the previous step.

Once the **load balancer** is created, it will distribute traffic to the nodes in the **GKE cluster** according to the rules specified in the **URL map**. Refer to the following:

```
- name: Provision GKE cluster and load balancer hosts: localhost

connection: local gather_facts: no


vars:

project_id: your_project_id zone: your_zone

cluster_name: your_cluster_name service_name: your_service_name
service_port: your_service_port

forwarding_rule_name: your_forwarding_rule_name target_proxy_name:
your_target_proxy_name


tasks:

- name: Create forwarding rule gcp_compute_forwarding_rule:

project: "{{ project_id }}"

name: "{{ forwarding_rule_name }}" region: "{{ zone|split('-')[0] }}"

ip_address: "{{ gcp_ip_address }}" protocol: TCP

load_balancing_scheme: EXTERNAL

target: "{{ target_proxy_name }}"


name: Create target HTTP proxy gcp_compute_target_http_proxy:

project: "{{ project_id }}" name: "{{ target_proxy_name }}"

url_map: "{{ service_name }}"


name: Create URL map gcp_compute_url_map:

project: "{{ project_id }}" name: "{{ service_name }}"
```

```
default_service: "{{ service_name }}"

name: Create service gcp_container_k8s:
name: "{{ service_name }}" namespace: default
state: present kind: Service definition:
apiVersion: v1
kind: Service metadata:
name: "{{ service_name }}" spec:
type: NodePort selector:
app: your_app_name ports:
- port: "{{ service_port }}" targetPort: your_container_port
protocol: TCP

name: Wait for service to become ready k8s_info:
api_version: v1
kind: Service
name: "{{ service_name }}" namespace: default
wait: yes timeout: 300

name: Create deployment gcp_container_k8s:
name: your_deployment_name namespace: default
state: present kind: Deployment definition:
apiVersion: apps/v1
kind: Deployment metadata:
name: your_deployment_name spec:
replicas: 1 selector:
matchLabels:
app: your_app_name template:
metadata: labels:
app: your_app_name spec:
containers:
name: your_container_name image: your_image_name ports:
containerPort: your_container_port env:
name: your_env_name value: your_env_value
```

Conclusion

In this chapter, we have immersed ourselves in the world of infrastructure automation using Red Hat Ansible. We explored how Ansible can be utilized to automate the provisioning and management of resources on prominent cloud platforms like AWS, Azure, and GCP. With the help of specific Ansible modules such as `ec2_vpc_net`, `ec2_vpc_subnet`, `aws_s3`, `azure_rm_subnet`, and `gcp_compute_subnetwork`, we have seen how to effectively manage and automate tasks related to `EC2`, `S3`, `VPC`, Azure virtual networks, and `GCP` subnetworks, among others.

We demonstrated the practical use of Ansible by creating playbooks to set up networking on AWS, handle operations with S3 buckets, and manage Azure and GCP infrastructure resources. Through this, we have understood the power of Ansible in turning complex, tedious, and error-prone tasks into simple, reliable, and repeatable processes.

Looking ahead, our journey with Ansible continues in the next chapter, where we will delve into the realm of network automation. We will learn how to leverage Ansible to automate network devices from various vendors. We will see how to use Ansible modules to interact with network devices, automate common network tasks, manage configurations, and ensure that our networks are state compliant. This will help us ensure a reliable, stable, and effcient network infrastructure.

By taking these steps further into Ansible’s capabilities, we are set to enhance our skills and expertise in automation. Stay tuned for an exciting journey into network automation with Ansible in our next chapter!

CHAPTER 5
Network Automation Using Ansible

Introduction

Networks are an integral part of all IT enterprises, and yet, in most places, setting up a network is manual. One of the main reasons for this is that setting up networking needs vendor-specific training. As a result, in most cases, network teams are isolated and are kept away from the DevOps revolution. This can act as a bottleneck in the journey towards infrastructure and platform modernization, as networking is only the first part. You need to configure to ensure that all downstream components work as expected. This is where Red Hat Ansible comes into play. The network teams can benefit in the following ways by using Ansible:

- Network teams can use the same simple, powerful, and agentless automation framework, that IT operations and development are already using.
- Use a data model (a playbook or role) that is separate from the execution layer (automation execution environments), and that easily spans heterogeneous network hardware.
- Benefit from a wide variety of community and vendor-generated playbook and role content, to help accelerate network automation projects.

Structure

In this chapter, we will go over the following topics:

- Gathering network information with Ansible
- Viewing system settings
- Backing up network device configurations
- Configuring the host name of the network device
- Configuring the system settings of the network device
- Real-life scenario example

Objectives

In this chapter, we will learn about network automation with a specific focus on the role and capabilities of Ansible in this domain. We explore how Ansible, a leading open-source automation tool, can empower network engineers and administrators by automating routine tasks, such as gathering host names and network information, while minimizing the potential for human error. This exploration further extends to various Ansible modules designed to gather facts from network devices, execute commands, configure settings, and perform backup operations. These modules represent a critical advantage of Ansible, as they offer a standardized and simplified interface to interact with a wide range of network devices. Through this chapter, the readers will gain a comprehensive understanding of Ansible's functionality and usability in network automation, equipping them with the knowledge to harness its full potential in their respective network environments.

Further, we delve into the diverse range of Ansible modules specifically designed for network operations. These modules, each with its distinct capabilities, allow for a broad spectrum of interactions with network devices. For instance, modules exist to gather `facts` or specific data points from network devices, enabling an informed and contextual understanding of the network state. These `facts` can range from device status, interface details, routing information, and more.

Gathering network information with Ansible

In this section, we will see how Ansible can be used to gather data about the network devices using facts, ad hoc commands and Ansible playbooks.

Naming of common network device modules

Ansible provides many platform specific modules which can be used to automate the management of these networking devices. The names of these modules follow a common pattern:

- Gather facts from networking devices using the `*os_facts` family of Ansible modules.
- Issue commands to networking devices using the `*os_command` family of Ansible modules.
- Configure networking devices using the `*os_config` family of Ansible modules.
- Configure layer 3 interfaces using the `*os_l3_interface` family of Ansible modules.

Naming of other network device modules

The `*os_facts`, `*os_commands`, `*os_config`, and `*os_l3_interface` families of modules represent a small part of the very large collection of `*os_*` modules that make it easier to manage networking devices.

Modules that gather facts on network devices

The modules in the `*os_facts` family are used to gather the following facts from networking devices:

- **cnos_facts**
- **edgeos_facts**
- **enos_facts**
- **eos_facts**
- **ios_facts**
- **junos_facts**
- **nxos_facts**
- **vyos_facts**

Modules that run commands on network devices

The modules in the ***os_command** family, are used to issue the following commands to networking devices:

- **aireos_command**
- **cnos_command**
- **edgeos_command**
- **enos_command**
- **eos_command**
- **ios_command**
- **junos_command**
- **nxos_command**
- **sros_command**
- **vyos_command**

Modules that configure network devices

The modules in the ***os_config** family, are used to configure networking devices as follows:

- **aireos_config**
- **edgeos_config**
- **enos_config**
- **eos_config**
- **fortios_config**
- **ios_config**
- **junos_config**
- **nxos_config**
- **sros_config**
- **vyos_config**

Modules that configure layer 3 interfaces

The modules in the ***os_l3_interface** family, are used to configure layer 3 interfaces on networking devices:

- **eos_l3_interface**
- **ios_l3_interface**
- **junos_l3_interface**
- **nxos_l3_interface**
- **vyos_l3_interface**

Exploring vyos_facts and iso_facts

Let us see what default sets of facts are returned by **vyos_facts** module and **ios_facts** module. For this, we can use ad hoc commands as shown:

```
ansible -i inventory --ask-vault-pass -m vyos_factsvyos
```

```
ansible -i inventory --ask-vault-pass -m ios_factsios
```

The following ad hoc command will show how many **facts** are supported by **vyos_facts** modules:

```
ansible -i inventory --ask-vault-pass -m vyos_facts spine01 > -a 'gather_subset=all'
```

Viewing system settings

In this section, let us see how we can use some ad hoc commands to view the system settings of a network device.

Let us use the following ad hoc commands on VyOS device **vyos01** to view **hostname**, **domain** and **nameservers**:

```
[pankaj@workstationproj]$ ansible -m vyos_command \
>  -a "commands='sh host name'" vyos01
[pankaj@workstationproj]$ ansible -m vyos_command \
> -a "commands='sh host domain'" vyos01
[pankaj@workstationproj]$ ansible -m vyos_command \
> -a "commands='sh dns forwarding nameservers'" vyos01
```

Now, let us run the following ad hoc commands on an IOS device **ios01** to check out the **hostname**, **domain** and **nameservers**:

```
[pankaj@workstationproj]$ ansible -m ios_command \
> -a "commands='sh run | include hostname'" ios01
[pankaj@workstationproj]$ ansible -m ios_command \
> -a "commands='sh run | include ip domain name'" ios01
```

```
[pankaj@workstationproj]$ ansible -m ios_command \
> -a "commands='sh run | include ip name-server'" iso01
```

We can set up a playbook as follows, in order to do all the above steps:

```
[pankaj@workstationproj]$ cat multivendor-dnsinfo.yaml
---
- name: multi-vendor play to display host name, domain name,
nameservers
  hosts: network

  tasks:

  - name: ios host, domain, nameserver
ios_command:
      commands:
        - sh run | include hostname
        - sh run | include ip domain name
        - sh run | include ip name-server
    register: result
    when: ansible_network_os == 'ios'

  - name: display ios host settings
    debug:
      var: result.stdout
    when: ansible_network_os == 'ios'

  - name: vyos host, domain, nameserver
vyos_command:
      commands:
        - sh host name
        - sh host domain
        - sh config | grep name-server
```

```
    register: result
    when: ansible_network_os == 'vyos'

  - name: display vyos host settings
    debug:
      var: result.stdout
    when: ansible_network_os == 'vyos'
```

Backing up network device configurations

It is easy to backup network device configurations using ***os_config** modules.

We can issue a backup manually using the ad hoc command as follows. Here, we are backing up both IOS device **ios01** and VyOS device **vyos01**:

```
$ ansible -i inventory --ask-vault-pass -m ios_config -a 'backup=yes' ios01
```

```
$ ansible -i inventory --ask-vault-pass -m vyos_config -a 'backup=yes' vyos01
```

We can issue backups using a playbook too. Let us now go over some playbooks for backing up **ios** devices and **vyos** devices respectively. We are using module **ios_config** for **ios** device backup and **vyos_config** for **vyos** device backup.

Here is the playbook for backing up **ios** devices:

```
- hosts: ios
  tasks:
    - name: Backup IOS running config
ios_config:
        backup: yes
```

Here is the playbook for backing up **vyos** devices:

```
- hosts: vyos
  tasks:
    - name: Backup VyOS running config
vyos_config:
        backup: yes
```

Backup files are named using this format:

```
<inventory_hostname> _config.yyyy-mm-dd@HH:MM:SS.
```

Configuring the host name of network device

In this section, we will see how we can use Ansible ad hoc commands to fetch the network device's **hostname**. We will also see how the Ansible Playbook can be used to set up the **hostname** for the network device.

Here, we will use the same set of modules that we used for backing up network devices, and that is ***os_config**.

Let us execute an ad hoc command to see the **hostname** for VyOS device **vyos01**:

```
[pankaj@workstationproj]$ ansible -m vyos_command  -a "commands='sh
host name'" vyos01
leaf02 | SUCCESS => {
    "changed": false,
    "stdout": [
        "vyos"
    ],
    "stdout_lines": [
        [
            "vyos"
        ]
    ]
}
```

So, we see that the hostname is **vyos** for the network device. Now, let us use the ad hoc command to update the hostname for the network device to **vyos01**. We will use **vyos_system** modules here:

```
[pankaj@workstationproj]$ ansible -m vyos_system  -a "host_name={{
inventory_hostname }}" vyos
leaf02 | SUCCESS => {
    "changed": true,
    "commands": [
        "set system host-name 'vyos'"
    ]
}
```

Let us verify if the **hostname** got updated to **vyos01** or no:

```
[pankaj@workstationproj]$ ansible -m vyos_command  -a "commands='sh
host name'" vyos
```

```
leaf02 | SUCCESS => {
    "changed": false,
    "stdout": [
        "vyos01"
    ],
    "stdout_lines": [
        [
            "vyos01"
        ]
    ]
}
```

Now we can set the **hostname** back to **vyos**:

```
[pankaj@workstationproj]$ ansible -m vyos_system  -a "host_name=vyos"
vyos01
leaf02 | SUCCESS => {
    "changed": true,
    "commands": [
        "set system host-name 'vyos'"
    ]
}
```

Now, let us write a simple playbook which will work on IOS and VyOS devices and update their **hostname** to **inventory_hostname**. Based upon if the device is **ios** or **vyos**, respective modules will be picked:

```
[pankaj@workstationproj]$ cat multi-vendor-set-hostname1.yml
---
- name: sets hostname in a multi-vendor way
  hosts: network


  tasks:


  - name: set hostname on vyos device
vyos_system:
```

```
host_name: "{{ inventory_hostname }}"
    when: ansible_network_os == 'vyos'


  - name: set hostname on ios device
ios_system:
      hostname: "{{ inventory_hostname }}"
    when: ansible_network_os == 'ios'
```

Now let us run this playbook:

```
[student@workstationproj]$ ansible-playbook -l vyos multi-vendor-set-hostname1.yml
```

Let us run ad hoc command and see if the update to **hostname** happened successfully or not:

```
[student@workstationproj]$ ansible -m vyos_command  -a "commands='sh host name'" vyos
```

Finally, let us now run the ad hoc command to revert name back to original name which is **vyos**:

```
[student@workstationproj]$ ansible -m vyos_system -a
"host_name=vyos01" vyos
leaf02 | SUCCESS => {
    "changed": true,
    "commands": [
        "set system host-name 'vyos'"
    ]
}
```

Configuring the system settings of network device

In this section, we will learn how we can configure the name server setting on the IOS devices.

Let us use the ad hoc command to fetch the current name server setting for IOS device **cs01**:

```
[student@workstationproj]$ ansible -m ios_command \
> -a "commands='sh run | include ip name-server'" cs01
cs01 | SUCCESS => {
    "changed": false,
    "stdout": [
        ""
```

```
    ],
    "stdout_lines": [
        [
            ""
        ]
    ]
}
```

So, there is no name server set up as we see. Let us use an ad hoc command to set up the **name-server** for IOS device **cs01** as **8.8.8.8**:

```
[student@workstationproj]$ ansible -m ios_system -a
"name_servers=8.8.8.8" cs01
cs01 | SUCCESS => {
    "changed": true,
    "commands": [
        "ip name-server 8.8.8.8"
    ]
}
```

As you can see, we have set up the **name-server** in previous command. Now let us verify if the name server is setup as expected or not, using the following ad hoc command:

```
[student@workstationproj]$ ansible -m ios_command \
> -a "commands='sh run | include ip name-server'" cs01
cs01 | SUCCESS => {
    "changed": false,
    "stdout": [
        "ip name-server 8.8.8.8"
    ],
    "stdout_lines": [
        [
            "ip name-server 8.8.8.8"
        ]
    ]
```

```
}
```

Let us use a playbook **ios-nameservers.yaml** for setting up two name servers for **ios** network device. We will use name servers **8.8.8.8** and **8.8.4.4** for this. Here is how our playbook will look like:

```
[student@workstationproj]$ cat ios-nameservers.yaml
---
- name: sets nameservers on ios device
  hosts: ios
  vars:
    nameservers:
    - 8.8.8.8
    - 8.8.4.4


  tasks:


  - name: set nameservers
ios_system:
name_servers: "{{ nameservers }}"
```

Now let us run the playbook:

```
[student@workstationproj]$ ansible-playbook ios-nameservers1.yml
```

Let us now verify if the **name-server** got configured as expected:

```
[student@workstationproj]$ ansible -m ios_command \
> -a "commands='sh run | include ip name-server'" cs01
cs01 | SUCCESS => {
    "changed": false,
    "stdout": [
        "ip name-server 8.8.8.8 8.8.4.4"
    ],
    "stdout_lines": [
        [
            "ip name-server 8.8.8.8 8.8.4.4"
        ]
```

```
    ]
}
```

Now let us try to configure name servers present as listed variable **nameservers**. We will have two name servers **8.8.4.4** and **8.8.8.8** in the list:

```
[student@workstationproj]$ cat group_vars/network/vars.yml
ansible_connection: network_cli
nameservers:
- 8.8.8.8
- 8.8.4.4
```

Let us set up a playbook which will use the above variable **nameservers**, and will configure them for IOS and VyOS network devices:

```
[student@workstationproj]$ cat multi-vendor-nameservers1.yml
---
- name: sets nameservers on devices
  hosts: network

  tasks:

  - name: set nameservers on ios devices
ios_system:
name_servers: "{{ nameservers }}"
    when: ansible_network_os == 'ios'

  - name: set nameservers on vyos devices
vyos_system:
name_servers: "{{ nameservers }}"
    when: ansible_network_os == 'vyos'
```

Before running this playbook, let us verify the name server configured for network device **spine02**:

```
[student@workstationproj]$ ansible -m vyos_command \
> -a "commands='sh config | grep name-server'" spine02
spine02 | SUCCESS => {
```

```
    "changed": false,
    "stdout": [
        ""
    ],
    "stdout_lines": [
        [
            ""
        ]
    ]
}
```

Now let us execute this playbook limited to device **spine02**:

```
[student@workstationproj]$ ansible-playbook -l spine02 \
> multi-vendor-nameservers1.yml
```

Let us verify if new **nameservers** got configured to **spine02** network device or not:

```
[student@workstationproj]$ ansible -m vyos_command \
> -a "commands='sh config | grep name-server'" spine02
spine02 | SUCCESS => {
    "changed": false,
    "stdout": [
        "name-server 8.8.8.8\n    name-server 8.8.4.4"
    ],
    "stdout_lines": [
        [
            "name-server 8.8.8.8",
            "    name-server 8.8.4.4"
        ]
    ]
}
```

Now let us configure the **domain name**, **hostname** and **nameserver** all together. Add **domain_name** variable to **var.yaml**:

```
[student@workstationproj]$ cat group_vars/network/vars.yml
```

```
ansible_connection: network_cli
domain_name: lab.example.com
nameservers:
- 8.8.8.8
- 8.8.4.4
```

Set up a playbook which will configure a **domain name**, **hostname** and **nameserver** for all VyOS and IOS network devices, as shown:

```
[student@workstationproj]$ cat multi-vendor-host-domain-ns1.yml
---
- name: sets host name, domain name, nameservers
  hosts: network

  tasks:

  # --- host name ---
  - name: set host name on vyos device
vyos_system:
host_name: "{{ inventory_hostname }}"
    when: ansible_network_os == 'vyos'

  - name: set host name on ios device
ios_system:
      hostname: "{{ inventory_hostname }}"
    when: ansible_network_os == 'ios'

  # --- domain name ---
  - name: set domain name on vyos device
vyos_system:
domain_name: "{{ domain_name }}"
    when: ansible_network_os == 'vyos'
```

```
  - name: set domain name on ios device
ios_system:
domain_name: "{{ domain_name }}"
    when: ansible_network_os == 'ios'

  # --- nameserver ---
  - name: set nameservers on ios devices
ios_system:
name_servers: "{{ nameservers }}"
    when: ansible_network_os == 'ios'

  - name: set nameservers on vyos devices
vyos_system:
name_servers: "{{ nameservers }}"
    when: ansible_network_os == 'vyos'
```

Let us check the syntax of the playbook:

```
[student@workstationproj]$ ansible-playbook --syntax-check \
> multi-vendor-host-domain-ns1.yml

Playbook: multi-vendor-host-domain-ns1.yml
```

Run a playbook limited to device **leaf02**. You can see the values for the **nameserver**, **domain name** and **hostname** are empty here:

```
[student@workstationproj]$ ansible-playbook -l leaf02 \
> multi-vendor-host-dnsinfo1.yml

PLAY [multi-vendor play to display host name, domain name,
nameservers] ********

TASK [ios host, domain, nameserver]
*****************************************

skipping: [leaf02]
```

```
TASK [display ios host settings]
*********************************************

skipping: [leaf02]


TASK [vyos host, domain, nameserver]
*****************************************

ok: [leaf02]


TASK [display vyos host settings]
*********************************************

ok: [leaf02] => {

    "result.stdout": [

        "vyos",

        "",

        ""

    ]

}


PLAY RECAP
*********************************************************************

leaf02                     : ok=2    changed=0    unreachable=0
failed=0
```

Run a playbook limited to device **leaf02**:

```
[student@workstationproj]$ ansible-playbook -l leaf02 \

> multi-vendor-host-domain-ns1.yml
```

Now let us see what the values for device **leaf02** are:

```
[student@workstationproj]$ ansible-playbook -l leaf02 \

> multi-vendor-host-dnsinfo1.yml


PLAY [multi-vendor play to display host name, domain name,
nameservers] ********
```

```
TASK [ios host, domain, nameserver]
*******************************************

skipping: [leaf02]


TASK [display ios host settings]
***********************************************

skipping: [leaf02]


TASK [vyos host, domain, nameserver]
*****************************************

ok: [leaf02]


TASK [display vyos host settings]
*********************************************

ok: [leaf02] => {

    "result.stdout": [

        "leaf02",

        "lab.example.com",

        "name-server 8.8.8.8\n    name-server 8.8.4.4"

    ]

}


PLAY RECAP
*********************************************************************

leaf02                     : ok=2    changed=0    unreachable=0
failed=0
```

Real-life scenario example

Example: Connecting new branch office using VyOS routers and switch configuration.

Scenario: A company is opening a new branch office and needs to set up a secure VPN connection between the headquarters and the branch. At the branch office, they have deployed a VyOS router and a few switches. The task is to automate the VPN setup on the VyOS router and configure VLANs on the switches for different departments.

Following are the steps using Ansible:

1. **Inventory creation**: First, you create an inventory file (`hosts.ini`) that lists the VyOS router and switches.

   ```
   [vyos_routers]
   branch_vyos ansible_host=10.10.10.1


   [switches]
   branch_switch1 ansible_host=10.10.10.2
   ...
   ```

2. **VyOS VPN configuration playbook**: Create an Ansible Playbook (`configure_vyos_vpn.yaml`) to set up the `VPN`.

   ```
   ---
   - name: Configure VPN on VyOS
     hosts: vyos_routers
     gather_facts: no
     tasks:
       - name: Set VPN configuration
         vyos_config:
           lines:
             - "set interfaces ethernet eth0 address '10.10.10.1/24'"
             - "set vpn ipsec ipsec-interfaces interface 'eth0'"
             ...
           provider: "{{ vyos_credentials }}"
   ```

3. **Switch VLAN configuration playbook**: This is similar to the previous example, but tailored for the branch's switches (`configure_vlan.yaml`).
4. **Variable file creation**: As before, create a variable file (`vars.yaml`) but now also include VyOS credentials.

   ```
   vyos_credentials:
     host: "{{ ansible_host }}"
     username: "vyos_admin"
     password: "vyos_password"
     authorize: yes
   ```

```
switch_credentials:
  host: "{{ ansible_host }}"
  username: "admin"
  password: "password"
  authorize: yes
```

5. **Run the playbooks**: Execute the Ansible Playbooks using the commands:

   ```
   ansible-playbook -i hosts.ini configure_vyos_vpn.yaml -e @vars.yaml

   ansible-playbook -i hosts.ini configure_vlan.yaml -e @vars.yaml
   ```

6. **Validation**: After running the playbooks, ensure configurations are correctly applied and test the VPN connection.
7. **Outcome**: The branch office is now securely connected to the headquarters through the VPN. Moreover, the switch ports at the branch office are appropriately configured for different departments. All of this is achieved with minimal manual intervention, ensuring accuracy and saving time.

   This example illustrates the flexibility of Ansible in managing different types of network devices, from traditional switches to open-source routers like VyOS.

Conclusion

In conclusion, this chapter has offered a detailed overview of the integral role of Ansible in network automation, underlining its power, versatility, and ease of use. Key to this discussion has been the broad array of Ansible modules specifically tailored for diverse network-related tasks. We have looked at the utility of various **`os_facts`** modules such as **`cnos_facts`**, **`edgeos_facts`**, **`vyos_facts`**, and **`ios_facts`**, each designed to retrieve and present a wealth of specific data points from their respective network operating systems. These facts, encompassing details such as device status, interface information, and routing configurations, provide a foundation for informed decision-making and enable proactive network management. The **`os_command`** family of modules was another focal point, bringing to light how these modules allow users to execute a broad range of commands on network devices. With these modules, network engineers can automate routine tasks, enforce uniform standards, and reduce the potential for human error.

Similarly, we examined the **`os_config`** family of modules, emphasizing their capacity to automate the deployment of configuration changes across a multitude of devices. These modules enhance both speed and accuracy while ensuring consistency, a key element of network stability and security.

Moreover, we touched on the **`os_l3_interface`** module family, which aids in automating layer 3 interface configurations. These modules enable rapid

deployment and management of network configurations at the layer 3 level, a critical aspect in large-scale network operations.

Lastly, we discussed the significance of regular and secure backups of network device configurations, and how Ansible simplifies this process. Backups, being an essential part of disaster recovery and configuration change tracking, can be automated with Ansible, ensuring regular execution and secure storage.

Looking ahead, the next chapter will extend the discussion of Ansible’s powerful automation capabilities to the domain of application automation in various public clouds such as AWS, Azure, and GCP. As organizations increasingly adopt cloud computing for its scalability, cost-effectiveness, and performance, the need for effective cloud application automation is more pronounced than ever.

In the upcoming chapter, we will unpack how Ansible can be employed to automate a multitude of tasks within these cloud environments, such as managing resources, deploying applications, and orchestrating complex multi-tier workflows. We will continue to focus on various Ansible modules designed for cloud platforms and provide insights into their utilization.

Join our book’s Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 6
App Automation Using Ansible

Introduction

In previous chapters, we have learned how Ansible can be used to automate repetitive tasks in the most reliable and error-free form. Ansible Playbooks are descriptions of the system's desired state and are kept in source control like **Git**. It uses no agents and no additional custom security infrastructure, so it is easy to deploy, and most importantly, it uses a simple language (YAML, in the form of Ansible Playbooks) that allows you to describe your automation jobs in a way that approaches plain English.

In this chapter, we will walk through the steps on how to deploy a web application on different cloud providers like **Amazon Web Service** (**AWS**), **Google Cloud Platform** (**GCP**), and **Microsoft Azure**.

Structure

In this chapter, we will learn the following topics:

- Using Ansible for app deployment on Amazon AWS
- Using Ansible app deployment on Google GCP using Ansible
- Using Ansible app deployment on Microsoft Azure using Ansible

Objectives

In this chapter, we explore Ansible's capability for app deployment across AWS, GCP, and Azure. Through real examples, readers will learn to deploy an **`EC2 instance`** with **`httpd`** on AWS, set up a RHEL8 VM with Nginx on GCP, and initiate IIS on Azure. The chapter emphasizes the significance of dynamic inventories in Ansible for managing diverse cloud environments. We will also touch on best practices in Ansible Playbook development tailored for multi-cloud deployments. By the end, readers will be adept at deploying apps across various cloud platforms using Ansible's robust features.

Using Ansible for app deployment on AWS

Setup AWS CLI: It is a good idea to have CLI for AWS client set on your machine. This will come in handy for testing and performing actions on your AWS account via CLI. To do so follow the steps:

1. Download the AWS CLI installation package for Linux using:

   ```
   Curl https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
   ```

2. You can unzip it by **`unzip awscliv2.zip`**

3. Install the AWS CLI installation package using `sudo ./aws/install`
4. Later, you can run the installer to configure using the `aws configure` command. You will be asked for `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`. After specifying them, you can then verify the AWS CLI version, as shown:

```
[ec2-user@ip-172-31-87-164 ~]$ aws --version

aws-cli/2.2.10 Python/3.8.8 Linux/4.14.133-113.105.amzn2.x86_64 botocore/2.0.0
```

This output indicates the following:

- `aws-cli/2.2.10`: This is the version of the AWS CLI that is installed.
- `Python/3.8.8`: This is the version of Python that the AWS CLI is using.
- `Linux/4.14.133-113.105.amzn2.x86_64`: This is the version of the Linux kernel on the `EC2 instance`.
- `botocore/2.0.0`: This is the version of `botocore`, which is the underlying library used by the AWS CLI.

Ansible role setup

Ansible roles are a vital aspect of your automation toolkit, ensuring the modularization and reusability of code. Properly structuring your roles not only facilitates collaboration but also makes your automation maintainable and understandable. Let us delve into some best practices:

- **Role hierarchy**: Ansible roles should be granular enough to represent one specific functionality. A clear hierarchy allows for easily identifying what each role accomplishes. For instance, separating provisioning from configuration – `ec2-provision` for instance provisioning and `webapp` for application setup.
- **Keep roles focused**: Roles should do one thing and do it well. This keeps the roles reusable and helps in troubleshooting. If you have a role called database, it should focus only on database-related tasks.
- **Use of default folders**:
  - `tasks`: This is the main list of tasks to be executed by the role. It is mandatory for a role to contain at least one task.
  - `handlers`: Handlers contain tasks that respond to a `notify` directive from other tasks, and they get executed only once, even if notified multiple times.
  - `defaults`: Contains default variables for the role. These variables have the lowest priority, meaning they can easily be overridden by variables defined elsewhere.
  - `vars`: Contains other variables for the role. They have a higher priority than defaults.
- **Role dependencies**: If one role depends on another, it is good practice to mention this in the `meta/main.yml` file. This ensures that any required role runs before the dependent role.
- **Use of README.md**: Every role should have a `README.md` file explaining the role's purpose, inputs it takes, and its outputs. This makes it easier for someone else (or even you, months later) to understand the role.
- **Always lint your roles**: Using tools like `ansible-lint` can help in identifying issues before they become problems. It checks for best practices, deprecated syntax, and potential errors.

- **Keep secrets out of roles**: Never hard-code secrets or sensitive information in roles. Use Ansible vault or environment-specific variables.
- **Use .ansibleignore**: If there are files or directories that should not be included in a role, mention them in **`.ansibleignore`**. This keeps the role clean and ensures unintended files are not executed or exposed.
- **Version your roles**: If you are using Ansible Galaxy or any other SCM system, versioning helps in rolling back, deploying specific versions, and understanding changes over time.

Remember, the structure and organization of Ansible's roles are as crucial as the tasks they execute. Well-organized roles can save time, reduce errors, and make collaborations smoother.

Let us see and understand this with real examples of Ansible roles in the following section.

First, let us see if Ansible is installed on your host. Here we are on the RHEL8 **`EC2`** node. In config, you can see we are verifying the Ansible version using **`ansible –version`** command. Later we will create the **`host`** file and **`site.yml`** ansible playbook along with the **`roles`** directory. Under this **`roles`** directory, we will create roles that the **`site.yml`** playbook can call:

```
ansible 2.9.21

  config file = /etc/ansible/ansible.cfg

  configured module search path = [u'/path/to/modules']

  ansible python module location = /usr/lib/python2.7/site-packages/ansible

  executable location = /usr/bin/ansible

  python version = 2.7.5 (default, May  3 2021, 12:49:01) [GCC 4.8.5
20150623 (Red Hat 4.8.5-44)]


touch site.yml hosts > hosts && mkdir -p roles
```

Below you can see that two Ansible roles, using **`ansible-galaxy`** command, are being created. Role **`ec2-provision`** will provision **`EC2 instance`** using your AWS account. Once the **`EC2`** is provisioned, it will run the ansible **`webapp`** role to deploy the **`httpd`** server and deploy the **`index.html`** page:

`cd roles`

`ansible-galaxy init ec2-provision`

Role **`ec2-provision`** was created successfully.

`ansible-galaxy init webapp`

Role **`webapp`** was created successfully.

Your role structure will be as shown below:

`roles`

`├── ec2-provision`

`│   ├── README.md`

`│   ├── defaults`

`│   │   └── main.yml`

```
│   ├── files
│   ├── handlers
│   │   └── main.yml
│   ├── meta
│   │   └── main.yml
│   ├── tasks
│   │   └── main.yml
│   ├── templates
│   ├── tests
│   │   ├── inventory
│   │   └── test.yml
│   └── vars
│       └── main.yml
└── webapp
    ├── README.md
    ├── defaults
    │   └── main.yml
    ├── files
    ├── handlers
    │   └── main.yml
    ├── meta
    │   └── main.yml
    ├── tasks
    │   └── main.yml
    ├── templates
    ├── tests
    │   ├── inventory
    │   └── test.yml
    └── vars
        └── main.yml

14 directories, 13 files
```

Setup the **host** file. The playbook is targeted to run on a local **EC2** machine, so **localhost** is specified as the host. Additionally, the path to Python is specified using the **which python** command. Below **host** file shows the **host** file, as follows:

```
[local]
```

```
localhost ansible_connection=local

[all:vars]

ansible_python_interpreter=/usr/bin/python3
```

In this version of the **host** file:

- **[all:vars]** is a special group name to specify variables that apply to all hosts.
- **ansible_python_interpreter=/usr/bin/python3** sets the path to the Python interpreter for all hosts. Replace **/usr/bin/python3** with the actual path to **Python3** on your system.
- **[local]** is the group name to which **localhost** belongs.
- **localhost** is the name of the host (**local EC2** machine).
- **ansible_connection=local** specifies that the playbook should run locally on the control machine.

If you have VMs in different clouds then the **host** file can look like:

```
[local]

localhost ansible_connection=local


[all:vars]

ansible_python_interpreter=/usr/bin/python3


[AWS]

aws_instance_1 ansible_host=ec2-xx-xx-xx-xx.compute-1.amazonaws.com

aws_instance_2 ansible_host=ec2-yy-yy-yy-yy.compute-1.amazonaws.com


[AWS:vars]

ansible_ssh_user=ec2-user

ansible_ssh_private_key_file=/path/to/aws/private/key.pem


[GCP]

gcp_instance_1 ansible_host=xx.xx.xx.xx

gcp_instance_2 ansible_host=yy.yy.yy.yy


[GCP:vars]

ansible_ssh_user=gcp-user

ansible_ssh_private_key_file=/path/to/gcp/private/key.json


[Azure]

azure_vm_1 ansible_host=vmxx.centralus.cloudapp.azure.com

azure_vm_2 ansible_host=vmyy.centralus.cloudapp.azure.com
```

```
[Azure:vars]

ansible_ssh_user=azure-user

ansible_ssh_private_key_file=/path/to/azure/private/key.pem
```

In the preceding **host** file we see:

- For each cloud provider, instances (or VMs) are organized under specific group names: **[AWS]**, **[GCP]**, and **[Azure]**.
- Specific variables relevant to each cloud provider (like SSH user or private key file location) are defined under each cloud group's **vars** section. This keeps configurations clear and avoids mixing cloud-specific configurations.

Remember, in a real-world scenario, you would replace placeholders like **ec2-xx-xx-xx-xx.compute-1.amazonaws.com** or **xx.xx.xx.xx** with your actual instance hostnames or IPs.

Now, let us set up the playbook. The following section shows the basic playbook, which will call ansible role **ec2-provision**:

```
---

- name: EC2 Provisioning Playbook

  hosts: local

  become: yes

  pre_tasks:

      - name: Gathering Facts

        setup:


  roles:

    - ec2-provision
```

Explanation of the playbook:

- **name: EC2 Provisioning Playbook**: This sets the name of the playbook as **EC2 Provisioning Playbook**. It is just a human-readable identifier for the playbook.
- **hosts: local**: This specifies that the playbook should run on the hosts in the group called **local**, which, in this case, is the **localhost** host specified in the **hosts** file.
- **become: yes**: This enables privilege escalation, which means the playbook will run with administrative privileges (using **sudo** or a similar mechanism) when necessary.
- The **pre_tasks** section is used to gather facts explicitly using the setup module.
- **roles**: This is the list of roles that the playbook should include and execute. In this case, it includes the **ec2-provision** role.

Let us now setup the **ec2-provision** role. From the **ec2-provision** directory modify the **file tasks/main.yaml** with file content:

Note: Make sure you specify your own public key under key_material

```
---

# tasks/main.yml
```

```
- name: Create VPC
  ec2_vpc_net:
    name: my_vpc
    state: present
    cidr_block: 10.0.0.0/16
    region: us-east-1
  register: my_vpc

- name: Create Restrictive Security Group
  ec2_group:
    name: my_restricted_security_group
    description: My Restrictive Security Group
    rules:
      - proto: tcp
        ports:
          - 22
        cidr_ip: 10.0.0.0/16
    vpc_id: "{{ my_vpc.vpc.id }}"
  register: restrictive_security_group

- name: Create Security Group for Public Access
  ec2_group:
    name: my_security_group
    description: My Security Group
    rules:
      - proto: tcp
        ports:
          - 22
          - 80
        cidr_ip: 0.0.0.0/0
    vpc_id: "{{ my_vpc.vpc.id }}"
  register: security_group

- name: Create EC2 Key Pair
  ec2_key:
```

```
    name: my_key_pair
    state: present
  register: key_pair
- name: Create EC2 Instance
  ec2:
    key_name: "{{ key_pair.key.name }}"
    instance_type: t2.micro
    image: "ami-0c55b159cbfafe1f0"
    group_id:
      - "{{ security_group.group_id }}"
      - "{{ restrictive_security_group.group_id }}"
    region: us-east-1
    vpc_subnet_id: "{{ my_vpc.subnets[0].id }}"
    count: 1
    instance_tags:
      Name: My EC2 Instance
  register: ec2_instance
```

Explanation of the tasks:

- **`Create VPC`**: This task creates a new **`VPC`** with a **`CIDR`** block of **`10.0.0.0/16`**.
- **`Create Restrictive Security Group`**: A security group that only allows **`SSH`** traffic from within the **`VPC`**.
- **`Create Security Group for Public Access`**: A security group that allows **`SSH`** and **`HTTP`** traffic from anywhere.
- **`Create EC2 Instance`**: We have now also specified the VPC subnet ID to ensure the **`EC2 instance`** is launched within the newly created **`VPC`**.

Let us then set up the variables for the **`ec2-provision`** role. From the **`ec2-provision`** location edit **`the vars/main.yaml`** file with content as shown below.

Here, **`us-east-1`** is being used as the region as shown:

```
---
# roles/ec2-provision/vars/main.yml

aws_region: us-east-1

# EC2 Specific
ec2_instance_type: t2.micro
ec2_image: ami-0c55b159cbfafe1f0
ec2_key_name: my_key_pair
```

```
ec2_instance_tags:
  Name: My EC2 Instance

# VPC Specific
vpc_name: my_vpc
vpc_cidr_block: 10.0.0.0/16

# Security Groups
ec2_security_group_name: my_security_group
ec2_security_group_description: My Security Group
ec2_restricted_security_group_name: my_restricted_security_group
ec2_restricted_security_group_description: My Restricted Security Group

ec2_security_group_rules:
  - proto: tcp
    ports:
      - 22
      - 80
    cidr_ip: 0.0.0.0/0

ec2_restricted_security_group_rules:
  - proto: tcp
    ports:
      - 22
    cidr_ip: 10.0.0.0/16

# IAM roles (if needed)
iam_role_name: my_ec2_iam_role

# Predefined IDs (if any)
predefined_vpc_id: vpc-xxxxxxxxxxxxx
predefined_security_group_ids:
  - sg-xxxxxxxxxxxxx
  - sg-yyyyyyyyyyyyy
```

Explanation of the variables

Following is the explanation of the variables:

VPC specific variables:

- `vpc_name`: The name you would like to give to the `VPC`.
- `vpc_cidr_block`: The IP address range for the `VPC` in `CIDR` notation. This determines the size and range of the IP address pool for the `VPC`.

Security groups:

- `ec2_restricted_security_group_name.` **and** `ec2_restricted_security_group_description`: These are variables for a new, more restricted security group. This might be useful if you have some `EC2 instances` that need tighter security.
- `ec2_restricted_security_group_rules`: Defines the rules for this restricted security group. In the provided example, only `SSH` (`port 22`) is allowed, and only from IPs in the `10.0.0.0/16` range.

IAM roles:

- `iam_role_name`: If you have an `EC2 instance` that needs to interact with other AWS services (for example, read from an `S3 bucket` or write to a `DynamoDB` table), you would typically assign an IAM role to that `EC2 instance`. This role will have the permissions required to interact with those services. The `iam_role_name` variable is the name of that IAM role.

Predefined IDs:

- `predefined_vpc_id`: There might be situations where you do not want to create a new `VPC`, but instead, want to use an existing one. In such cases, you can specify the ID of that `VPC` here and then reference it in your tasks.
- `predefined_security_group_ids`: Similar to the VPC ID, if you have existing security groups you would like to use, you can list their IDs here.

This structure promotes a best practice in Ansible called **variable precedence**. By defining variables in your role's `vars/main.yml`, you are giving users of the role the ability to override these variables at a higher precedence level if they need to (for example, at the playbook level, or via `host_vars` or `group_vars`). This makes your role flexible and more adaptable to different scenarios or environments.

Now, let us set the role `webapp`. From the `webapp` directory edit file `tasks/main.yaml` with content as shown in the following section. So, you can see that we are installing the `httpd` package, setting service for `http`, and installing the `index.html` page:

```
# roles/webapp/tasks/main.yml


- name: Install httpd package
  yum:
    name: httpd
    state: present


- name: Start and enable httpd service
  service:
    name: httpd
    state: started
    enabled: yes
```

```
- name: Create index.html page
  copy:
    content: "<html><body><h1>Hello, World!</h1></body></html>"
    dest: /var/www/html/index.html

- name: Install mod_ssl for httpd
  yum:
    name: mod_ssl
    state: present

- name: Obtain a Let's Encrypt SSL certificate
  include_role:
    name: geerlingguy.certbot
  vars:
    certbot_create_if_missing: yes
    certbot_certs:
      - domains:
          - 'example.com'
        email: 'webmaster@example.com'

- name: Configure httpd for SSL
  template:
    src: ssl.conf.j2
    dest: /etc/httpd/conf.d/ssl.conf
  notify:
    - Restart httpd

# Here you can add tasks for setting up authentication,
# like setting up Basic Auth or integrating with LDAP etc.

# Handlers section
handlers:
  - name: Restart httpd
    service:
      name: httpd
      state: restarted
```

Explanation of the tasks

Following is the explanation of the tasks:

- **`Install httpd package`**: This task uses the **`yum`** module to install the **`httpd`** package on the target system.
- **`Start and enable httpd service`**: This task uses the **`service`** module to start and enable the **`httpd`** service on the target system.
- **`Create index.html page`**: This task uses the **`copy`** module to create an **`index.html`** file in the **`/var/www/html`** directory with the content **`<html><body><h1>Hello, World!</h1></body></html>`**.
- **`Install mod_ssl for httpd`**: This task installs the **`mod_ssl`** module to provide **`SSL`**/**`TLS`** support for Apache.
- **`Obtain a Let's Encrypt SSL certificate`**: This task leverages the **`geerlingguy.certbot`** Ansible role to obtain and manage SSL certificates from **`Let's Encrypt`**. The role handles the installation of Certbot, certificate renewal, and can also obtain certificates if configured.
- **`Configure httpd for SSL`**: This task deploys an Apache SSL configuration template (**`ssl.conf.j2`**) to the target server. Any modification to this configuration will trigger the **`Restart httpd`** handler to apply the changes.

Remember, the **`ssl.conf.j2`** will be a **`Jinja2`** template that contains the necessary configurations for Apache to work with SSL. Ensure you have it in your **`template`** directory of the role.

Furthermore, for the Certbot role to work, you would need to include it in your project. You can use Ansible Galaxy to install this role (**`ansible-galaxy install geerlingguy.certbot`**) or add it to your **`requirements.yml`** file.

With these tasks defined in the **`tasks/main.yml`** file, the **`webapp`** role will be responsible for installing and setting up the Apache web server (**`httpd`**) and creating the **`index.html`** page with the specified content. Once you have set up both the **`ec2-provision`** and **`webapp`** roles with their respective tasks and variables, you can include them in your main playbook (**`site.yml`**) to deploy the web application on the provisioned **`EC2 instance`**.

Let us create a basic **`index.html`** page which this **`httpd`** webserver can use. As shown in the following section:

```
<!-- index.html -->


<!DOCTYPE html>

<html>

<head>

    <title>Welcome Everyone</title>

</head>

<body>

    <h1>Welcome to Blog on Multi Cloud With Ansible</h1>

</body>

</html>
```

This **`index.html`** file includes a title **`Welcome Everyone`** and a heading **`Welcome to Blog on Multi Cloud With Ansible`** to be displayed on the web page when served by the Apache web server. Save

this content in a file named **index.html** and place it in the **files** directory of your **webapp** role. This will allow the **copy** task in the **tasks/main.yml** file to copy the **index.html** file to the **/var/www/html/** directory on the target system when the role is executed.

Setup dynamic inventory file

Since in a public cloud like AWS, the host is dynamic, hence maintaining a static host file might not be a great approach. Here, we can manage dynamic host files, as discussed in previous chapters. These host files will be used by Ansible role **webapp** to get **EC2 instance** details where **httpd webapp** will be deployed. Let us set that up.

Let us make a directory **mkdir dynamic_host**. Under that directory we will download the **ec2.py** and **ec2.ini** files. You can use **wget** to download these files.

```
wget https://raw.githubusercontent.com/ansible/ansible/main/contrib/inventory/ec2.py
```

```
wget https://raw.githubusercontent.com/ansible/ansible/main/contrib/inventory/ec2.ini
```

In **ec.py** file you can modify the values for **AWS_ACCESS_KEY_ID**, **AWS_SECRET_ACCESS_KEY** and **EC2_INI_PATH**. Once that is done, you can test it using the **python ec2.py –list** command. This will provide you with all instance details under your AWS account.

Note: For this to work, ensure you ran the AWS configure step, as discussed earlier in this chapter.

Running the Ansible Playbook

Now, let us provision the **EC2 instance** by running the **ansible-playbook -i hosts site.yml** command as shown:

```
PLAY [EC2 Provisioning]
*********************************************************************


TASK [Gathering Facts]
*********************************************************************

ok: [localhost]


TASK [Create Security Group]
*********************************************************************

changed: [localhost]


TASK [Create EC2 Key Pair]
*********************************************************************

changed: [localhost]


TASK [Create EC2 Instance]
*********************************************************************

changed: [localhost]


PLAY [WebApp Deployment]
*********************************************************************
```

```
TASK [Gathering Facts]
*********************************************************************

ok: [localhost]


TASK [Install httpd package]
*************************************************************

changed: [localhost]


TASK [Configure httpd service]
*********************************************************

changed: [localhost]


TASK [Copy index.html file]
*************************************************************

changed: [localhost]


PLAY RECAP
*********************************************************************************
***

localhost                  : ok=8    changed=6    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0
```

Optional:

It is optional to add a debug task in the playbook **site.yaml**.

```
- name: Print all gathered facts

  debug:

    var: ansible_facts
```

If you add the debug task to print all the gathered facts to your **site.yml** playbook, the output will be extensive since **ansible_facts** contains a lot of information about the target host. Following is how the output will look like:

```
PLAY [all] *********************************************************************


TASK [Print all gathered facts] ************************************************

ok: [your_target_host] => {

    "ansible_facts": {

        "ansible_all_ipv4_addresses": [...],

        "ansible_architecture": "...",

        "ansible_bios_date": "...",

        "ansible_bios_version": "...",

        "ansible_cmdline": {...},
```

```
        "ansible_date_time": {...},
        "ansible_default_ipv4": {...},
        "ansible_default_ipv6": {...},
        "ansible_distribution": "...",
        "ansible_distribution_version": "...",
        ...
        ...
        ... (many more facts)
    }
}
```

Once the **EC2** is provisioned you can run the command **ansible-playbook -i dynamic_host/ec2.py site.yml** to call the Ansible **webapp** role for installing the **httpd** package, starting the **httpd** service and then deploying the **index.html** page. Following is how the output should look like:

```
PLAY [Provision EC2 and Configure Web Server]
*************************************


TASK [Gathering Facts]
*************************************************************

ok: [localhost]


TASK [ec2-provision : Create Security Group]
************************************

changed: [localhost]


TASK [ec2-provision : Create EC2 Key Pair]
************************************

changed: [localhost]


TASK [ec2-provision : Launch EC2 Instance]
************************************

changed: [localhost]


TASK [webapp : Install httpd package]
*****************************************

changed: [ec2-123-45-67-89.compute-1.amazonaws.com]


TASK [webapp : Start httpd service]
*****************************************

changed: [ec2-123-45-67-89.compute-1.amazonaws.com]
```

```
TASK [webapp : Copy index.html]
*********************************************

changed: [ec2-123-45-67-89.compute-1.amazonaws.com]


PLAY RECAP
*********************************************************************

ec2-123-45-67-89.compute-1.amazonaws.com : ok=5    changed=4
unreachable=0    failed=0    skipped=0    rescued=0    ignored=0

localhost                  : ok=1    changed=0    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0
```

Using Ansible for app deployment on GCP

Let us deploy the NGINX webserver on the **Google Cloud Platform** (**GCP**) to serve our landing page using Ansible in this section:

1. First, we need to create a **VM instance** from our **GCP** console. The steps have been covered in previous chapters. Here **RHEL8 VM** is being used, and you can see **instance-1** as shown in *Figure 6.1*:

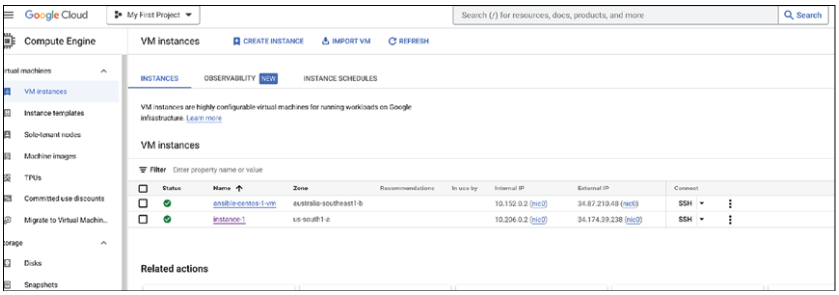

Figure 6.1: *VM Instances on GCP console*

2. Download the sample code from the public GIT repository using **`git clone https://github.com/do-community/html_demo_site.git`**.

   Verify if Ansible is installed on RHEL8 VM. If not, then install that first as discussed earlier. Use the **`ansible –version`** to verify **`ansible`** details on the VM as shown:

   ansible 2.9.27:

   ```
   config file = /etc/ansible/ansible.cfg

   configured module search path = ['/usr/share/ansible']

   ansible python module location = /usr/lib/python3.6/site-packages/ansibl

   executable location = /usr/bin/ansible

   python version = 3.6.8 (default, Nov 16 2020, 16:55:22) [GCC 8.3.1 2019]
   ```

3. Create the host or inventory file for Ansible

   In our setup, we have defined the Google VM instances in the inventory file under the **`[gcp_servers]`** group, preparing them for the NGINX Webserver installation. Here is how the inventory file is structured:

   ```
   [gcp_servers]
   ```

```
gcp-instance-1 ansible_host=10.128.0.2 ansible_ssh_user=your_ssh_user
gcp-instance-2 ansible_host=10.128.0.3 ansible_ssh_user=your_ssh_user

[all:vars]
ansible_python_interpreter=/usr/bin/python3
ansible_ssh_private_key_file=/path/to/your/private/key.pem
```

Explanation of inventory file.

- **`gcp_servers`**: This defines a group of GCP servers.
- **`gcp-instance-1` and `gcp-instance-2`**: These are the names you have given to the instances in your inventory. You can give any name as per your preference.
- **`ansible_host`**: This indicates the IP address of the GCP instance.
- **`ansible_ssh_user`**: This is the username used to SSH into the GCP instance.
- **`ansible_python_interpreter`**: It tells Ansible which Python interpreter to use.
- **`ansible_ssh_private_key_file`:** Path to the private key file you will use for authentication. This is especially useful if you are not using the default SSH key to connect.

For your application, please substitute **`gcp-instance-1`** and **`gcp-instance-2`** with appropriate names for your VM instances and adjust the **`ansible_host`** IP addresses and **`ansible_ssh_user`** accordingly. Also, ensure that the **`ansible_ssh_private_key_file`** points to the correct path of your private SSH key. If you are unsure about the path to Python 3 on your system, use the command **`python3`** to determine it.

You can verify the **`hosts`** file or inventory using command **`ansible-inventory –list -y`** command as shown:

```
all:
  hosts:
    gcp-instance-1:
      ansible_host: 10.128.0.2
      ansible_ssh_user: your_ssh_user
    gcp-instance-2:
      ansible_host: 10.128.0.3
      ansible_ssh_user: your_ssh_user
  vars:
    ansible_python_interpreter: /usr/bin/python3
    ansible_ssh_private_key_file: /path/to/your/private/key.pem
  children:
    gcp_servers:
      hosts:
```

```
gcp-instance-1:
gcp-instance-2:.
```

4. Setup NGINX playbook for installing and configuring NGINX

   In the following section, you can see the Ansible Playbook being used. Following are some points about this playbook which will help you understand the playbook better:

   - In the host file, **`all`** is used as the host. Currently, only **`localhost`** is listed in the **`host`** file, but if needed, additional hosts can be added in the future without making any changes to this playbook.
   - The connection type specified is **`local`** since **`NGINX`** is being installed on the same host where the playbook is being executed. This eliminates the need for an **`SSH`** call and avoids SSH key exchange.
   - The variables for the application root and document root are defined, which are then utilized by downstream tasks.
   - **`Yum`** is used to install the NGINX package.
   - **`Copy`** module is used to copy the landing page file.
   - **`Template`** module is used to and define and get the NGINX configuration.

   `Handler` is used to restart the NGINX service. This handler is called if the

```
---
- name: Setup NGINX on GCP VM
  hosts: localhost
  gather_facts: false

  vars:
    nginx_config_file: /etc/nginx/nginx.conf
    landing_page_src: /path/to/landing_page/index.html
    landing_page_dest: /usr/share/nginx/html/index.html

  tasks:
    - name: Install NGINX package
      become: true
      yum:
        name: nginx
        state: present

    - name: Copy landing page file
      become: true
      copy:
```

```
        src: "{{ landing_page_src }}"
        dest: "{{ landing_page_dest }}"
        owner: root
        group: root
        mode: 0644

    - name: Define and get NGINX configuration
      become: true
      template:
        src: nginx.conf.j2
        dest: "{{ nginx_config_file }}"
        owner: root
        group: root
        mode: 0644
      notify: Restart NGINX

  handlers:
    - name: Restart NGINX
      become: true
      service:
        name: nginx
        state: restarted
```

In this playbook, you need to create a **`nginx.conf.j2`** template file that defines your **`NGINX configuration`**. The **`template`** file may look like:

```
worker_processes 1;
error_log /var/log/nginx/error.log;
events {
    worker_connections 1024;
}
http {
    include /etc/nginx/mime.types;
    default_type application/octet-stream;
    sendfile on;
    keepalive_timeout 65;
    server {
```

```
        listen 80;
        server_name localhost;
        location / {
            root /usr/share/nginx/html;
            index index.html;
        }
    }
}
```

5. Now let us execute the playbook.

   You can see the Ansible Playbook getting executed using **`ansible-playbook nginx-playbook.yaml`** command. Here any **`hosts`** or **`inventory`** file is not being specified so it will pick the default **`hosts`** file at **`/etc/ansible/hosts`**.

```
PLAY [Setup NGINX Webserver] ***************************************************

TASK [Install NGINX Package]
**********************************************

changed: [localhost]

TASK [Copy Landing Page]
************************************************************

changed: [localhost]

TASK [Define NGINX Configuration]
*******************************************

changed: [localhost]

RUNNING HANDLER [Restart NGINX Service]
****************************************

changed: [localhost]

PLAY RECAP
*********************************************************************

localhost                  : ok=4    changed=4    unreachable=0    failed=
```

Using Ansible for app deployment on Azure

Here, we will deploy the IIS Webserver on Azure Windows Server. Like **`*nix`** systems, Ansible provides multiple modules for Windows system, normally those are prefixed with **`win_`**. In this section we will use multiple such modules, as described:

- Installing **`IIS`** with **`win_feature:`**

This module can be compared to **yum** or **apt** which we use in ***nix** systems to deploy packages. Here we are using **win_feature** to install IIS for hosting web applications. Most parameters used for this module are similar to those of **yum** or **apt** but some new parameters are **include_management_tooks** and **include_sub_features** which are used for installing management tools for **IIS** and sub-features for **IIS**, respectively.

The following Ansible code shows the use of the **win_feature** module. Install **IIS** (also known as **Internet Information Services**): It installs the **Web-Server** feature on Windows hosts. The **state: present** ensures that the feature is installed:

```
---
- name: Install IIS (Internet Information Services) on Windows
  hosts: windows_hosts
  tasks:
    - name: Install IIS
      win_feature:
        name: Web-Server
        state: present
```

- Setting index page for **IIS** using **win_copy:**

This module is similar to the **copy** module we have used numerous times in ***nix** systems. In our playbook we will use this for copying the **index.html** to provide the landing page for the **IIS** webserver.

Following is the sample HTML page which can serve as a landing page:

```
<!DOCTYPE html>
<html>
<head>
    <title>Using Azure with Ansible</title>
</head>
<body>
    <h1>Using Azure with Ansible</h1>
    <p>Ansible was here</p>
</body>
</html>
```

The following section will show, how we use the **win_copy** module to deploy or copy the landing page for the **IIS** webserver:

```
---
- name: Set Index Page for IIS
  hosts: windows_hosts
  tasks:
```

```
    - name: Copy index.html to IIS webroot
      win_copy:
        src: /path/to/index.html  # Replace with the actual path to your index.html file
        dest: C:\inetpub\wwwroot\index.html
      become: yes  # Ensure that Ansible has administrative privileges to copy the file
```

- Using **win_choclatey** to install **dotnetCore** runtime:

  We will be installing **dotnetcore** runtime using the **win_chocolatey** module. **chocolatey** is a package manager for Windows which simplifies the process of installing software packages in windows instances. Following is a sample code of how we will use **win_chocolatey** for our purpose and you can look at some parameters being used to specify the version (**version**) and arguments (**install_args**). Once the package is installed, we will use a **handler** from Ansible to restart **IIS** as shown:

```
- name: Install .NET Core Runtime
  win_chocolatey:
    name: dotnetcore-runtime
    state: present
  notify: Restart IIS

  handlers:
  - name: Restart IIS
    win_shell: iisreset
    when: iis_service is changed
```

  In this task, we are using the **win_chocolatey** module to install the **.NET Core Runtime** on the Windows host. The name parameter specifies the package name (**dotnetcore-runtime** in this case), and **state: present** indicates that we want to ensure the package is installed on the host.

  In this example, the first task installs the **.NET Core Runtime** using the **win_chocolatey** module. The second task ensures that the **IIS** service is running and set to start automatically on boot. The third task copies the **index.html** file to the **IIS wwwroot** directory.

  The notify keyword in the third task triggers the **Restart IIS** handler if the task changes the **IIS configuration**. The handler uses the **win_shell** module to execute the **iisreset** command, which restarts the **IIS** service.

- Using **win_file** to create logs files:

  Next, we will specify the directory where **IIS** will store its log files. For this, we are using the **win_file** module. Here, the main parameters are **path** and **state** to specify the location of **log** files and **directory** type respectively. Here directory means folders for Windows systems.

```
- name: Create logs directory
```

```
    win_file:
      path: C:\logs
      state: directory
```

- Using Ansible with **`PowerShell`** and variables:

  As we all know by now, Ansible maintains a list of servers using **`hosts`** or **`inventory`** files. Maintaining these can be additional tasks for teams who want to run automation in smaller numbers. They have options to use **`PowerShell`** to specify the host name where this automation needs to be run. This is just optional, and you can still use **`hosts`** and **`inventory`** files.

  In the next section, we are specifying the variables for our automation:

  Note: You should not store sensitive information like passwords in plain text. You can use var_prompt which asks you about that parameter when you run the playbook or you can use ansible_vault to encrypt the password.Using var_prompt: You can prompt for sensitive data interactively during playbook execution using the var_prompt feature. This way, the sensitive data will not be stored in plain text in the playbook. Here's an example of how to use var_prompt:

```
  vars_prompt:
    - name: my_password
      prompt: "Enter your password"
      private: yes

  tasks:
    - name: Print password
      debug:
        var: my_password
```

- Running the playbook for **`IIS`** webserver install on Azure:

Following is what your final playbook will look like. You can name it anything, here, it is named as **`iisInstallPlaybook.yaml`**. For example, if you want to run it on a Windows host on Azure whose IP address is **`126.98.98.1`**, the command used for execution will be:

```
ansible-playbook iisInstallPlaybook.yaml -I 126.98.98.1
```

```
---
- name: Install IIS and .NET Core Runtime
  hosts: windows_servers
  gather_facts: yes
  become: yes

  vars_prompt:
    - name: iis_admin_username
      prompt: "Enter IIS Administrator Username"
      private: no
```

```
    - name: iis_admin_password
      prompt: "Enter IIS Administrator Password"
      private: yes

  tasks:
    - name: Install IIS feature
      win_feature:
        name: Web-Server
        state: present

    - name: Install .NET Core Runtime
      win_chocolatey:
        name: dotnetcore-runtime
        state: present

    - name: Set up index page for IIS
      win_copy:
        src: index.html
        dest: C:\inetpub\wwwroot\index.html

    - name: Configure IIS to restart on change
      win_shell: |
        Set-WebConfigurationProperty -Filter
/system.applicationHost/applicationPools/applicationPoolDefaults -PSPath
IIS:\ -Name "startMode" -Value "AlwaysRunning"
      register: iis_config_changed
      changed_when: iis_config_changed.rc == 0

  handlers:
    - name: Restart IIS
      win_service:
        name: W3SVC
        state: restarted
```

In this playbook, we first prompt for the **`IIS administrator username`** and **`password`** using **`vars_prompt`**. Then, we proceed with the tasks:

- Install the **`IIS`** feature using the **`win_feature`** module.
- Install the **`.NET Core Runtime`** using the **`win_chocolatey`** module.

- Set up the index page for `IIS` using the `win_copy` module, which copies the `index.html` file to the specified location.
- Configure `IIS` to restart changes using the `win_shell` module. We register the output of this task to the variable `iis_config_changed`.
- Finally, we define a handler called `Restart IIS` that will restart the `IIS` service if there is a change in the configuration.
- Please note that the `index.html` file should be placed in the same directory as the playbook and should contain the HTML content for the index page.

Conclusion

In this chapter, we delved into the immense capabilities of Ansible in deploying applications across diverse cloud providers, namely AWS, GCP, and Azure. The simplicity and adaptability of Ansible earmark it as an exceptional choice for automating deployment processes and orchestrating infrastructure across these cloud platforms.

Initially, we established the groundwork by setting up Ansible locally, understanding its versions, and grasping basic configurations. We journeyed through creating inventory files that spotlighted target `hosts`, encompassing AWS EC2 instances, GCP VMs, and Azure virtual machines. As we advanced, the versatility of Ansible Playbooks for automating application deployments came to the forefront. We highlighted modules like `ec2` for AWS and `gcp_compute_instance` for GCP, showcasing the creation of VM instances.

Comparison of cloud Modules for portable playbooks:

Modules availability: AWS, with its matured market presence, boasts an extensive range of Ansible modules, including EC2, RDS, and more. GCP, while robust, is slightly overshadowed by the exhaustive AWS modules. Azure ensures a dedicated Ansible collection, catering to a multitude of Azure resources.

Ease of use: Each cloud provider, with its unique infrastructure, presents nuances in Ansible usage. From AWS's IAM roles to GCP's service account keys and Azure's AD credentials, the user experience varies.

Integration with Ansible Galaxy: Official Ansible collections for all three cloud providers on Ansible Galaxy simplify playbook portability.

Special features: AWS shines with integrations like Elastic Load Balancing, GCP with Google Cloud Storage and BigQuery, while Azure's strength lies in Windows VMs and Azure AD integrations.

Diving deeper into specific cloud platforms, we showcased the prowess of Ansible in configuring applications like TensorFlow on AWS, .NET Core Runtime on GCP, and managing log files on Azure. Emphasis was consistently placed on ensuring secure practices, highlighting the encryption of sensitive data and the avoidance of plaintext password storage in playbooks.

In conclusion, it is evident that Ansible's unified approach makes deploying applications on varied cloud providers a breeze. Its agentless and declarative style simplifies the entire process, making it an indispensable tool for DevOps teams. With Ansible, not only can cloud resources be adeptly managed, but seamless application deployments across diverse cloud platforms become a reality. The inherent value of Ansible in multi-cloud environments cannot be overstated, and by mastering it, DevOps teams are poised to achieve efficient, secure, and consistent deployments across the cloud spectrum.

In next chapter, we will look how Ansible can be used in area of security domain.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 7
Security Automation Using Red Hat Ansible

Introduction

Today, many organizations rely on multiple cloud providers to meet their business needs. While this provides numerous benefits, it also introduces a number of security challenges. In a multi-cloud environment, organizations must navigate different security policies, management tools, and compliance requirements. This can make it difficult to maintain a consistent security posture across all cloud providers. To address these challenges, organizations need a way to automate security processes and ensure consistency across all cloud providers. That is where Ansible comes in. Ansible is a powerful automation tool that can help organizations manage security across multi-cloud environments. With Ansible, organizations can automate security patching, manage firewalls, monitor compliance, and more, all from a single interface. In this chapter, we will explore the security challenges faced by organizations in multi-cloud environments and how Ansible can help address them. We will also cover a range of topics, including automating security patching, managing access controls,

monitoring security events, and more. By the end of this chapter, you will have a better understanding of how Ansible can help organizations maintain a consistent security posture across multiple cloud providers.

Structure

In this chapter, we will go over the following topics:

- Security challenges and how Ansible fits in
- Enterprise firewall management
- Intrusion detection and prevention system
- Security information and event management
- Policy access management
- Endpoint protection platforms
- Integrating security automation with ITSM and ticketing

Objectives

In the digital age, ensuring the security of data and systems is paramount, especially with the increasing complexity of cyber threats. This chapter elucidates the indispensable role that Ansible plays in addressing these security challenges through automation. The text navigates through the intricacies of enterprise firewall management and discusses how Ansible ensures uniformity across extensive networks. It explores how Ansible refines **intrusion detection and prevention systems (IDPS**) and bolsters **security information and event management (SIEM)**. The importance of **policy access management (PAM)** is highlighted, showing how Ansible automates its functions. By the chapter’s conclusion, the reader is expected to have a profound understanding of how Ansible is instrumental in

reinforcing an organization’s security framework within today’s evolving landscape.

Security challenges and how Ansible fits in

Following are some common challenges related to security, which are faced by most cloud providers:

- **Complexity and lack of consistency**: In a multi-cloud environment, organizations are dealing with different infrastructure, security policies, and management tools. This can make it difficult to maintain a consistent security posture across all cloud providers. Organizations may have different access controls, compliance requirements, and patch management processes, leading to a lack of consistency in their security practices.
- **Increased attack surface**: With multiple cloud providers, there are more entry points for attackers to exploit. Each cloud provider may have different vulnerabilities, making it difficult to secure all cloud providers equally. This can lead to increased risk for data breaches and cyber-attacks.
- **Lack of visibility**: With multiple cloud providers, it can be challenging to have complete visibility into the security posture of each cloud provider. Organizations may struggle to keep track of changes and events across all cloud providers, leading to blind spots in their security strategy.
- **Compliance challenges:** Each cloud provider may have different compliance requirements that organizations must adhere to. Ensuring compliance across multiple cloud providers can be difficult and time-consuming, leading to potential compliance violations.

- **Resource management:** In a multi-cloud environment, it can be difficult to manage and secure all resources effectively. Organizations must ensure that resources are correctly configured, maintained, and decommissioned. Without proper management, resources can become vulnerable to attacks or pose a risk to compliance requirements.

Ansible can help address these challenges by providing a centralized way to automate security processes and ensure consistency across all cloud providers. By leveraging Ansible, organizations can automate security patching, manage access controls, monitor for security events, and more, all from a single interface. Ansible provides a simple, flexible, and powerful way to automate security processes across multiple cloud providers. Its agentless architecture, cross-platform support, and modular design make it an ideal choice for organizations with a multi-cloud environment.

Enterprise firewall management

Ansible can automate the creation, deletion, and modification of firewall rules across multiple devices. This can be particularly useful in large, complex network environments where managing firewall rules manually can be time-consuming and error-prone. Let us now go over the various aspects of enterprise firewall management:

- **Firewall rule management:** Ansible is capable of automating the processes of creating, deleting, and altering firewall rules on a variety of devices. In expansive and intricate network settings, this automation proves especially valuable, mitigating the time-intensive and susceptible-to-error nature of manual firewall rule management.The Ansible Playbook shown below is used to manage firewall rules on a device. In this example, we are using the `firewalld`

module to allow **HTTP** and **SSH traffic** and block **SMTP traffic**. The **zone** parameter specifies the firewall zone to modify, the service specifies the name of the service to allow or block, and the port specifies the TCP or UDP port to allow or block. The **state** parameter specifies whether the rule should be **enabled** or **disabled**.

```
---
- name: Manage firewall rules
  hosts: your_target_hosts
  become: yes
  tasks:
    - name: Install firewalld
      package:
        name: firewalld
        state: present

    - name: Start firewalld
      service:
        name: firewalld
        state: started

    - name: Allow HTTP traffic
      firewalld:
        service: http
        zone: public
```

```
      state: enabled
      immediate: yes
      permanent: true

  - name: Allow SSH traffic
    firewalld:
      service: ssh
      zone: public
      state: enabled
      immediate: yes
      permanent: true

  - name: Block SMTP traffic
    firewalld:
      service: smtp
      zone: public
      state: disabled
      immediate: yes
      permanent: true
```

...

- **Policy compliance:** Ansible can be used to ensure that firewall policies are in compliance with industry standards and internal security policies. Ansible Playbooks can check the configuration of firewall rules

against established policies, alerting administrators if there are any violations.

The playbook shown below is used to check the compliance of the firewall rules. In this example, we are using the **`firewalld`** module to check the state of the **`HTTP`**, **`SSH`**, and **`SMTP rule`**s. The register parameter saves the output of each module to a variable, that can be used later. The **`mail`** module is then used to send a **`compliance report`** to the specified email address:

```
---
- name: Check firewall compliance
  hosts: your_target_hosts
  become: yes
  tasks:
    - name: Check HTTP rule
      firewalld_info:
        service: http
      register: http_rule

    - name: Check SSH rule
      firewalld_info:
        service: ssh
      register: ssh_rule

    - name: Check SMTP rule
      firewalld_info:
```

```
        service: smtp
      register: smtp_rule

    - name: Prepare compliance report
      set_fact:
        compliance_report: >
          HTTP rule state: {{
http_rule.firewalld_info.services.http.state |
default('unknown') }}\n
          SSH rule state: {{
ssh_rule.firewalld_info.services.ssh.state |
default('unknown') }}\n
          SMTP rule state: {{
smtp_rule.firewalld_info.services.smtp.state |
default('unknown') }}

    - name: Send compliance report via email
      mail:
        host: your_smtp_server
        port: your_smtp_port
        subject: Firewall Compliance Report
        body: "{{ compliance_report }}"
        from: your_email_address
        to: recipient_email_address
        secure: starttls
```

```
...
```

This playbook checks the state of the **HTTP**, **SSH**, and **SMTP rule**s and saves the output to respective variables using the **register** parameter. Then, it constructs a **compliance report** by setting a new variable **compliance_report** based on the registered outputs. Finally, it sends this report via email using the **mail** module.

- **Device configuration:** Ansible can automate the configuration of firewall devices, including settings such as **logging**, **NAT**, and **SSL VPN**. This can reduce the risk of mis-configuration and improve the consistency of firewall device configurations.

  The following playbook is used to configure a device with firewall settings. In this example, we are using the **firewalld** module to set the logging level, configure **NAT**, and enable the **SSL VPN service**. The **masquerade** parameter is used to enable **NAT** and the **service** is used to enable the **SSL VPN service**:

```
---

- name: Configure device with firewall
settings

  hosts: your_target_hosts

  become: yes

  tasks:

    - name: Install firewalld

      package:

        name: firewalld

        state: present
```

```
- name: Start firewalld
  service:
    name: firewalld
    state: started

- name: Set firewall logging level
  command:
    cmd: firewall-cmd --set-log-denied=all
    warn: false

- name: Configure NAT
  firewalld:
    zone: public
    masquerade: yes
    state: enabled
    permanent: true
    immediate: yes
- name: Enable SSL VPN service
  firewalld:
    service: openvpn
    zone: public
    state: enabled
    permanent: true
```

```
        immediate: yes

...
```

This playbook installs and starts **`firewalld`**, sets the firewall logging level, configures **`NAT`** by enabling **`masquerade`**, and enables the **`SSL VPN service`** (using **`openvpn`** as an example for the **`SSL VPN service`**).

- **Log collection and analysis:** Ansible can be used to collect logs from firewall devices and send them to a central log management system. Ansible Playbooks can also analyze firewall logs for suspicious activity and alert administrators to potential security breaches.

  The playbook shown below is used to collect and analyze firewall logs. In this example, we are using the **`fetch`** module to collect the firewall logs and save them to a specified directory. The **`handler`** section is used to trigger the **`analyze-firewall-logs.sh`** script to analyze the collected logs:

```
---
- name: Collect and analyze firewall logs
  hosts: your_target_hosts
  become: yes
  tasks:
    - name: Fetch firewall logs
      fetch:
        src: /var/log/firewalld
        dest: /path/to/your/local/directory/
```

```
        flat: yes
      register: fetch_result
    - name: Analyze logs
      command: /path/to/your/script/analyze-firewall-logs.sh "{{ fetch_result.dest }}"
      when: fetch_result.changed
      delegate_to: localhost
...
```

In this example playbook, we use the **`fetch`** module to copy the firewall logs from the remote host to a **`local directory`**. The **`path`** to the logs on the remote host **`(/var/log/firewalld`** in this case) might vary depending on the system and how the firewall is configured. You should replace **`/path/to/your/local/directory/`** with the actual path on the local machine where you want to store the fetched logs.

Then, we execute the **`analyze-firewall-logs.sh`** script if the logs were successfully fetched (indicated by **`fetch_result.changed`**). The **`delegate_to: localhost`** line is used to run the script on the local machine (the machine that runs the playbook), not on the remote host. You should replace **`/path/to/your/script/analyze-firewall-logs.sh`** with the actual path to your script.

The **`analyze-firewall-logs.sh`** file is a **`shell`** script that you need to create to analyze the firewall logs according to your requirements. Following is how the **`analyze-firewall-logs.sh`** should look like and you can customize as per your need:

```
#!/bin/bash
```

```
# The location of the log file is passed as an argument to the script

logfile=$1


# Count and print the number of firewall 'DENIED' messages in the log

echo "Number of DENIED messages:"

grep -c 'DENIED' $logfile


# Count and print the number of firewall 'ACCEPTED' messages in the log

echo "Number of ACCEPTED messages:"

grep -c 'ACCEPTED' $logfile


# Print any lines containing 'ERROR' or 'WARNING'

echo "Errors and warnings:"

grep 'ERROR\|WARNING' $logfile


# Add more commands as needed to perform your specific log analysis
```

- **Automated testing:** Ansible can automate the testing of firewall devices to ensure that they are functioning correctly. Ansible Playbooks can simulate traffic and test firewall rules to ensure that they are working as intended.

This playbook is used to test the firewall rules by simulating traffic. In this example, we are using the **firewalld** module to simulate **HTTP** and **SSH traffic** from specified IP addresses. The **register** parameter saves the output of each module to a variable that can be used later. The **assert** module is then used to check that the simulated **traffic** did not cause any changes to the **firewall rules**:

```
---

- name: Test firewall rules by simulating
traffic

  hosts: your_target_hosts

  tasks:

    - name: Simulate HTTP traffic

      command: >

        curl -I http://target_ip_or_hostname

      register: http_traffic

      ignore_errors: true


    - name: Simulate SSH traffic

      command: >

        ssh -o BatchMode=yes -o
ConnectTimeout=5 user@target_ip_or_hostname
true

      register: ssh_traffic

      ignore_errors: true
```

```
    - name: Assert HTTP traffic did not cause
changes
      assert:
        that:
          - "http_traffic is not changed"

    - name: Assert SSH traffic did not cause
changes
      assert:
        that:
          - "ssh_traffic is not changed"
...
```

In this playbook, we are using the **`command`** module to simulate **`HTTP`** and **`SSH traffic`** from the Ansible host to the target IP or hostname. The **`register`** parameter is used to save the output of each command to a variable (**`http_traffic`** and **`ssh_traffic`** respectively). The **`assert`** module is then used to check that the simulated traffic did not cause any changes to the **`firewall rules`**.

Intrusion detection and prevention system

An **intrusion detection and prevention system** (**IDPS**) is a security technology that monitors and analyzes network traffic in real time to detect and prevent malicious activity. The goal of an IDPS is to identify potential security breaches, alert security personnel, and prevent the attack before it can cause any damage.

IDPSs work by using a combination of signature-based detection, behavioral analysis, and anomaly detection.

Signature-based detection involves looking for known patterns of malicious activity such as virus signatures, while behavioral analysis looks for unusual patterns of activity that may indicate an attack. Anomaly detection involves looking for unusual network traffic or unusual behavior on the system. IDPSs typically monitor network traffic at various points in the network, including at the network perimeter, within the network, and on individual endpoints. They can be either network-based or host-based. Network-based IDPSs monitor network traffic at various points within the network infrastructure, such as routers, switches, and firewalls. Host-based IDPSs, on the other hand, run on individual endpoints, such as servers and workstations. IDPSs can generate alerts or take automated actions in response to detected threats. Alerts can be sent to security personnel for further investigation, or automated actions can be taken to block or prevent the attack. IDPSs are an important component of a multi-layered security strategy for organizations. They help to detect and prevent attacks that may bypass other security controls, such as firewalls and antivirus software. Integrating IDPSs with automation tools such as Ansible can further enhance security by enabling automated responses to detected threats, reducing response times and minimizing the impact of security incidents.

Ansible can be used to help manage and automate tasks related to IDPS. Ansible can be used to deploy and configure IDPS agents on host systems, configure network-based IDPS sensors, and manage IDPS policies and rules.

For example, Ansible Playbooks can be used to automate the installation and configuration of IDPS agents on all hosts in a network. Playbooks can also be used to configure network-based IDPS sensors to monitor network traffic for potential threats. Additionally, Ansible can be used to

manage the policies and rules for IDPSs to ensure that they are up-to-date and configured correctly.

Automating IDPS management tasks with Ansible can help to improve security and reduce the risk of security incidents. By automating these tasks, organizations can ensure that IDPSs are configured consistently and that they are continuously monitored for potential threats. This can help to minimize the risk of security breaches and ensure that security incidents are detected and addressed in a timely manner.

For our chapter here, we will use the **Snort IDPS system**. Snort is an open-source IDPS, that can be used to monitor and analyze network traffic for signs of suspicious activity, including attempts to exploit known vulnerabilities, brute-force attacks, and other types of malicious behavior.

Some common use cases for Snort include:

- **Network intrusion detection:** Snort can be used to monitor network traffic for signs of unauthorized access, such as attempts to exploit known vulnerabilities or brute-force attacks.
- **Malware detection:** Snort can be configured to detect known malware signatures and can be used to monitor for potential infections on the network.
- **Compliance monitoring:** Snort can be used to monitor network traffic for compliance with industry-specific regulations, such as **HIPAA** or **PCI-DSS**.

Installing Snort and deploying Snort rules using Ansible

In this playbook, the `vars` section defines two variables: `snort_interface` and `snort_rules_file`. These variables are used

to specify the network interface that Snort should monitor and the location of the Snort rules file, respectively.

The playbook below performs the following tasks:

- Installs Snort on the target hosts using the **`dnf`** module.
- Configures Snort by rendering a configuration file from a **`Jinja2`** template using the **`template`** module. The template contains the necessary configuration for Snort, such as the network interface to monitor, the logging settings, and the rules files to use.
- Copies a **`custom Snort rules`** file to the target hosts using the **`copy`** module. This file contains custom rules to detect specific security threats that are not covered by the default **`Snort rules`**.
- Starts the Snort service using the **`systemd`** module, ensuring that the service is enabled at boot time:

```
---
- name: Install and configure Snort
  hosts: your_target_hosts
  become: yes
  tasks:
    - name: Install Snort
      dnf:
        name: snort
        state: present

    - name: Configure Snort
```

```
    template:
      src: /path/to/your/template/snort.j2
      dest: /etc/snort/snort.conf
      owner: root
      group: root
      mode: '0644'
    notify: Restart Snort

  - name: Copy custom Snort rules
    copy:
      src: /path/to/your/rules/custom.rules
      dest: /etc/snort/rules/custom.rules
      owner: root
      group: root
      mode: '0644'
    notify: Restart Snort

handlers:
  - name: Restart Snort
    systemd:
      name: snort
      state: restarted
      enabled: yes
```

```
...
```

In this playbook, the `dnf` module installs Snort, the `template` module renders a configuration file from a `Jinja2` template and writes it to `/etc/snort/snort.conf` on the target hosts, and the `copy` module copies a `custom Snort rules` file to `/etc/snort/rules/custom.rules` on the target hosts.

If changes are made to the `configuration` file or the `rules` file, the Snort service is restarted by the `systemd` module. The `notify` keyword triggers a handler named `Restart Snort`, which is defined in the `handlers` section of the playbook. The `enabled: yes` option ensures that the service will start automatically at boot time.

Following is an example of the Snort rule, in case you are wondering how the Snort rules look like:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET 22 (msg:"SSH Brute Force Attack"; flow:to_server,established; content:"SSH-"; depth:4; content:"Login"; nocase; distance:0; content:"failed"; nocase; distance:0; classtype:attempted-login; sid:1000001; rev:1;)
```

This rule is triggered when a connection is made to `port 22` (`SSH`) on the external network (`$EXTERNAL_NET`), and the connection attempt includes the string `SSH-` (indicating an `SSH` connection attempt), followed by the word `Login` and the word failed. This is a common indicator of a brute-force attack against an `SSH` server.

When this rule is triggered, an alert is generated in the Snort log file, which can then be used by a security analyst to investigate and respond to the attack. By deploying this and other `custom Snort rules` using Ansible, you can ensure that your IDPS solution is tailored to your specific security requirements and can

help improve the effectiveness of your security monitoring and incident response capabilities.

As we proceed from our exploration of Snort, it is essential to introduce another powerful toolset in the realm of intrusion detection and prevention systems:

Security Onion: Security Onion is a free and open-source Linux distribution tailored for intrusion detection, network security monitoring, and log management. Bundled with a diverse suite of network monitoring tools and platforms, Security Onion provides an integrated environment to monitor and defend your networks. A significant advantage is its ability to integrate with various IDS engines, one of which is Suricata.

Suricata: Suricata stands as a robust, high-performance network IDS, IPS, and **network security monitoring** (**NSM**) engine. Open-source and community-driven, Suricata has been developed by the **Open Information Security Foundation** (**OISF**). It is known for its speed, scalability, and compatibility with modern networks and architectures. Its multi-threaded nature allows for very fast log processing and matching against set rules. Furthermore, Suricata's versatility lets it be deployed in a variety of environments, from large-scale enterprise networks to smaller, individual setups.

Both Security Onion and Suricata complement each other remarkably well. While Security Onion provides the framework and the user interface, Suricata offers a powerful engine that inspects the network traffic, looking for signs of malicious activity based on its rule set. Integrating these tools offers a comprehensive solution for those looking to bolster their network's security defenses.

Now, let us see how we can deploy and integrate Security Onion with Suricata in our environment using Ansible:

```
- name: Install and Setup Security Onion with
Suricata
  hosts: security_servers
  become: yes
  tasks:

    - name: Install necessary packages
      apt:
        name:
          - curl
          - software-properties-common
        state: present

    - name: Add Security Onion repository
      apt_repository:
        repo: 'ppa:securityonion/stable'
        state: present

    - name: Update repositories
      apt:
        update_cache: yes

    - name: Install Security Onion
      apt:
```

```
    name: securityonion-all
    state: present

- name: Setup Security Onion
  command:
    cmd: "sudo sosetup"
    warn: false
  when: onion_first_time_setup
  vars:
    onion_first_time_setup: true

- name: Install Suricata
  apt:
    name: suricata
    state: present

- name: Update Suricata rules
  command:
    cmd: "sudo suricata-update"
    warn: false

- name: Start Suricata in IDS mode
  ystem:
    name: suricata
```

```
        state: started
        enabled: yes

  handlers:
    - name: restart suricata
      ystem:
        name: suricata
        state: restarted
```

This playbook does the following:

- **`Install necessary packages`**: Adds utilities required for the installation process.
- **`Add Security Onion repository`**: Adds the official **`Security Onion repository`** to the apt sources list.
- **`Update repositories`**: Ensures the package list is updated.
- **`Install Security Onion`**: Installs all components of Security Onion.
- **`Setup Security Onion`**: Uses the **`sosetup`** command to set up Security Onion. This step assumes a first-time setup; hence the conditional.
- **`Install Suricata`**: Installs the Suricata IDS/IPS system.
- **`Update Suricata rules`**: Uses **`suricata-update`** to fetch the latest **`IDS/IPS`** rules.
- **`Start Suricata in IDS mode`**: Ensures the Suricata service is started and enabled at boot.

Remember, you will need to adjust parameters such as the hosts (**`security_servers`**) based on your environment, and the playbook assumes you are working in a Debian-based distribution (given the use of the apt module).

Also, note that some steps, especially the `sosetup` command, might require user intervention or further configuration based on your specific setup.

Security information and event management

Security information and event management (SIEM) is a technology that enables security teams to collect and analyze security-related data from across an organization's IT infrastructure. SIEM solutions can collect data from a variety of sources, including network devices, servers, endpoints, and security tools, and can use machine learning and other analytical techniques to identify security threats and anomalies.

Ansible can help organizations automate SIEM-related tasks, such as:

- **Log management**: Ansible can be used to automate the collection and aggregation of log data from across an organization's IT infrastructure, which can be fed into a SIEM solution for analysis.
- **Configuration management**: Ansible can be used to automate the configuration of SIEM agents and collectors on network devices, servers, and endpoints, which can simplify deployment and reduce the risk of configuration errors.
- **Security tool integration**: Ansible can be used to automate the integration of security tools with a SIEM solution, enabling security teams to collect data from a variety of sources and improve their overall security posture.
- **Incident response**: Ansible can be used to automate incident response workflows, enabling security teams

to quickly respond to security incidents and minimize the impact of security breaches.

There are multiple solutions and some of them are discussed as follows:

- **IBM Qradar**: IBM Qradar is a SIEM solution that uses machine learning and behavioral analytics to detect and prioritize security threats.
- **Splunk enterprise security**: Splunk enterprise security is a SIEM solution that provides real-time visibility into security events and helps security teams detect and respond to security threats.
- **McAfee enterprise security manager**: McAfee enterprise security manager is a SIEM solution that provides real-time threat intelligence and automates security operations to help security teams respond to security threats more quickly.
- **LogRhythm NextGen SIEM platform**: LogRhythm NextGen SIEM platform is a SIEM solution that uses AI and machine learning to identify and respond to security threats in real-time.

These are just a few examples of SIEM solutions available in the market. Organizations can choose a SIEM solution that best fits their needs and budget.

Setting up Qradar using Ansible approach

In the following code, we will set up `QRadar` using Ansible approach:

```
- name: Configure Qradar Network Settings

  hosts: qradar

  become: yes
```

```
  tasks:

    - name: Set network interface settings for
eth0

      ibm.qradar.network_interface_module:   #
Replace with the actual module name for interface
configuration

        interface: eth0

        ip_address: 10.0.0.1

        subnet_mask: 255.255.255.0

        default_gateway: 10.0.0.254

        dns_server:

          - 8.8.8.8

          - 8.8.4.4


    - name: Configure Qradar network

      ibm.qradar.network_module:   # Replace with
the actual module name for network configuration

        network_name: 'Qradar'

        ip_range: 10.0.0.0/24

        gateway: 10.0.0.254

        dns_server:

          - 8.8.8.8

          - 8.8.4.4
```

…

This playbook considers two main tasks:

- **`Configuring network interface settings:`** This task is assumed to be accomplished using a specific Ansible module for setting network interface details (**`ibm.qradar.network_interface_module`** in the example). This module is expected to take parameters such as interface name, IP address, subnet mask, default gateway, and DNS servers.
- **`Configuring QRadar networks:`** This task is assumed to be done with another Ansible module responsible for network configuration (**`ibm.qradar.network_module`** in the example). This module is expected to take parameters like network name, IP range, gateway, and DNS servers.

The actual module names, parameters, and values should be replaced as per the **`qradar_network`** module's documentation from **`IBm QRa'ar`**'s Ansible collection.

Similarly, Ansible can be used to automate many other tasks related to **`QRadar`** configuration, such as creating and configuring log sources, managing users and permissions, and configuring data retention policies.

Setting up log resources on QRadar

The playbook shown below, adds a log source called **`My App Logs`** to **`IBm QRadar`** with the following properties:

- **`Type: Universal Syslog`**
- **`Protocol: TCP`**
- **`Port: 514`**
- **`Format: RFC 3164`**
- **`Destination: 10.0.0.1`**
- **`Description: Logs from My Application`**

To use this playbook, you would need to replace the **`qradar_host, qradar_user,`** and **`qradar_password`** variables with the appropriate values for your QRadar deployment. You would also need to modify the **`log_source_name, log_source_type, log_source_protocol, log_source_port, log_source_format, log_source_destination`**, and **`log_source_description`** variables, in order to match your specific log source configuration---

```
- name: Add a log source to IBm QRadar

  host": "{{ qradar_host"}}"

  vars:

    qradar_use": "ad"in"  # replace with actual
username

    qradar_passwor": "passw"rd"  # replace with
actual password

    log_source_nam": "My App L"gs"

    log_source_typ": "Universal Sys"og"

    log_source_protoco": ""CP"

    log_source_port: 514

    log_source_forma": "RFC 3"64"

    log_source_destinatio": "10.0.".1"

    log_source_descriptio": "Logs from My
Applicat"on"

  tasks:

    - name: Add log source to QRadar

      ibm.qradar.log_source_module:  # replace
with actual module name
```

```
        use": "{{ qradar_user"}}"
        passwor": "{{ qradar_password"}}"
        nam": "{{ log_source_name"}}"
        typ": "{{ log_source_type"}}"
        protoco": "{{ log_source_protocol"}}"
        por": "{{ log_source_port"}}"
        forma": "{{ log_source_format"}}"
        destinatio": "{{
log_source_destination"}}"
        descriptio": "{{
log_source_description"}}"
```

Offense management in QRadar using Ansible

The Ansible Playbook below talks about managing offenses using QRadar. This playbook uses the QRadar API to fetch a list of offenses that have a severity level of 8 or higher and are currently open. It then closes these offenses with a closing reason ID of 1 and adds a note indicating that they were closed automatically by Ansible.

The playbook uses the **`uri`** module to make HTTP requests to the **`QRadar`** API and the register keyword to capture the response from the API. The **`loop`** keyword is used to loop over the list of offenses returned by the API, and the **`when`** keyword is used to conditionally execute certain tasks based on the severity level and status of each offense.

This playbook can be run periodically to automatically manage and close high-severity offenses in **`QRadar`**, reducing the workload for security analysts and ensuring that important security incidents are promptly addressed:

```
- name: Manage offenses in QRadar
  hosts: localhost
  vars:
    qradar_api_ur': 'https://qradar_host/'pi'  # replace with actual QRadar API URL
    qradar_use': 'ad'in'  # replace with actual username
    qradar_passwor': 'passw'rd'  # replace with actual password
  tasks:
    - name: Fetch list of offenses from QRadar
      uri:
        ur": "{{ qradar_api_url }}/siem/offen"es"
        method: GET
        use": "{{ qradar_user"}}"
        passwor": "{{ qradar_password"}}"
        force_basic_auth: yes
        return_content: yes
        status_code: 200
      register: response

    - name: Close high-severity offenses
      uri:
```

```
        ur": "{{ qradar_api_url
}}/siem/offenses/{{ item.id"}}"
        method: POST
        use": "{{ qradar_user"}}"
        passwor": "{{ qradar_password"}}"
        force_basic_auth: yes
        return_content: yes
        status_code: 200
        body_format: json
        body:
          statu': 'CLO'ED'
          closing_reason_id: 1
          note_tex': 'Closed automatically by
Ansi'le'
      loo": "{{ response.json"}}"
      when: item.severity >= 8 and item.status '=
'O'EN'
```

Now, you might be wondering what QRadar offense looks like. Example of an offense is as follows:

```
{
  "id": 12345,
  "description": "Multiple failed login attempts",
  "severity": 8,
  "status": "OPEN",
```

```
  "category": "Authentication",
  "offense_type": "Anomaly",
  "relevance": 8,
  "source_address": "192.168.1.10",
  "source_network": "Internal",
  "destination_address": "192.168.1.20",
  "destination_network": "DMZ",
  "protocol": "TCP",
  "start_time": "2023-06-20T14:45:30Z",
  "last_updated_time": "2023-06-20T14:50:30Z",
  "event_count": 15,
  "flow_count": 0,
  "assigned_to": "admin",
  "follow_up": false,
  "offense_source": "Log Event"
}
```


Here is a brief explanation of some of the key properties:

- **`id`**: The unique identifier of the offense.
- **`description`**: A human-readable description of the offense.
- **`severity`**: The severity level of the offense on a scale from 1 to 10.
- **`status`**: The current status of the offense (For example, **`OPEN`**, **`CLOSED`**).

- **`category`**: The category of the offense (For example, **`Authentication`**).
- **`offense_type`**: The type of offense (For example, **`Anomaly`**).
- **`source_address`**: The IP address where the offense originated.
- **`destination_address`**: The IP address where the offense was targeted.
- **`start_time`**: The date and time when the offense started.
- **`last_updated_time`**: The date and time when the offense was last updated.
- **`event_count`**: The number of events associated with the offense.
- **`assigned_to`**: The username of the user to whom the offense is currently assigned.

Note that this is just a basic example, and offense rules can be much more complex depending on the specific needs of your organization.

Let us also look at Splunk in addition to QRadar.

Introduction to Splunk integration

Splunk provides capabilities to analyze and visualize machine data from various sources. It is often utilized as a SIEM solution, aggregating logs, events, and other data from various parts of an infrastructure to provide a centralized view of security events. Integrating Ansible with Splunk can enable you to send Ansible Playbooks' run summaries, facts gathered from systems, or even custom messages into Splunk for analysis and visualization.

Install Splunk using Ansible

Here is a basic Ansible Playbook that can be used to install and set up a Splunk instance on a target machine. For this example, we will be setting up **Splunk Enterprise** on an **Ubuntu** system. Remember to tweak as necessary for your environment:

```
---
- name: Install Splunk Enterprise on Ubuntu
  hosts: splunk_servers
  become: yes
  vars:
    splunk_url: 'https://download.splunk.com/products/splunk/releases/8.1.3/linux/splunk-8.1.3-63079c59e632-linux-2.6-amd64.deb'
    splunk_start_command: '/opt/splunk/bin/splunk start --accept-license'
    splunk_user: 'admin'
    splunk_password: 'YourStrongPassword'  # Change this!

  tasks:
    - name: Download Splunk .deb file
      get_url:
        url: "{{ splunk_url }}"
        dest: "/tmp/splunk.deb"
        mode: '0755'
```

```
    - name: Install Splunk
      apt:
        deb: "/tmp/splunk.deb"

    - name: Start Splunk and accept license
      command: "{{ splunk_start_command }}"
      become: yes
      become_method: sudo
      become_user: root
      environment:
        SPLUNK_START_ARGS: "--accept-license"
        SPLUNK_PASSWORD: "{{ splunk_password }}"

      ignore_errors: yes  # Splunk start command might throw a false error even if it starts correctly

    - name: Enable Splunk to start on boot
      command: "/opt/splunk/bin/splunk enable boot-start"
```

Note:

- **This playbook sets up Splunk Enterprise on an Ubuntu machine. You might need to adjust the download link (splunk_url variable) depending on the Splunk version and the OS you are targeting.**
- **It is important to modify the splunk_password variable to a strong password of your choosing. This is the password for the default admin account.**

- **Splunk's startup command sometimes gives an error due to the fact that Splunk takes a while to start, even though it starts up correctly. That is why we use ignore_errors: yes. However, make sure to check the target machine to ensure Splunk started as expected.**

After running this playbook, Splunk should be installed and running on the target machine. You can then access the Splunk web interface using the IP address of the machine on port `8000`, like **http://ip_address:8000**, and log in using the credentials provided (admin and the password you set).

Remember, always refer to Splunk's official documentation when setting up in a production environment, as there are many configuration options and best practices to be aware of.

Integrating systems with Splunk

This playbook demonstrates sending a simple message to Splunk after gathering facts from a target system:

```
---
- name: Ansible Splunk Integration
  hosts: target_systems
  tasks:

    - name: Gather Facts
      setup:

    - name: Send facts to Splunk
      uri:
```

```
        url:
"https://YOUR_SPLUNK_URL:8088/services/collector/e
vent"

        method: POST

        body: '{"event": "{{ ansible_facts }}"}'

        headers:

          Authorization: "Splunk YOUR_HEC_TOKEN"

        validate_certs: no  # adjust based on your
environment

      delegate_to: localhost
```

Replace **`YOUR_SPLUNK_URL`** with the URL of your Splunk instance and **`YOUR_HEC_TOKEN`** with the HEC token you created in Splunk.

This playbook performs the following actions:

- **`Gather Facts`**: Collects facts from the target system.
- **`Send facts to Splunk`**: Sends these gathered facts to Splunk using the HTTP event collector.

Remember, in a real-world scenario, you would want to limit the amount of data you send to Splunk, or perhaps process and filter it to just send specific details.

Further integrations:

Once you have basic integration set up, you can:

- Create alerts in Splunk based on certain events or outputs from Ansible Playbooks.
- Send custom logs or messages from Ansible Playbooks to Splunk for specific actions or errors.
- Integrate Splunk searches and dashboards with Ansible to trigger playbooks based on specific events detected in Splunk.

When integrating Ansible with SIEM solutions like Splunk, you enhance the ability to detect, analyze, and respond to security incidents, making your infrastructure more resilient and proactive.

Policy access management

The need for PAM has increased in recent years due to the growing threat of cyber-attacks and data breaches. Attackers often target privileged accounts because they provide access to sensitive data and critical systems. PAM solutions help organizations protect privileged accounts and improve their overall security posture.

There are several benefits of using a PAM system, including:

- **Enhanced security:** PAM solutions provide an additional layer of security for privileged accounts, reducing the risk of unauthorized access, data breaches, and cyber-attacks.
- **Compliance:** Many regulatory standards require privileged access management to protect sensitive data and critical systems. PAM solutions help organizations meet these compliance requirements.
- **Accountability:** PAM solutions provide detailed audit logs, so organizations can track and monitor privileged access, enabling accountability and regulatory compliance.
- **Improved productivity:** PAM solutions can automate routine tasks, reducing the workload on IT teams and enabling them to focus on more strategic initiatives.
- **Reduced risk:** PAM solutions can help organizations reduce the risk of privileged account misuse, improving overall security posture and reducing the risk of data breaches.

Here, we will use **CyberArk** as an example of a PAM solution. Follow the given steps:

1. Install the **cyberark.conjur** collection by running the following command:

   ansible-galaxy collection install cyberark.conjur

2. Confirm that the **collection** is installed by running the following command:

   ansible-galaxy collection list | grep cyberark.conjur

3. The output should show the version number of the **CyberArk Conjur** collection.

   cyberark.conjur   1.0.0

For example, the **cyberark.conjur.conjur_variable** module can be used to retrieve secrets from **CyberArk Conjur** and store them as variables in your playbook. An example playbook is illustrated below*:*

```
---

- name: Retrieve secret from CyberArk Conjur

  hosts: localhost

  vars:

    api_key: 'YourConjurAPIKey'  # replace with
your actual Conjur API key


    account: 'YourConjurAccount'  # replace with
your actual Conjur account name

  tasks:

    - name: Retrieve secret

      cyberark.conjur.conjur_variable:
```

```
        variable_id: 'mysecret'
        api_key: "{{ api_key }}"
        account: "{{ account }}"
      register: secret

    - name: Print secret
      debug:
        msg: "The secret is: {{
secret.result.value }}"
```

In this playbook, the **`variable_id`** parameter is used to specify the ID of the **`Conjur`** variable from which to retrieve the secret.

In this playbook:

- The **`vars`** section defines the **`Conjur`** API key and account name.
- The **`cyberark.conjur.conjur_variable`** module retrieves the value of the **`mysecret`** variable from **`Conjur`** using the **`api_key`** and account defined in the **`vars`** section. This value is stored in the **`secret`** variable.
- The debug task prints the retrieved secret to the console.

In addition to **`CyberArk`** there are other PAM solutions too. Let us look at BeyondTrust's PAM solution which provides a centralized repository to manage and store sensitive credentials. These can be **`SSH`** keys, passwords, or certificates, among others. BeyondTrust offers a comprehensive solution for **privileged access management** (**PAM**) designed to protect privileged accounts from potential breaches. By integrating Ansible

with BeyondTrust, IT and security teams can leverage the enhanced security features of BeyondTrust while automating tasks across their environments. For instance, using BeyondTrust's API, an Ansible playbook can fetch necessary SSH credentials securely, ensuring that operations like remote server logins are both automated and secure. This integration ensures that sensitive credentials are never hard-coded or stored insecurely within automation scripts, but instead, are retrieved dynamically when needed, maintaining the principle of least privilege and ensuring tight access controls.

BeyondTrust can be integrated to fetch credentials for various tasks. Suppose Ansible needs to configure a service on a remote machine. Instead of storing the SSH key or password for that machine in the playbook or in an easily accessible file, Ansible can be configured to fetch this key/password from BeyondTrust at runtime. Once Ansible completes the task and the session ends, BeyondTrust can rotate that password or key. This ensures:

- No hardcoding of secrets.
- A full audit trail of when and why a particular secret was accessed.
- Compliance with best practices for secret management.

Creating a sample Ansible Playbook that integrates with BeyondTrust requires the use of the BeyondTrust API (or potentially other tools offered by BeyondTrust for integration). Here is a simplified example, assuming BeyondTrust offers a RESTful API for fetching credentials. This playbook fetches SSH credentials from BeyondTrust, uses them to log in to a remote server, and then performs a simple task:

```
---
```

```
- name: Integrate with BeyondTrust for Privileged
Access Management

  hosts: your_target_hosts

  gather_facts: no

  tasks:

    - name: Fetch SSH credentials from BeyondTrust

      uri:

        url:
"https://beyondtrust_api_endpoint/credentials/your
_credential_id"

        method: GET

        headers:

          Authorization: "Bearer {{
beyondtrust_api_token }}"

        return_content: yes

      register: fetched_credentials

      no_log: true


    - name: Store the fetched SSH username

      set_fact:

        remote_ssh_user: "{{
fetched_credentials.json.username }}"


    - name: Store the fetched SSH password

      set_fact:
```

```
        remote_ssh_pass: "{{
fetched_credentials.json.password }}"
      no_log: true


    - name: Sample task using the fetched
credentials
      command: echo "Hello from BeyondTrust
integrated host!"
      delegate_to: "{{ inventory_hostname }}"
      remote_user: "{{ remote_ssh_user }}"
      vars:
        ansible_ssh_pass: "{{ remote_ssh_pass }}"
```

Note:

- **The above is a simplified and hypothetical playbook to demonstrate the concept. It assumes BeyondTrust offers a REST API that can provide credentials in a particular JSON format.**
- **The no_log: true directive ensures that the fetched credentials are not printed in the Ansible output.**
- **Proper handling of the API token (beyondtrust_api_token in this case) is crucial. Consider storing it securely using Ansible vault or another secrets manager.**
- **Before executing any playbook that integrates with third-party services, especially those dealing with sensitive data, thoroughly read and understand their official documentation, ensure best practices, and test in a controlled environment.**

Endpoint protection platforms

endpoint protection platforms (**EPP**) are a category of security software designed to protect endpoint devices such as laptops, desktops, and mobile devices from various cyber threats. These platforms provide a comprehensive set of

security solutions that are essential for protecting endpoints from viruses, malware, ransomware, and other types of cyber-attacks.

Endpoint protection platforms usually include a suite of security technologies that work together to secure endpoints. These technologies may include antivirus software, firewall protection, intrusion detection and prevention, web filtering, application control, and data loss prevention.

Endpoint protection platforms are essential for securing enterprise networks, as they provide a centralized management platform for securing endpoint devices. These platforms allow security administrators to manage security policies, deploy security updates, and monitor endpoint devices for security threats.

Endpoint protection platforms have become increasingly important as more organizations have adopted a remote workforce model. With employees working from home and accessing corporate resources through personal devices, the need for comprehensive endpoint security solutions has increased dramatically.

By using endpoint protection platforms, organizations can ensure that their endpoint devices are protected against a wide range of cyber threats, which can help to prevent data breaches, intellectual property theft, and other types of cybercrime.

EPP can be set up and managed using Ansible. Some examples of EPPs that can be managed with Ansible are as follows:

- **Symantec Endpoint Protection**: Ansible can be used to install, configure and manage Symantec Endpoint Protection across multiple systems. Ansible can be used to automate the installation of Symantec

Endpoint Protection on servers and workstations, manage policies and configurations, and monitor the status of the software.

- **McAfee Endpoint Protection**: Ansible can be used to automate the installation and configuration of McAfee Endpoint Protection across multiple systems. This includes deploying agents, managing policies and configurations, and monitoring the status of the software.
- **Microsoft Defender ATP:** Ansible can be used to automate the installation and configuration of Microsoft Defender ATP on Windows endpoints. This includes deploying agents, configuring policies, and monitoring the status of the software.
- **CrowdStrike Falcon:** Ansible can be used to automate the deployment and configuration of the CrowdStrike Falcon agent on endpoints. This includes managing policies and configurations, monitoring the status of the software, and responding to security incidents.

Setting up Symantec Endpoint Protection

The following section shows the playbook for setting up Symantec Endpoint Protection using Ansible:

```
---

- name: Install Symantec Endpoint Protection

  hosts: your_target_hosts  # replace with your actual target hosts

  become: yes

  tasks:
```

```
    - name: Add Symantec repository
      yum_repository:
        name: Symantec
        description: Symantec Repository
        baseurl: http://your-repo-url/  # replace with your actual repo URL
        enabled: yes
        gpgcheck: yes

        gpgkey: file:///etc/pki/rpm-gpg/Symantec-GPG-KEY  # replace with the location of your GPG key
    - name: Install SEP
      yum:
        name: symantec-endpoint-protection
        state: latest
```

In this playbook:

- The **yum_repository** module is used to add the Symantec repository to the system's yum configuration. You need to replace **http://your-repo-url/** with the actual URL of your Symantec repository, and **/etc/pki/rpm-gpg/Symantec-GPG-KEY** with the location of your GPG key (if you have one and if it is required by the repository).
- The **yum** module is used to install Symantec Endpoint Protection (**symantec-endpoint-protection**) from the repository. The **state: latest** option ensures that the latest version of the software is installed.

Setting up Microsoft Defender ATP Endpoint Protection

This preceding playbook uses the **`Chocolatey`** package manager to install **`Microsoft Defender`** on a Windows host. It then uses the **`win_security_policy`** module to enable **`Windows Defender Antivirus`** by setting the appropriate security policy. You can customize the playbook as needed for your specific use case and environment.

Refer to an Ansible Playbook below:

```
---

- name: Install and Configure Microsoft Defender

  hosts: windows_hosts  # replace with your actual target hosts

  gather_facts: yes

  tasks:

    - name: Install Microsoft Defender using Chocolatey

      win_chocolatey:

        name: defender

        state: present

    - name: Enable Windows Defender Antivirus

      win_security_policy:

        section: "Windows Defender Antivirus"

        key: "EnableAntivirus"

        value: "Enabled"
```

In this playbook:

- The `win_chocolatey` module is used to install `Microsoft Defender` using `Chocolatey`, a package manager for Windows.
- The `win_security_policy` module is used to enable `Windows Defender Antivirus` by setting the `EnableAntivirus` key in the `Windows Defender Antivirus` section of the Windows security policy.

Integrating security automation with ITSM and ticketing

In the modern enterprise landscape, security automation does not function in isolation. It often intertwines with **it service management (ITSM)** and ticketing systems, creating an orchestrated ecosystem where security incidents, alerts, or changes result in actionable ITSM tickets and vice-versa.

Why integrate with ITSM and ticketing systems?

- **Visibility and accountability**: Every security alert or incident, if tracked as a ticket, offers greater visibility across teams. It ensures every alert is attended to and there is an audit trail for every action taken.
- **Orchestrated responses**: By integrating with ticketing systems, automated workflows can be triggered based on the type, severity, or source of a security incident, leading to faster and more accurate responses.
- **Operational efficiency**: Manual handovers between security and operations teams can introduce errors and delays. Direct integration reduces manual intervention and streamlines operations.

ServiceNow integration:

ServiceNow, a leading ITSM solution, can be tightly integrated with Ansible for security automation. Here is an example:

Scenario: Let us assume a situation where a vulnerability scanner in our environment identifies a potential threat on a set of servers. Instead of manually creating a ticket, the scanner triggers an Ansible Playbook through an API call, which in turn, creates a ticket in **`ServiceNow`** detailing the vulnerability.

Sample Ansible Playbook for **`ServiceNow`** integration:

```
---

- name: Create ServiceNow Ticket for Security Alert

  hosts: localhost

  tasks:

    - name: Create ServiceNow Incident

      servicenow.itsm.incident:

        instance: your_instance_name

        username: your_username

        password: your_password

        state: new

        short_description: "Security vulnerability detected."

        description: "Vulnerability detected on server {{ inventory_hostname }}. Immediate action required."

        urgency: '1'
```

```
      severity: '1'
    register: result

- name: Print ServiceNow Incident
  debug:
    var: result
```

In this playbook, upon detecting a vulnerability, a new **`ServiceNow Incident`** is created with a specified urgency and severity. By automating this process, teams can ensure every detected vulnerability results in a tracked action item, promoting faster resolution times and more effective collaboration between security and operations teams.

Integrating with ITSM systems like **`ServiceNow`** ensures that there is a seamless flow of information and action between automated security operations and IT management processes, resulting in a more robust and responsive IT environment.

Conclusion

In conclusion, our journey into Ansible's capabilities has shown it to be an invaluable tool for security and network automation. Whether extracting critical network data with modules such as **`cnos_facts`**, **`edgeos_facts`**, **`os_command`**, **`os_config`**, and **`os_l3_interface`**, or managing tasks like automating backups and streamlining configurations, Ansible stands out in its efficiency and flexibility.

Diving deeper into the security landscape, we have seen how firewalld simplifies firewall management, and how Ansible empowers automation in log collection, analysis, and maintenance. Its ability to integrate with proprietary systems, as demonstrated through vendor-specific modules

like `qradar_network` for IBM QRadar, exemplifies Ansible's far-reaching versatility.

Beyond traditional security and network automation, our exploration underscored Ansible's role in interweaving security automation with ITSM through systems like ServiceNow. This integration ensures a harmonious and reactive collaboration between automated security operations and IT management processes, bolstering both speed and reliability. Furthermore, our dive into cybersecurity reveals Ansible's prowess in automating tasks for systems such as CyberArk Conjur and Symantec Endpoint Protection. This further cements its indispensable role in fortifying and uplifting security stances across diverse infrastructures.

As we pivot to the next chapter, our lens focuses on the avant-garde arena of edge computing. We will delve into how Ansible effectively manages and orchestrates tasks at the network's edge, addressing the distinct demands of edge computing environments. It is clear that Ansible's reach extends beyond mere network and infrastructure automation, making inroads into edge computing, and reaffirming its position as a cornerstone in our constantly evolving technological landscape.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 8

Red Hat Ansible Automation for Edge Computing

Introduction

In the evolving landscape of computing, edge computing has emerged as a revolutionary architecture, propelling computation closer to the source of data generation—the edge of the network, away from the conventional centralized data centers. This paradigm is crucial in optimizing latency, bandwidth, and data privacy, especially in applications like autonomous vehicles, smart cities, and IoT devices, ensuring rapid and secure data processing and analytics.

However, edge computing introduces unique challenges, notably due to the inherent decentralization and geographical distribution of computing resources. One specific challenge is the lack of on-site expertise due to the widespread deployment of edge devices, necessitating advanced remote management and automation solutions. Another is the necessity to maintain uniformity and coordination across geographically disparate environments, which demands robust and adaptive management solutions.

Red Hat Ansible Automation Platform emerges as a transformative solution in such demanding scenarios, offering powerful automation and orchestration tools designed to navigate the complexities of managing distributed edge computing resources. It enables streamlined operations, heightened security, and enhanced scalability across varied edge computing landscapes.

By utilizing the Red Hat Ansible Automation Platform, organizations can address the intricate challenges posed by edge computing, ensuring seamless and efficient management of diverse computing resources, reducing the risks associated with manual interventions, and fostering a more responsive and adaptive computing environment. This chapter aims to delve deeper into how Ansible can be leveraged to overcome the challenges and optimize the benefits of edge computing.

Structure

In this chapter, we will learn the following topics:

- Enterprise automation
  - Ansible Automation Platform
  - Automation mesh
- Industry use case
  - Transportation
  - Retail
  - Industry 4.0
  - Telecommunications
  - Financial services and insurances
  - Smarter cities
  - Health care

Objectives

The objectives stemming from the transformative impact of Red Hat Ansible in edge computing encompass enhancing retail efficiency and customer experiences. Ensuring finance sector data privacy and compliance, managing complexity in diverse industries, fostering innovation through open-source collaboration, preparing for the IoT era, and enabling businesses to fully exploit edge computing's potential.

Enterprise automation

Enterprise can have devices or computing infrastructure hosts at various locations. We will see how the **Red Hat Ansible Automation Platform** (**AAP**) is used to manage automation for all these devices. In *Figure 8.1*, we see computing devices at different locations. Some are infrastructure edge devices meaning that are closer to data centers and then we have edge server, which means server or computing resource that performs data computation at the end of the network. Then we also see far edge devices like cameras, IoT devices and so on. which are devices that are deployed farthest from the data center and closest to the end user. *Figure 8.1* also shows how an automation controller is used by users of AAP to create, operate and manage automations across the enterprise. Basically, here automation controller is used to manage the automations defined in the automation mesh, which we will discuss further.

Ansible Automation Platform

We have a complete chapter on the Ansible Automation Platform, so in this chapter, we will cover AAP only briefly. AAP is a strategic end-to-end automation solution. It connects cross-functional teams, like developers, to operations or security, so they can automate together. AAP

provides an enterprise framework for building and operating IT automations at scale from hybrid cloud to edge.

Features provided by AAP:

- With AAP we get distributed computing elements.
- Inventory IoT device endpoints are connected at the far edge.
- Network edge devices are nothing but devices or LANs that connect to the internet.
- Application lifecycle on Windows and Linux endpoints.
- Actions in response to user input at the infrastructure edge.

Automation mesh

Automation mesh, as a pivotal component of AAP, acts as the backbone to sustain automation at scale, catering especially to cloud-native environments. It grants unparalleled flexibility and acumen in orchestrating distributed, remote, and elaborate automation deployments, coupled with superior visibility, control, and reporting.

To illustrate, consider a scenario where network latency poses significant challenges due to the geographical dispersion of automation nodes. Automation mesh adeptly mitigates such issues by leveraging latency tolerance mechanisms, ensuring that the delay in data transmission does not compromise the execution and performance of automation tasks. This leads to uninterrupted and reliable automation workflows, even in environments with substantial latency.

Another exemplification of its resilience is its dynamic rerouting capability. In circumstances where a primary route is unavailable or compromised, automation mesh swiftly recalculates and establishes an alternative pathway for task

execution, avoiding any disruption in service. This feature is particularly crucial in maintaining the continuity and reliability of automation processes across dispersed and diverse environments.

Therefore, automation mesh is more than just an overlay network; it is a sophisticated framework designed to facilitate the distribution of work across a vast and varied array of workers. It achieves this through nodes that create peer-to-peer connections with each other, utilizing existing networks, thereby ensuring that the expansive and diverse nature of today's automation needs are met with resilience, reliability, and efficiency.

Features provided by automation mesh

The following are the features provided by automation mesh:

- Dynamic cluster capacity that scales independently, allowing you to create, register, group, ungroup and deregister nodes with minimal downtime.
- Control and execution plane separation enable you to scale playbook execution capacity independently from control plane capacity.
- Deployment choices that are resilient to latency, reconfigurable without outage, and dynamically re-reroute to choose a different path when outages may exist, mesh routing changes.
- Connectivity that includes bi-directional, multi-hopped mesh communication possibilities which are **Federal Information Processing Standards** (**FIPS**) compliant.

In the following figure, AAP components managing edge automation are showcased. Please refer to the following figure:

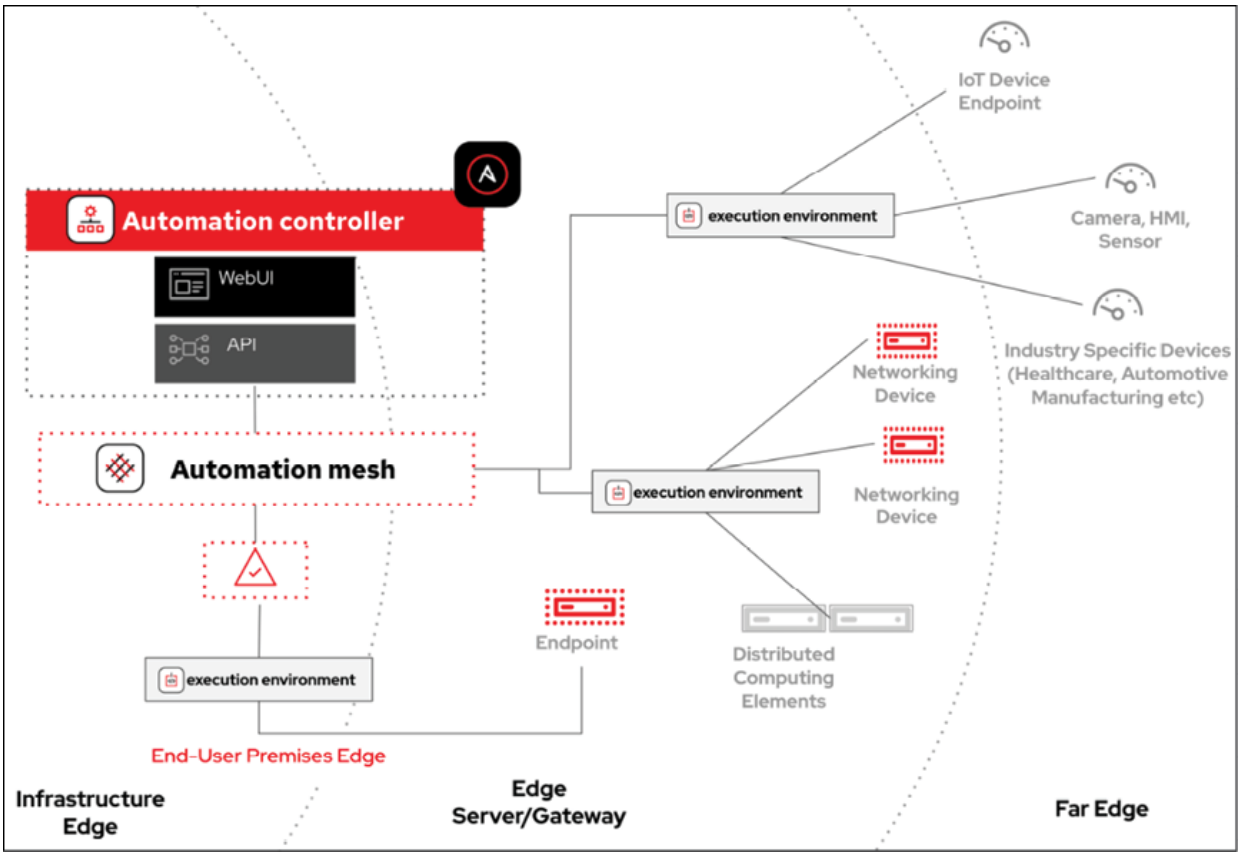

Figure 8.1: *AAP components managing edge automation*

Industry use case

Within the broad scope of Red Hat Ansible and edge computing, a particularly pertinent exploration is its application across different industries. This section, industry use cases, underscores this versatile applicability by delving into specific scenarios from various sectors. Whether it is transportation, retail, healthcare, finance, or banking, each has harnessed the capabilities of Red Hat Ansible within its edge computing infrastructure to address unique challenges and objectives. By investigating these instances, we aim to illustrate not just the broad applicability of Ansible within edge computing but also its powerful potential in driving sector-specific transformations. Let us proceed to unveil how these technologies have been woven into the fabric of

diverse industries, shaping the future of their operations and strategies.

Transportation

Commencing our exploration of industry-specific applications, let us embark on the journey of the transportation sector, an industry that thrives on speed, efficiency, and accuracy. With the advent of edge computing and automation tools like Red Hat Ansible, this industry has undergone a significant digital transformation, optimizing its sprawling network of operations. Whether it is managing fleet operations, monitoring traffic flow, improving route planning, or ensuring safety measures, Ansible in tandem with edge computing has proven to be an invaluable ally. This powerful synergy allows real-time data processing and automated decision-making, which is essential for timely actions and responses within the fast-paced transportation environment. In the following sections, we will delve deeper into how the transportation industry leverages Red Hat Ansible and edge computing, addressing challenges, improving efficiencies, and paving the way for a more streamlined and sustainable future of mobility.

Refer to *Figure 8.2*, illustrating the various elements of the transportation sector, where edge computing and Red Hat Ansible are poised to make significant strides:

Figure 8.2: *Transport industry*

Across the transportation industry, customer demand is inspiring new, innovative services, but it is also creating challenges for nearly every form of transportation. An airline, for example, can have planes taking off every 60 seconds with passengers, cargo, and security to manage and monitor every step of the journey. Railway companies face increasing customer demands for connectivity while also having to manage device configuration, security of their data and networks, and pressure to provide new innovative services for passengers.

Modernizing transportation to be smart, safe, and highly efficient is no small feat. For example, if a railway company wanted to include intelligent features such as dynamic LED information displays, without central control, they would likely have to go onto every train to manually check, update, or fix each device. The time and resources to set up and maintain such a feature would make it unfeasible.

Automation of these edge devices, their configuration, and their software life cycles makes centralized control possible

and helps the railway company gain a single view of all devices so that monitoring and updates happen automatically. Centralized control through automation also makes other intelligent features possible, such as digital seat booking systems, **closed-circuit television** (**CCTV**), safety monitoring, or onboard Wi-Fi access.

By automating complex, manual device configuration processes, transportation companies can deploy software and application updates to trains, airplanes, or other moving vehicles without needing specialized, proprietary software from the same hardware vendor. This approach helps save time that teams can put toward more valuable service innovation.

Compared to a manual approach, automating device installation and management is safer and more reliable. Automating device management eliminates the need for individual technicians to physically plug in USB drives to multiple endpoints. Updates are managed by vehicle type, avoiding any fleet-wide service effect and can even be done while a vehicle, such as a train, is in motion.

Advantages that automating at edge brings to the transport industry:

- Decreases device configuration times.
- Enhances security for critical transportation infrastructure.
- Establishes comprehensive device access for user-friendly service updates and innovation.

Let us understand this more with the following scenario:

Scenario: In a bustling metropolis, traffic management systems are pivotal to maintaining the flow of traffic and ensuring the safety of commuters. These systems include traffic lights, speed cameras, and other critical infrastructure that require constant updates and configuration.

Challenge: The primary challenges faced in this scenario include the manual, time-intensive configuration of each device and the implementation of security protocols for every piece of critical transportation infrastructure. Additionally, providing seamless and user-friendly service updates is crucial to innovate and adapt to the ever-evolving urban landscape.

Ansible's solution: Ansible's automation capabilities can significantly streamline the configuration processes of these devices. By automating configuration tasks, Ansible reduces the timespan traditionally required to configure each device manually, allowing for rapid deployment and adjustments.

For instance, if there is a need to modify the traffic light sequence at a busy intersection to alleviate congestion during peak hours, Ansible can facilitate instant modifications without the need for on-site interventions, thereby reducing downtime and improving traffic flow.

Security enhancement: Moreover, Ansible strengthens the security posture of the critical transportation infrastructure by automating the deployment of security patches and updates. It ensures that all devices are fortified against the latest security threats, reducing vulnerabilities and safeguarding the integrity of transportation systems.

Innovation and user-friendly service updates: With comprehensive device access, Ansible enables the introduction of innovative solutions and user-friendly service updates. Transport authorities can effortlessly roll out enhancements, optimize traffic flow algorithms, or implement new features, ensuring the continual evolution and improvement of transportation systems to meet the demands of modern cities.

Conclusion: By employing Ansible for automating at the edge in the transport industry, cities can leverage decreased device configuration times, enhanced security for crucial

infrastructures, and establish comprehensive device access for seamless service updates and innovations. This not only bolsters the reliability and efficiency of transportation systems but also paves the way for future innovations in traffic management and control.

Retail

Compared to a manual approach, automating device installation and management is safer and more reliable. Automating device management eliminates the need for individual technicians to physically plug in USB drives to multiple endpoints. Updates are managed by vehicle type—avoiding any fleet-wide service effect—and can even be done while a vehicle, such as a train, is in motion.

On Day 1 and beyond, consistency and reliability become the most important features of a connected environment in the retail space. Any disruption to the network affects front-line service personnel. Whether it is a temperature sensor that affects the freezer in an ice cream shop or a cash register that is running slowly, every minor glitch in the performance of the various **Internet of Things** (**IoT**) devices used at the edge can disrupt the interaction between the customer and public-facing employees. Automation and standardization provide the consistency of configuration and software life cycle management of these edge devices, helping to enhance the customer experience, which in turn promotes a healthy bottom line.

Examine *Figure 8.3*, representing the multifaceted retail industry, a prime candidate for transformation through the application of edge computing and Red Hat Ansible:

Figure 8.3: *Retail industry use case*

Advantages that automating at edge brings to the retail industry:

- Enhance the customer experience and the company's bottom line.
- Stand up, configure, and audit new devices with the speed and scalability needed.
- Maintain consistent and reliable functionality of edge devices.

Let us understand this more with scenario, as follows:

Scenario: In the competitive landscape of retail, maintaining optimal inventory levels and delivering personalized in-store experiences are critical for customer retention and revenue growth. Retailers deploy numerous interconnected devices and systems like **Point of Sale** (**POS**) systems, inventory management systems, and customer engagement platforms to achieve these goals.

Challenge: Retailers grapple with the manual and cumbersome process of updating and configuring each in-store device and system. Ensuring real-time synchronization between inventory levels and sales data and deploying

timely and relevant customer engagement strategies are persisting challenges. The lack of immediate system updates and configurations can result in stock discrepancies, lost sales, and diminished customer satisfaction.

Ansible’s solution: Ansible can automate the configuration and updating of in-store devices and systems, reducing manual effort and ensuring real-time accuracy and synchronization. For instance, if a product is sold, Ansible can facilitate instant updates across POS and inventory management systems, reflecting accurate stock levels and preventing over-selling.

Personalized customer experience: Retailers can leverage Ansible to automate and enhance customer engagement strategies by personalizing in-store experiences. For example, Ansible can automate the deployment of personalized promotions and offers to individual customers based on their purchase history and preferences, directly through the retailer’s app or other engagement platforms, thus enhancing customer satisfaction and loyalty.

Inventory optimization: By automating inventory management processes, retailers can maintain optimal stock levels, preventing stockouts or overstocks and ensuring that the right products are available at the right time. This leads to improved operational efficiency, reduced holding costs, and increased sales and profitability.

Security and compliance: In the retail environment, securing sensitive customer and transaction data is paramount. Ansible ensures that all in-store systems are consistently compliant with security policies and regulations by automating the deployment of security patches and updates, protecting against vulnerabilities and data breaches.

Conclusion: By integrating Ansible into the retail industry, retailers can realize the benefits of automated and synchronized inventory management, enhanced and personalized customer experiences, and robust security and compliance measures. This leads to increased operational efficiency, customer satisfaction, and revenue growth, allowing retailers to stay competitive and responsive to market demands.

Industry 4.0

Industry 4.0 is revolutionizing the way companies manufacture, improve, and distribute their products. From oil and gas refineries to smart factories to supply chains, organizations are integrating technologies such as IoT, cloud computing and analytics, and AI/ML into their production facilities and across their operations. The goal is higher production volumes, lower costs, and better quality control. But bringing these technologies to sophisticated manufacturing operations will take more than the flip of a switch.

Observe the *Figure 8.4*, illustrating the concept of **Industry 4.0**, a revolutionary movement significantly empowered by technologies such as edge computing and Red Hat Ansible.

The key areas where solutions need to be identified are:

- How to manage and process massive amounts of data to assess production quality effectively.
- How to create and deploy specialized AI models to hundreds or thousands of machines and devices on manufacturing floors.
- How to scale and maintain deployments across multiple facilities and refinery sites around the world.
- How to do all this cost-effectively.

Please refer to the following figure:

Figure 8.4: *Industry 4.0 use case*

Key factors for automation at the edge in the oil and gas industry include the consistency it provides and the potential for lowering costs. With facilities and machinery spread across wide geographic areas, automation provides the opportunity for greater efficiency by reducing or eliminating the need to send expertise on-site. For example, automating the launch of new software to edge devices spread across a series of refineries not only saves the time of sending a technician to the site, but it ensures the correct version is applied and maintained after deployment. On the manufacturing floor, automation supported by visualization algorithms can help detect defects in manufactured components on the assembly line and ensure safe factory operations by identifying and alerting hazardous conditions or unpermitted actions.

Following are the advantages that automating at edge brings to industry 4.0:

- Improves quality control during manufacturing processes.
- Prevents disruptions, supporting *24x7* production

- Minimizes human intervention needed for repeatable tasks, allowing skilled employees to focus on higher-value tasks.
- Reduces downtime with more accurate, scheduled maintenance.
- Improves worker safety.

The following detailed scenario illustrates how Ansible's automation capabilities can address the inherent challenges in Industry 4.0, optimizing and enhancing the modern manufacturing environment:

Scenario: Industry 4.0 requires the seamless integration of diverse systems such as robotics, sensors, and intelligent systems across the manufacturing process. This interconnected environment is critical for achieving real-time monitoring, analysis, and optimization of manufacturing operations.

Challenge: The integration and management of multiple systems and technologies pose significant challenges, including system compatibility, real-time data processing, and analysis, and the implementation of adaptive and self-optimization processes. Efficiently configuring and updating each system, device, and ensuring real-time data exchange are crucial for optimizing manufacturing processes and output.

Ansible's solution: Ansible can automate the configuration, management, and coordination of diverse systems and devices in the manufacturing environment. For instance, if a sensor detects a deviation in the manufacturing process, Ansible can facilitate immediate adjustments in the robotic systems to correct the deviation, ensuring optimal production quality and efficiency.

Real-time data analysis and optimization: Ansible can enable the automation of real-time data analysis, allowing

manufacturers to monitor and optimize every aspect of the production process continually. For example, Ansible can automate data collection and analysis from various sensors, enabling predictive maintenance of machinery, reducing downtime, and extending equipment life.

Adaptive manufacturing processes: Through the automation capabilities of Ansible, manufacturers can implement adaptive manufacturing processes that self-optimize based on real-time data. This means production lines can automatically adjust to changes in demand, material availability, and other variables, ensuring optimal resource utilization and output.

Security and compliance: In the interconnected environment of Industry 4.0, securing sensitive data and ensuring system integrity is critical. Ansible can automate the deployment of security measures and compliance checks, safeguarding systems against vulnerabilities and ensuring the integrity and confidentiality of data.

Conclusion: By leveraging Ansible in Industry 4.0, manufacturers can automate and optimize the configuration and coordination of diverse systems and devices, enabling real-time monitoring, analysis, and adaptation of manufacturing processes. This results in enhanced manufacturing efficiency, productivity, and quality, allowing manufacturers to adapt to market demands swiftly and effectively.

Telecommunications

Every telecommunications company faces two common challenges as they strive to remain competitive: how to enhance the customer experience and how to improve network efficiency. As customers demand more personalized experiences, service providers must find ways to transform data into new services and proactively deliver them to their

customers. At the same time, service providers are looking for ways to reduce the amount of human interaction required to manage and maintain the expanding number of endpoints across their network.

Edge devices for telco companies include any device that is connected to their network, typically found in customers' homes and offices. Like many connected devices, these are producing data that can provide valuable insights, which can be used to improve the customer experience through automation. For example, service providers are collecting telemetry data from their customers on an ongoing basis. Automation can help proactively turn that data into opportunities to reach out to customers who might be experiencing connectivity issues and provide aid before they raise concerns. Another opportunity for automation is in the delivery of new services. In an ideal state, service providers can simply send a device to a customer's home or office that they can plug in and run, without the need for a technician on-site. Automating service delivery not only improves the customer experience, but also creates a more efficient network maintenance process, with the potential of reducing costs.

Refer to *Figure 8.5*, showcasing the dynamic telecommunication sector, an area ripe for innovation with the integration of edge computing and Red Hat Ansible:

Figure 8.5: *Telecommunication industry use case*

Advantages of automating at edge brings to the telecom industry:

- Use telemetry data to proactively support customers.
- Improve the overall customer experience.
- Shorten the time to deploy new services.
- Reduce or mitigate network downtime by automatically deploying updates and patches.
- Increase network efficiency and limit the need for human intervention.

Financial services and insurance

Financial agencies recognize the need for IT modernization to be more agile. Customers are demanding more personalized financial services and tools that can be accessed from virtually anywhere, including the customer's mobile devices. To meet this need, financial services providers must find ways to accelerate and de-risk the delivery of new services, scale with customer demand, and provide uninterrupted uptime while maintaining strict security standards and adherence to changing government regulations.

Banks and new entrants in the financial services industry can benefit from extending automation to the edge. Whether it is an **Automated Teller Machine** (**ATM**) in a bank branch, a self-service kiosk outside the bank, an application running on a customer's device, or managing the IT infrastructure across branches, automation provides the speed and access that customers want, with the reliability and scalability that financial service providers need. Imagine that a bank launches a self-service tool to help their customers find the right offering, which could be insurance, a mortgage, or a credit card. Automation at the edge not only makes it possible for that bank to scale the new service but also

automatically update and meet strict industry security standards without impacting the customer experience.

Take a look at *Figure 8.6*, demonstrating the complexity of the financial sector, a field where edge computing and Red Hat Ansible can provide impactful solutions:

Figure 8.6: *Financial industry use case*

Advantages of automating at the edge in the financial industry:

- Provides a more personalized customer experience.
- Shortens the time it takes to deploy new services.
- Offers reliable services with minimal downtime.
- Maintains strict security standards and adherence to changing regulations.

Let us understand this more with an example scenario:

Scenario: Telecommunication providers manage complex networks that consist of a myriad of technologies, devices, and standards. They need to ensure uninterrupted services, optimal network performance, and security, amidst the challenges posed by rapidly evolving technologies,

increasing customer demands, and the constant threat of security breaches.

Challenge: Managing, configuring, and updating diverse network devices and systems, ensuring optimal network performance, security, and compliance, and rapidly deploying new services and updates are significant challenges in telecommunications. The need for real-time responsiveness, high availability, and adaptability is crucial.

Ansible's solution: Ansible can automate the deployment, configuration, and management of diverse network devices and systems, ensuring consistency, optimal performance, and security. For example, if there is a need to update the firmware on multiple devices across different locations, Ansible can automate this task, reducing the time and resources required and eliminating the risk of errors.

Real-time network optimization: Ansible can automate real-time network analysis and optimization, enabling telecommunication providers to monitor and optimize network performance continually. For instance, if a bottleneck is detected in the network, Ansible can automate the necessary adjustments to alleviate the congestion, ensuring uninterrupted services and optimal network performance.

Rapid device deployment: With Ansible, telecommunication providers can rapidly deploy new services and updates, responding swiftly to customer demands and market trends. For example, if a new service is to be rolled out, Ansible can automate the deployment process, ensuring that the service is available to customers in the shortest time possible.

Security and compliance: In the telecommunications industry, ensuring the security and compliance of network systems is paramount. Ansible can automate the deployment of security policies and compliance checks,

safeguarding the network against vulnerabilities and ensuring adherence to industry standards and regulations.

Conclusion: By integrating Ansible in the telecommunications domain, service providers can ensure optimal network performance, rapid service deployment, and enhanced security and compliance. The automation capabilities of Ansible enable telecommunication providers to respond swiftly to technological advancements, customer demands, and market trends, ensuring sustained competitiveness in the dynamic telecommunications landscape.

Smarter cities

From an IT perspective, smarter cities, also known as smart cities, refer to urban areas that utilize different types of electronic methods and sensors to collect data. Insights gained from that data are used to manage assets, resources, and services efficiently. In return, this data is used to improve operations across the city.

Key technologies in a smart city include:

- **Internet of Things devices**: IoT devices are used to collect and analyze data from urban areas. This can range from monitoring traffic patterns, weather conditions, or energy usage in buildings.
- **Artificial intelligence and machine learning**: These technologies are used to process the vast amounts of data collected by IoT devices. They help make sense of this data and can provide predictions and insights to inform city management.
- **Big data**: The management and analysis of large volumes of data in real-time is a core aspect of smart cities. The data collected is processed and interpreted to improve city operations, planning, and policy.

- **Connectivity**: This includes high-speed internet across the city to facilitate real-time data transfer and communication between different systems and services.
- **Cloud computing**: It allows cities to use software and hardware managed by third parties to store, manage, and process data.
- **Cybersecurity**: Security is a top priority in smart cities due to the sensitive nature of the data collected. Advanced cybersecurity measures are put in place to protect the data and systems from potential threats.

Cities are always changing and so are the needs of their citizens. From rising traffic congestion to garbage pickup to responding to emergencies, cities rely on a massive amount of data to respond to immediate needs and plans. To improve services while increasing efficiency, many municipalities are incorporating technologies such as IoT and AI/ML to monitor and respond to issues affecting public safety, citizen satisfaction, and environmental sustainability. Early smart city projects were constrained by the technology of the time. Originally, devices were connected through copper cables or optic fiber, which limited the number of devices that could be deployed. However, the roll out of 5G networks and new communications technologies still to come will not only continue to increase speeds, but also make it possible to connect more devices. To scale edge capabilities effectively, smart cities need to automate. See *Figure 8.7*, depicting the vision of smarter cities, a concept that can be realized through the power of edge computing and Red Hat Ansible. Please refer to the following figure:

Figure 8.7: *Smarter cities industry use case*

To illustrate the opportunity for automation in smart cities, let us consider an edge device like traffic cameras. A single traffic camera has the potential to capture data about any number of variables such as road conditions, weather, traffic patterns, congestion, and emergencies. Edge computing helps these devices gather and process the data in near-real time. Data is then sent back to a datacenter so both technicians and automated processes can make decisions and act. To multiply this process across a large city, it would quickly become impossible for the technicians to respond to the data in a reasonable time. The added burden of security, patches, and updates would not only be unfeasible, but it would also increase the risk of security threats and service disruptions.

For example, a traffic camera could detect an accident at an intersection and automatically adjust traffic lights to block traffic while also notifying emergency services, all without human assistance. With these essential first-response steps taken, a team member can better assess the situation and reopen lanes as necessary once it is safe to do so.

Advantages of automating at edge bring to smart cities:

- Reduces the time it takes to deploy services.
- Improves safety and service delivery for citizens.
- Plans future infrastructure developments.

The following scenario emphasizes how Ansible can contribute to the realization of smarter cities by automating and optimizing diverse technologies and urban services, ensuring the cities are more sustainable, efficient, and livable.

Scenario: In smarter cities, the integration of various technologies, such as IoT devices, sensors, and advanced communication networks, is pivotal to managing urban services efficiently and enhancing the quality of life for its inhabitants. The deployment and management of these technologies can be a colossal task due to their disparate nature.

Challenge: Managing the extensive array of devices and technologies and ensuring their seamless interaction is crucial in smarter cities. They need to efficiently process vast amounts of data, enable real-time responses, ensure security and privacy, and adapt to evolving needs and technologies.

Ansible's solution: Ansible can automate the deployment, management, and orchestration of diverse technologies and devices in smarter cities. For example, if a city deploys a network of sensors for monitoring environmental conditions, Ansible can automate the configuration, data collection, and analysis of these sensors, allowing city administrators to make informed decisions swiftly.

Efficient urban services: By using Ansible, city administrators can automate and optimize urban services such as traffic management, waste collection, and emergency response. For instance, Ansible can automate

traffic light adjustments in real-time based on traffic conditions, optimizing traffic flow and reducing congestion.

Enhanced security and privacy: Security and privacy are paramount in smarter cities due to the extensive use of technology and the collection of data. Ansible can automate the deployment of security policies and protocols, ensuring the integrity and confidentiality of data collected by urban services.

Sustainable urban development: Ansible can aid in the implementation of sustainable solutions by automating energy management in public buildings, optimizing water consumption, and enabling the efficient use of urban resources, contributing to the city's sustainability goals.

Conclusion: In the context of smarter cities, Ansible serves as a versatile tool enabling cities to integrate and manage diverse technologies efficiently, automate urban services, enhance security, and contribute to sustainable urban development. Its automation capabilities allow cities to adapt to new technologies and evolving urban needs rapidly, paving the way for more livable, sustainable, and efficient urban environments.

Health care

Providing better care and enhanced services for patients is an ongoing pursuit in healthcare. As clinicians and paying organizations work to improve healthcare there is also a focus on ensuring they are prepared for the changes happening around the world. Twenty years ago, healthcare started to move away from hospitals toward remote care treatment options such as outpatient centers, clinics, and freestanding emergency rooms. The role of technology across such a dispersed network should aim to support clinicians as their needs evolve. One key aspect of addressing these needs is connecting different types of care

systems and providers across a diverse healthcare network, helping clinicians share and access timely, filtered, and patient-specific information. For example, when a patient is discharged from a hospital and needs to see a specialist outside the hospital's network, automation can ensure that the specialist receives the patient's history and current diagnosis prior to their visit. Automating these tasks ensures that the patient receives timely care, a better patient experience, and improved clinical outcomes. Observe *Figure 8.8*, highlighting the diverse aspects of the healthcare industry, an area where edge computing and Red Hat Ansible can drive significant advancements:

Figure 8.8: *Healthcare industry use case*

Beyond just supporting clinical systems, automation needs to be used to improve clinical decision-making in real time. This need is being accelerated by several factors, including more complex treatments being provided for patients in remote office settings and even in an at-home setting. Decision making involving medical transportation services, trauma services, and home-based care can also be improved and personalized based on patient data generated from wearables and a variety of other medical devices. Using automation, edge computing, and analytics, clinicians can convert data into new insights to help improve patient

outcomes while delivering financial and operational value. Traditionally, a sepsis diagnosis required a manual chart review, potentially delaying diagnosis of a condition that becomes *4% - 7%* more deadly every hour. Automation at the edge is already improving patient experiences and healthcare outcomes, including saving lives. Clinicians, data scientists, and IT professionals have collaborated on solutions that automate the collection and analysis of clinical data such as patient location, vital signs, and laboratory results. When the data indicates potential sepsis, automation at the edge coordinates workflow between the patient's nurses and the sepsis team, who may be in different locations or outside the hospital's network, helping them to initiate the appropriate care. This capability helps doctors detect sepsis indicators up to 20 hours earlier, saving thousands of lives.

Advantages of automating at the edge in the healthcare industry:

- Enables remote patient monitoring, thereby enhancing the interaction between clinicians and patients at earlier stages of disease progression. Assists in improving the delivery of care in locations where expertise is not available.
- Uses ML to improve the speed and accuracy of diagnoses and treatments provided to the patients.
- Tracks and deploys vaccines and other medication as needed.

The following scenario will help us understand this better. This highlights how Ansible can be pivotal in healthcare, offering solutions that are adaptable, secure, and efficient, ensuring seamless integration and management of technologies, which is crucial for enhanced patient care and operational efficiency.

Scenario: In the healthcare sector, the integration of advanced technologies and the **Internet of Medical Things** (**IoMT**) are pivotal in delivering enhanced patient care, efficient operations, and robust data security. Managing and securing these diverse technologies and ensuring their seamless interaction are crucial.

Challenge: The healthcare sector needs to manage a myriad of devices, applications, and systems, each serving a unique purpose, from patient care to data analysis. The sector requires solutions that can streamline workflows, ensure data privacy and security, and adapt to the ever-evolving healthcare technologies and increasing data volumes.

Ansible's solution: Ansible can automate the deployment, management, and orchestration of diverse healthcare technologies and systems. For example, if a hospital deploys numerous IoMT devices for patient monitoring, Ansible can automate the configuration, data collection, and analysis of these devices, enabling healthcare providers to deliver personalized and timely care.

Efficient patient care: By using Ansible, healthcare providers can optimize patient care services such as automated patient monitoring, real-time data analysis, and rapid response to medical emergencies. For instance, Ansible can automate the data collection and analysis from wearables and sensors, enabling healthcare providers to monitor patients' health conditions in real-time and make quick, informed decisions.

Enhanced data security and compliance: In healthcare, maintaining data security and compliance with regulations such as HIPAA is paramount. Ansible can automate the deployment of security measures, enforce compliance policies, and ensure the secure handling and storage of sensitive patient data.

Streamlined operations and workflows: Ansible's automation capabilities can streamline various healthcare operations, from appointment scheduling to medical billing, reducing administrative overhead and allowing healthcare professionals to focus more on patient care.

Innovation and research: Ansible facilitates research and innovation in healthcare by automating the deployment and management of research tools and platforms, allowing researchers to focus on developing new healthcare solutions and medical treatments.

Conclusion: In healthcare, Ansible serves as a linchpin, enabling the sector to integrate and manage a plethora of technologies, automate workflows, and enforce data security and compliance. Its automation capabilities allow healthcare providers to adapt swiftly to new technologies and evolving healthcare needs, ensuring enhanced patient care, streamlined operations, and secure, compliant data management.

Conclusion

In conclusion, the transformative power of Red Hat Ansible in edge computing is undeniable. As we have explored through various industry use-cases, its impact is far-reaching, providing solutions that are not only robust and reliable but also scalable and easy to implement.

In the retail industry, Ansible has been instrumental in unifying the management of distributed edge sites, enhancing efficiency, and ensuring seamless customer experiences. Through automation, it has drastically simplified the management of numerous edge devices, enabling real-time inventory tracking, personalized customer experiences, and overall streamlined operations. In finance, Ansible has played a crucial role in ensuring data privacy and compliance at the edge, essential factors in an industry

heavily reliant on trust and confidentiality. By automating complex workflows, Ansible helps financial institutions maintain rigorous security standards while enabling faster, more efficient data processing.

Moreover, Ansible has proven to be an invaluable tool for managing the growing complexity of edge computing in industries across the spectrum. From healthcare to manufacturing and telecommunications, its simplicity and scalability have paved the way for organizations to push boundaries, innovate, and adapt to the rapidly evolving digital landscape.

However, the potential of Ansible in edge computing extends beyond what we have discussed. The open-source nature of Ansible fosters a vibrant community, constantly driving new developments and improvements. The flexibility and adaptability of Ansible means that it can be tailored to solve unique industry-specific challenges as they arise.

In a future increasingly dominated by IoT devices and the need for real-time, data-driven decision-making, edge computing will continue to grow in significance. Red Hat Ansible, with its powerful automation capabilities and ease of use, will undoubtedly be at the forefront of this revolution, enabling businesses to fully harness the potential of edge computing.

This exploration of Ansible’s capabilities and potential is just the tip of the iceberg. As we move forward, the tool’s continuous evolution promises even greater possibilities for efficiency, innovation, and growth in edge computing.

Your journey through this chapter may end here, but the journey of Ansible in revolutionizing edge computing is just beginning. Stay tuned and keep exploring.

Join our book’s Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 9

Red Hat Ansible for Kubernetes and OpenShift Clusters

Introduction

OpenShift Container Platform (**OCP**) is the Red Hat implementation of Kubernetes, which not only provides the container platform for running containerized applications but is an end-to-end solution for running cloud-native containerized applications, starting from building the application, testing the application, and using a **DevSecOps-based pipeline** to deploy the application in production, in a most efficient and secure way.

While Ansible offers a powerful and agentless automation framework, especially integrated with OpenShift, it is essential to acknowledge other tools in the Kubernetes automation landscape. **Terraform**, for instance, is renowned for its infrastructure as code capabilities and its multi-provider support, making it a popular choice for setting up cloud infrastructures. **Pulumi** takes a different approach by allowing users to define their infrastructure using general-purpose programming languages. Both Terraform and Pulumi have their strengths, but Ansible stands out with its simplicity and its vast collection of modules and playbooks, making it particularly efficient for operational tasks and application deployment on platforms like OpenShift.

Ansible and OpenShift go hand in hand. We can set up **OpenShift 3 clusters** and **OpenShift 4 clusters** using Ansible right from provisioning the baseline infrastructure on vSphere and AWS, to deploying the OpenShift platform and deployment of Day2Ops applications. Single cluster, if deployed manually, would take over 11-12 hours with lots of runbooks or documents to follow and of course human errors but with Ansible, we can automate the process end to end and reduce the cluster setup time to under 2 hours without any human error. On top of it, this automation can be run in parallel by passing the right parameters to set up multiple clusters at a given time.

OpenShift comes up with something called the **`OpenShift Plus`** package, which provides additional capabilities by including the following components in addition to the standard OpenShift Container Platform:

- **RedHat OpenShift Platform**: Cloud native and containerized platform.
- **RedHat Advanced Cluster Manager** (**RHACM**): Manages attached OpenShift clusters by GitOps capabilities.
- **RedHat Advance Cluster Security** (**RHACS**): Elevates DevSecOps practices with advanced cluster security solutions.
- **RedHat Open Data Foundations** (**ODF**): Highly available storage.
- **RedHat Quay**: A scalable central registry to provide a single source of truth for available software and distribute them to multiple clusters efficiently.

In this chapter, we will discuss the use of Ansible with RHACS and RHACM in addition to OCP, such as:

- Network team can use the same simple, powerful, and agentless automation framework that IT operations and development are already using.
- Use a data model (A playbook or role) that is separate from the execution layer (Automation execution environments), that easily spans heterogeneous network hardware.
- Benefit from a wide variety of community and vendor-generated playbook and role content to help accelerate network automation projects.

Structure

In this chapter, we will go over the following topics:

- Ansible for Kubernetes operators
- Kubernetes modules in Ansible
- Managing Kubernetes/OpenShift clusters and Day2Ops operations using Ansible

Objectives

By the end of this chapter, readers should be able to understand the synergy between Ansible and Kubernetes, grasp the foundational knowledge of how Ansible complements Kubernetes operations and why it is a preferred tool for many organizations. We will learn how to implement Ansible for Kubernetes operators, delve into practical methods and techniques for utilizing Ansible in the creation and management of Kubernetes operators, streamlining cluster operations and application management. Utilize Kubernetes modules in Ansible, familiarize with specific Ansible modules tailored for Kubernetes operations, enabling efficient orchestration, scaling, and management of containerized applications. Manage Kubernetes and OCP clusters with Ansible and learn the best practices for provisioning, configuring, and maintaining both Kubernetes and OpenShift Container Platform clusters using Ansible. Gain insights into automating tasks, from setting up the baseline infrastructure to ensuring cluster health and resilience. We will also learn about Day2Ops with Ansible and transition from cluster setup to `Day2` operations. Explore how Ansible aids in monitoring, scaling, updating, and troubleshooting both Kubernetes and OCP clusters, ensuring optimal performance and minimal downtime.

Ansible for Kubernetes operators

In simple terms, operators automate the creation, configuration and management of Kubernetes native applications. Operators provide automation at every level of the stack, from managing the parts that make up the platform all the way to applications that are provided as managed services. There are multiple benefits of using operators, such as, operators providing repeatability of installation and upgrades, continuous health checks of each system component, and options for **over the air** (**OTA**) updates for OpenShift components.

Why Ansible for operators

As we see, Operators are a type of Kubernetes controller that manages complex, stateful applications on the platform. They typically automate the deployment, configuration, scaling, and management of these applications.

Ansible is a powerful automation tool that can be used to manage infrastructure and automate repetitive tasks. It is particularly well-suited for managing Kubernetes resources

because it is designed to be idempotent and declarative. This means that it can ensure that the desired state of a Kubernetes resource is always maintained, even if changes are made outside of Ansible.

Following are some specific reasons why Ansible is a good fit for creating Kubernetes operators:

- **Idempotence**: Ansible's idempotent nature means that it can ensure the desired state of Kubernetes resources, including operators, is maintained. This makes it easy to automate the deployment, scaling, and management of Kubernetes operators.
- **Declarative configuration management**: Ansible's declarative syntax makes it easy to define and manage complex configurations, including Kubernetes resources. This can simplify the process of creating and managing Kubernetes operators.
- **Extensibility**: Ansible is highly extensible and can be easily customized to meet specific use cases. This makes it a good choice for creating custom Kubernetes operators that are tailored to specific applications.
- **Community support**: Ansible has a large and active community of contributors who have created many useful modules and plugins for working with Kubernetes. This can simplify the process of creating Kubernetes operators and reduce the amount of custom coding required.

Overall, Ansible's idempotence, declarative configuration management, extensibility, and community support make it a strong choice for creating Kubernetes operators.

Creating Kubernetes operator with Ansible

To create a Kubernetes operator with Ansible, follow the given steps:

1. **Installing operator sdk**: You will have `operator-sdk` binaries installed on your machine in order to build the operator. Following are a few options for installing an operator:

    a. For Linux based OS run the following commands:

    ```
    curl -LO https://github.com/operator-framework/operator-sdk/releases/download/v1.16.0/operator

    tar -xzf operator-sdk_v1.16.0_linux_amd64.tar.gz

    sudo mv operator-sdk /usr/local/bin/
    ```

    b. For Mac OS, run the following commands:

    ```
    brew install operator-sdk
    ```

    Once you have `operator-sdk` installed, you can verify the `install` and `version` using the `operator-sdk version` command, as shown:

    ```
    operator-sdk version

    operator-sdk version: "v1.13.1", commit: "754b1db5a2f3b27aaabc17e3e14b3f720557c751", kubernete
    ```

2. **Creating Ansible-based operator using operator sdk**: Now let us create an operator `memcached-operator` using `operator-sdk`. We will run the following command:

    ```
    mkdir -p $HOME/projects/memcached-operator
    ```

    a. Now let us add `cd` to the directory

    ```
    cd $HOME/projects/memcached-operator
    ```

    b. Now from this directory, run the following command with the ansible plugin to initialize the project, as shown:

```
operator-sdkinit --plugins=ansible --domain=example.com
```

The following shows how the file structure will look now:

```
memcached-operator/
├── Dockerfile
├── Makefile
├── PROJECT
├── config
│   ├── default
│   ├── manager
│   ├── prometheus
│   ├── rbac
│   ├── samples
│   └── scorecard
├── playbooks
├── roles
├── watches.yaml
└── molecule
    └── default
        ├── converge.yml
        ├── molecule.yml
        ├── prepare.yml
        └── verify.yml
```

In this structure, the **PROJECT** file would contain information about your project, including the domain and the layout, which should be **ansible.sdk.operatorframework.io**. The **Dockerfile** contains instructions to build the operator image. The **config** directory contains Kubernetes manifests for deploying your operator. The **roles** directory will contain your Ansible roles, and the **playbooks** directory will contain Ansible playbooks. The **watches.yaml** file defines how the operator monitors the resources. The **molecule** directory contains Molecule scenarios for end-to-end testing of your operator.

Following is the output of **cat PROJECT** which shows how the project file should look like:

```
domain: example.com
layout: ansible.sdk.operatorframework.io/v1
projectName: memcached-operator
resources:
- api:
    crdVersion: v1
```

```
    namespaced: true
  group: cache
  kind: Memcached
  version: v1alpha1
```

Note: The resources section might be different or even empty if you did not add an API for the Memcached kind yet. The projectName field reflects the name of your operator. The layout field indicates that the project uses the Ansible layout. The domain field should match whatever domain you have provided when initializing the project.

3. Now let us create an API using the command:

   `operator-sdk create api --group cache --version v1 --kind Memcached --generate-role`

   This would scaffold out the API and generate a default Ansible role which fulfills the API and the output of command should look like:

```
Writing scaffold for you to edit...
api/v1/memcached_types.go
controllers/memcached_controller.go
Update dependencies in go.mod
Running make:
$ make
/Users/user/go/bin/controller-gen object:headerFile="hack/boilerplate.go.t
go fmt ./...
go vet ./...
go build -o bin/manager main.go
```

4. Now after the API is created, our directory structure should be updated with the following objects:

   a. A CRD for **`memcache`**

   b. Manager

      This is a program that reconciles the state of the cluster to the desired state of the cluster. This is done by the following:

      - A reconciler, which is Ansible role or Ansible Playbook.
      - A **`watches.yaml`** file which connects the **`memcached`** CRD to the Ansible role for **`Memcached`**.

5. Now let us modify the **`manager`** to make sure action is taken every time when **`memcached`** CRD is created, updated or deleted.

   For this, update the task file as shown:

```
# tasks file for Memcached
- name: start memcached
  community.kubernetes.k8s:
    definition:
```

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: '{{ ansible_operator_meta.name }}-memcached'
  namespace: '{{ ansible_operator_meta.namespace }}'
spec:
  replicas: "{{size}}"
  selector:
    matchLabels:
      app: memcached
  template:
    metadata:
      labels:
        app: memcached
    spec:
      containers:
      - name: memcached
        command:
        - memcached
        - -m=64
        - -o
        - modern
        - -v
        image: "docker.io/memcached:1.4.36-alpine"
        ports:
          - containerPort: 11211
```

So, the **memcached** role ensures that **memcached** deployment exists and also sets the deployment size.

6. Set the default values for variables by updating **roles/memcached/defaults/main.yml**, as shown:

```
---
# defaults file for Memcached
size: 1
```

Following **yaml** is the output of **cat config/samples/cache_v1_memcached.yaml,** which shows how to update the sample **memcached** resource:

```
apiVersion: cache.example.com/v1
kind: Memcached
metadata:
  name: memcached-sample
spec:
  size: 1
```

In this example, a Memcached custom resource named **memcached-sample** is created with **size: 1**, indicating that **1 replica** of **Memcached** is desired. Your actual sample file may vary based on the specifics of your Memcached API.

Kubernetes modules in Ansible

K8s modules in Ansible refer to a set of modules that enable users to interact with Kubernetes/OpenShift clusters using Ansible playbooks. These modules allow you to automate various tasks such as deploying, scaling, and managing applications running on Kubernetes/OpenShift clusters.

Here, we will see how we can use the **K8s** module to deploy and manage Kubernetes objects. The prerequisite is that you need to have **OpenShift** or **Kubernetes cluster** running as that setup is not covered in this section.

Creating Kubernetes object using ad hoc commands

Here, we will see how you can create a Kubernetes object using the Ansible ad hoc command using the **K8s** module. So, login to the **Kubernetes/OpenShift cluster** from the Ansible controller host. Let us see how you can create a namespace/project called **demo-namespace**:

```
ansible web -m k8s -a "name=demo_namespace state=present kind=Namespace api_version=v1"
```

Here, you can see that we are using the **K8s** module (**-m k8s**) to create a namespace (**name=demo_namespace**). The state is specified as **present** which means we are creating the namespace. Similarly, if you have to delete a namespace you can use state as **absent**.

Creating Kubernetes object using Ansible playbook

When you have multiple Kubernetes objects to deploy or manage, then it will make more sense to use playbooks instead of running ad hoc commands.

Let us use the Ansible Playbook for creating a **namespace** called **nginx** on the deployed OpenShift 4 cluster. Refer to the following Ansible Playbook:

```
---
- hosts: localhost
  gather_facts: false
  tasks:
    - name: Create a namespace in OpenShift
      community.kubernetes.k8s:
        name: nginx
        api_version: v1
```

```
    kind: Namespace
    state: present
```

This playbook creates a namespace named **nginx**. The **state: present** means the **namespace** will be created if it does not exist.

Note: You need to install the community.kubernetes collection with ansible-galaxy if you have not done so.

You need to be authenticated to the OpenShift cluster. If you are running this on the same machine where **oc** commands are working, then Ansible should pick up and use the same configuration. If you are running this from a remote machine, you will need to provide the authentication details to Ansible, possibly using a **kubeconfig** file:

1. Now let us create a, Ansible Playbook which will do the following on OpenShift 4 cluster which is already running:
   a. Create a namespace **nginx**
   b. Deploy nginx on namespace **nginx**
   c. Create a service for **nginx**
   d. Expose the service nginx as **route**
   e. Verify if **nginx** can be accessed
2. Now let us take this approach step by step:

   The following code shows the Ansible task which creates a **namespace** named **nginx** in your **OpenShift cluster**. The **state: present** ensures the **namespace** is created if it does not exist.

```
- name: Create a namespace in OpenShift
  community.kubernetes.k8s:
    name: nginx
    api_version: v1
    kind: Namespace
    state: present
```

3. The following Ansible task deploys a NGINX deployment in the **nginx namespace** you just created. The **replicas: 1** means only one **pod** will be deployed. The **pod** will have one container that runs the **nginx:1.14.2** image and exposes **port 80**.

```
- name: Deploy nginx in the 'nginx' namespace
  community.kubernetes.k8s:
    namespace: nginx
    definition:
      kind: Deployment
      apiVersion: apps/v1
      metadata:
        name: nginx
```

```
    spec:
      replicas: 1
      selector:
        matchLabels:
          app: nginx
      template:
        metadata:
          labels:
            app: nginx
        spec:
          containers:
          - name: nginx
            image: nginx:1.14.2
            ports:
            - containerPort: 80
```

4. The next Ansible task, which is shown as follows, creates a service named **`nginx-service`** in the **`nginx namespace`**, which exposes the NGINX deployment to other **`pods`** within the cluster. The service uses a selector **`app: nginx`** to target the NGINX pods, and it listens on **`port 80`**, forwarding traffic to the same **`port`** on the NGINX pods.

```
- name: Create a service for nginx
  community.kubernetes.k8s:
    namespace: nginx
    definition:
      kind: Service
      apiVersion: v1
      metadata:
        name: nginx-service
      spec:
        selector:
          app: nginx
        ports:
        - protocol: TCP
          port: 80
          targetPort: 80
```

5. Now, it is time to expose the NGINX services we created above so NGINX can be accessed externally. The following task creates a **`route`** named **`nginx-route`** in the **`nginx`**

`namespace`, which exposes the `nginx-service` outside of the cluster. This is equivalent to running `oc` expose `svc/nginx-service` on the command line, the `route` allows external traffic to reach the NGINX pods:

```
- name: Expose the nginx service as a route
  community.kubernetes.k8s:
    namespace: nginx
    definition:
      apiVersion: route.openshift.io/v1
      kind: Route
      metadata:
        name: nginx-route
      spec:
        to:
          kind: Service
          name: nginx-service
```

6. Now, let us verify if NGINX can be accessed externally or not. The following Ansible task sends a GET request to the URL of the exposed route. The URL is built by using the route name (`nginx-route`), the namespace (`nginx`), and your cluster's domain name. The `status_code: 200` expects a `200` HTTP status code in the response, which would indicate that the NGINX service is accessible.

```
- name: Verify if nginx can be accessed
  uri:
    url: http://nginx-route-nginx.apps.your-cluster-domain.com
    status_code: 200
```

Here is how the complete playbook will look like. Also note that, in the end, we have clubbed a task that uses `uri` module of Ansible for verifying the NGINX application and if its deployed correctly.

```
---
- hosts: localhost
  gather_facts: false
  tasks:
    - name: Create a namespace in OpenShift
      community.kubernetes.k8s:
        name: nginx
        api_version: v1
        kind: Namespace
        state: present
```

```
- name: Deploy nginx in the 'nginx' namespace
  community.kubernetes.k8s:
    namespace: nginx
    definition:
      kind: Deployment
      apiVersion: apps/v1
      metadata:
        name: nginx
      spec:
        replicas: 1
        selector:
          matchLabels:
            app: nginx
        template:
          metadata:
            labels:
              app: nginx
          spec:
            containers:
            - name: nginx
              image: nginx:1.14.2
              ports:
              - containerPort: 80

- name: Create a service for nginx
  community.kubernetes.k8s:
    namespace: nginx
    definition:
      kind: Service
      apiVersion: v1
      metadata:
        name: nginx-service
      spec:
```

```
        selector:
          app: nginx
        ports:
        - protocol: TCP
          port: 80
          targetPort: 80

  - name: Expose the nginx service as a route
    community.kubernetes.k8s:
      namespace: nginx
      definition:
        apiVersion: route.openshift.io/v1
        kind: Route
        metadata:
          name: nginx-route
        spec:
          to:
            kind: Service
            name: nginx-service

  - name: Verify if nginx can be accessed
    uri:
      url: http://nginx-route-nginx.apps.your-cluster-domain.com
      status_code: 200
```

GitOps workflow using OpenShift GitOps operator

The **`OpenShift GitOps`** operator is essentially a tailored version of ArgoCD optimized for OpenShift. Below is an example Ansible Playbook that leverages the **`k8s`** module to implement a GitOps workflow using the **`OpenShift GitOps`** operator, refer to the following code:

```
---
- name: Implement GitOps workflow with OpenShift GitOps Operator
  hosts: localhost
  tasks:

  - name: Ensure OpenShift GitOps Operator is installed
    k8s:
```

```
    api_version: operators.coreos.com/v1alpha1
    kind: Subscription
    name: openshift-gitops-operator
    namespace: openshift-operators
    state: present
    definition:
      spec:
        channel: stable
        installPlanApproval: Automatic
        name: openshift-gitops-operator
        source: redhat-operators
        sourceNamespace: openshift-marketplace

- name: Create a Git repository for nginx manifests
  git:
    repo: 'https://github.com/your-username/nginx-manifests.git'
    dest: '/path/to/local/directory'
  register: git_repo

- name: Create an Application CR for GitOps operator to deploy nginx
  k8s:
    definition:
      apiVersion: argoproj.io/v1alpha1
      kind: Application
      metadata:
        name: nginx-gitops
        namespace: openshift-gitops
      spec:
        source:
          repoURL: 'https://github.com/your-username/nginx-manifests.git'
          targetRevision: HEAD
          path: manifests/nginx
        destination:
          server: 'https://kubernetes.default.svc'
```

```
          namespace: 'nginx'
        project: default
  when: git_repo is changed

- name: Sync the Application to ensure nginx is deployed
  k8s:
    definition:
      apiVersion: argoproj.io/v1alpha1
      kind: Application
      metadata:
        name: nginx-gitops
        namespace: openshift-gitops
      spec:
        syncPolicy:
          automated:
            prune: true
            selfHeal: true
```

Note:

- **The first task ensures that the OpenShift GitOps operator is installed in the cluster.**
- **The playbook then checks out an example nginx manifests repository to a local directory.**
- **The when: git_repo is changed condition ensures that the Application CR is only created if there were changes in the nginx manifests repository.**
- **As with the previous example, the syncPolicy ensures that the nginx deployment is kept in sync with the manifests in the git repository.**

Do adjust the repository URL, namespaces, and other specifics according to your particular setup.

Managing Kubernetes/OpenShift clusters and Day2Ops operations with Ansible

Day 2 operations, or Day 2Ops, refer to the ongoing maintenance and management tasks required to ensure the smooth and reliable operation of a Kubernetes/OpenShift cluster. Following are some examples of Day 2Ops operations on a Kubernetes/OpenShift cluster:

- **Scaling**: Day 2Ops involves scaling the Kubernetes/OpenShift cluster to meet the changing demands of the applications running on it. This may involve adding or removing nodes, adjusting the number of `replicas` for deployments, or changing the resource requests and limits for pods.
- **Upgrades**: Kubernetes/OpenShift releases regular updates with bug fixes, security patches, and new features. Day 2Ops involves planning and executing upgrades to keep the cluster up-to-date with the latest version of Kubernetes/OpenShift.

- **Monitoring**: Monitoring the health and performance of the Kubernetes/OpenShift cluster is essential for identifying and resolving issues before they cause service disruptions. Day 2Ops involves configuring and maintaining monitoring tools such as **Prometheus** and **Grafana** to track key metrics and alert on anomalies.
- **Logging**: Kubernetes/OpenShift generates a lot of logs, including container logs, node logs, and cluster logs. Day 2Ops involves managing log collection, storage, and analysis using tools such as **Fluentd** or **Elasticsearch**.
- **Backup and recovery**: Kubernetes clusters contain critical data and configurations that need to be backed up regularly to ensure disaster recovery. Day 2Ops involves setting up backups and testing recovery procedures to ensure they work correctly.
- **Security**: Kubernetes clusters are a prime target for cyber-attacks, and security needs to be a top priority. Day 2Ops involves implementing security best practices such as RBAC, network policies, and using secure container images.
- **Configuration management**: Configuration management involves managing the Kubernetes/OpenShift cluster's configuration files and ensuring they are consistent across all nodes. Day 2Ops involves using tools such as **GitOps** or **Helm** to manage and deploy configurations.
- **Resource optimization:** Kubernetes/OpenShift clusters can be costly to operate, especially if they are not optimized for resource usage. Day 2Ops involves monitoring and tuning the cluster's resource usage to reduce costs while maintaining performance.

These are just a few examples of Day 2Ops operations on a Kubernetes/OpenShift cluster. In general, Day 2Ops involves ongoing maintenance and management tasks that ensure the Kubernetes/OpenShift cluster is reliable, performant, and secure.

Here, we will go over how we can use Ansible to scale the nodes on the Kubernetes/OpenShift cluster. Scaling of nodes can be a critical step if the cluster needs to be scaled out for different reasons, such as performance improvement demands on the application or if specific POD needs to be run on a separate node altogether for security reasons and so on.

Following we can see the playbook which first logins to OpenShift cluster and then scales the machine sets by updating the replica count. Here, `MachineSets` is group of OpenShift nodes doing specific functions:

```
---
- hosts: localhost
  gather_facts: false
  vars_prompt:
    - name: "oc_username"
      prompt: "Enter OpenShift username"
      private: no
    - name: "oc_password"
      prompt: "Enter OpenShift password"
      private: yes
  tasks:
```

```
    - name: Login to OpenShift
      command:
        cmd: "oc login https://your-openshift-api-url --username={{
oc_username }} --password={{ oc_password }} --insecure-skip-tls-verify=true"
      register: login_result
      changed_when: false


    - name: Scale MachineSet in OpenShift
      community.kubernetes.k8s:
        api_version: machine.openshift.io/v1beta1
        kind: MachineSet
        name: "<your-machineset-name>"
        namespace: openshift-machine-api
        definition:
          spec:
            replicas: <desired-replica-count>
      when: login_result.rc == 0
```

In this playbook, the **`vars_prompt`** section prompts you for your OpenShift **`username`** and **`password`** when the playbook is run, and the entered values are stored in the **`oc_username`** and **`oc_password`** variables. These are then used in the **`oc`** login command.

The **`command`** module is used to run **`oc`** login with your provided **`username`** and **`password`**. The **`--insecure-skip-tls-verify=true`** option is included to skip certificate verification for the sake of this example, but it should not be used in production unless you understand the risks. Replace **`https://your-openshift-api-url`** with your actual OpenShift API URL.

The output from the **`oc`** login command is registered to the **`login_result`** variable. The return code (**`rc`**) from the command is checked before running the task to scale the **`MachineSet`**. If the login command is successful, **`rc`** will be **`0`**, and the playbook will proceed to scale the **`MachineSet`**.

Note: It is usually better to have Ansible use a pre-existing and appropriately secured kubeconfig file to authenticate with OpenShift or Kubernetes, rather than including authentication details directly in a playbook. Using Ansible to handle OpenShift logins can be tricky and is typically not recommended due to security concerns. Embedding credentials in an Ansible Playbook or passing them in as variables can expose sensitive data. However, if you understand the risks and still want to proceed, you can use the command or shell module in Ansible to execute the oc login command.

In the following playbook, we are doing the same action as above, but we are using a different approach. Instead of using the **`MachineSet`** object, we are running the **`oc scale`** command to increase the number of nodes on the **`OpenShift cluster`**. In this playbook, the **`command`** module is used to run the **`oc`** scale command, which scales the **`MachineSet`** to your desired **`replica`** count. Replace **`<your-machineset-name>`** with the name of your **`MachineSet`** and **`<desired-replica-count>`** with the number of **`replicas`** you want to scale to:

```
---
- hosts: localhost
```

```
  gather_facts: false
  vars_prompt:
    - name: "oc_username"
      prompt: "Enter OpenShift username"
      private: no
    - name: "oc_password"
      prompt: "Enter OpenShift password"
      private: yes
  tasks:
    - name: Login to OpenShift
      command:
        cmd: "oc login https://your-openshift-api-url --username={{
oc_username }} --password={{ oc_password }} --insecure-skip-tls-verify=true"
      register: login_result
      changed_when: false


    - name: Scale MachineSet in OpenShift
      command:
        cmd: "oc -n openshift-machine-api scale machineset/<your-machineset-
name> --replicas=<desired-replica-count>"
      when: login_result.rc == 0
```

In the following *Figure 9.1*, we are doing the same action as above, but we are using a different approach. Instead of using **`MachineSet`** object, we are running **`oc scale`** command to increase the number of nodes on the **`OpenShift cluster`**:

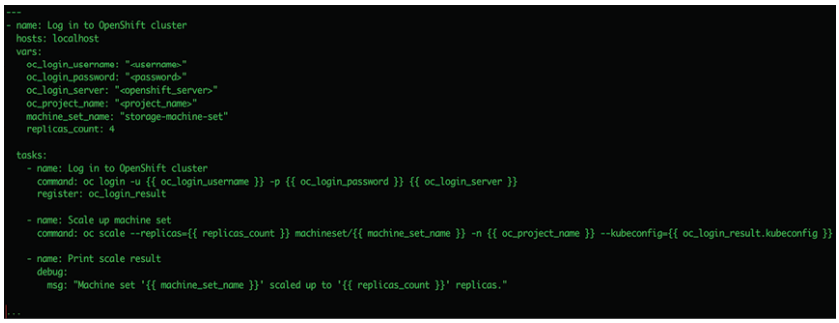

Figure 9.1: *Ansible playbook using oc scale command to increase the number of openshift nodes*

Now, let us cover one of major day2ops operations using Ansible. We will see how Ansible can be used to automate the **`OpenShift cluster`** upgrade process. The playbook is a single **`yaml`** file, but we are dividing the screenshots into 3 separate sections, so that each step is clear.

In the first section, which is shown in the following playbook, we are defining the variables, performing pre upgrade tasks and preparing the cluster to upgrade so that there are no

issues during the upgrade and no running workload is impacted.

```
- hosts: localhost
  vars:
    openshift_api_url: https://api.your-cluster-domain:6443
    openshift_username: your-username
    openshift_password: your-password
    desired_version: 4.x.y


  tasks:
    - name: Login to the cluster


      shell: oc login {{ openshift_api_url }} -u {{ openshift_username }} -p
{{ openshift_password }} --insecure-skip-tls-verify=true
      no_log: true


    - name: Drain all worker nodes
      shell: oc adm drain {{ item }} --ignore-daemonsets


      with_items: $(oc get nodes -l node-role.kubernetes.io/worker -o
jsonpath='{.items[*].metadata.name}')


    - name: Wait for nodes to drain
      pause:
        minutes: 10


    - name: Backup cluster data
      shell: oc get all --all-namespaces -o yaml > pre_upgrade_backup.yaml
```

This sets up the required variables such as the **`openshift_api_url`**, **`username`**, **`password`**, and the desired version for upgrade. It sets the stage for the tasks to follow:

- **Login to the cluster**: This task is responsible for logging in to your **`OpenShift cluster`** using the credentials provided. This step is essential as it gives the playbook access to perform tasks on the **`OpenShift cluster`**. We use **`no_log: true`** to prevent the logging of sensitive data such as your password.
- **Drain all worker nodes**: This task uses **`oc adm`** drain to mark all worker nodes as unschedulable and evicts all pods. Draining nodes is a standard practice before performing maintenance tasks like cluster upgrades. This ensures that running workloads are safely moved to other nodes.
- **Wait for nodes to drain**: Here, the playbook pauses for 10 minutes to allow the draining process to complete. This is crucial as rushing this process may lead to

complications during the upgrade.

- **Backup cluster data**: This task backs up all cluster resources across all namespaces and saves them in a file named `pre_upgrade_backup.yaml`. Backups are vital in case the upgrade process encounters issues and you need to roll back to the previous state.

Now, following is the second part which is an actual upgrade of the OpenShift 4 cluster:

```
    - name: Start the cluster upgrade

      shell: oc adm upgrade --to={{ desired_version }} --allow-explicit-
upgrade
```

This task performs the actual upgrade of the `OpenShift cluster` using `oc adm` upgrade. The `--allow-explicit-upgrade` option is used to force an upgrade to a specific version, which might not be the next minor version suggested by the cluster version operator.

The third part is to perform post upgrade activities which are shown as follows:

```
    - name: Wait for nodes to become ready

      shell: oc wait --for=condition=Ready nodes --all --timeout=300s


    - name: Uncordon all worker nodes

      shell: oc adm uncordon {{ item }}


      with_items: $(oc get nodes -l node-role.kubernetes.io/worker -o
jsonpath='{.items[*].metadata.name}')
```

So basically, we are performing the following two actions in this last step:

- **Wait for nodes to become ready**: After the upgrade is complete, this task waits for all nodes to become ready. This is done using the `oc` wait command which will pause the playbook until all nodes report a `Ready` condition. If the nodes do not become ready within the specified timeout (`300` seconds in this example), the command will fail.
- **Uncordon all worker nodes**: Once all nodes are ready, this task will make all worker nodes schedulable again using the `oc adm uncordon` command. Uncordoning allows the worker nodes to accept new pods, thereby making them fully operational post-upgrade.

We have seen the OpenShift upgrade process. It is good practice to have regular `etcd` backups before any upgrades. Below we will see how to schedule daily `etcd` backups.

Backing up `etcd` is a critical operation for maintaining a healthy and resilient Kubernetes cluster. Regular backups, combined with a clear disaster recovery plan, ensure the cluster's longevity and robustness against unforeseen data loss events.

What is etcd

`etcd` is a consistent and highly available key-value store used as Kubernetes' backing store for all cluster data. It stores the configuration data of the Kubernetes cluster, representing the state of the cluster at any given point in time. `etcd` is crucial for Kubernetes, and ensuring its data integrity and availability is vital for the Kubernetes cluster's resilience and health.

Why backup etcd

Following are the reasons why you should backup `etcd`:

- **Data safety**: Just like backing up any other database, backing up `etcd` ensures that you can restore your cluster's state in case of data loss events such as accidental deletion of data, hardware failures, or data corruption.
- **Disaster recovery**: If a catastrophic event causes the loss of all `etcd` instances in a cluster, the cluster's state can be restored from a backup. This makes backups essential for disaster recovery plans.
- **Rollbacks**: Sometimes, configurations or changes might need to be reverted. Having a snapshot of `etcd` before significant changes allows for rollbacks to a previous cluster state.

Now let us see how to setup `etcd` backups. Following is the playbook which does it end to end:

```
---
- name: Set up etcd backup on OpenShift
  hosts: localhost
  tasks:
    - name: Create a new project for etcd backup
      k8s:
        name: ocp-etcd-backup
        kind: Namespace
        api_version: v1
        definition:
          metadata:
            name: ocp-etcd-backup
            labels:
              app: openshift-backup
          annotations:
            description: "Openshift Backup Automation Tool"


    - name: Create Service Account for etcd backup
      k8s:
        namespace: ocp-etcd-backup
        kind: ServiceAccount
        definition:
          apiVersion: v1
          metadata:
            name: openshift-backup
```

```
- name: Create ClusterRole for etcd backup
  k8s:
    kind: ClusterRole
    definition:
      apiVersion: rbac.authorization.k8s.io/v1
      metadata:
        name: cluster-etcd-backup
      rules:
      - apiGroups: [""]
        resources:
        - nodes
        verbs: ["get", "list"]
      - apiGroups: [""]
        resources:
        - pods
        - pods/log
        - pods/attach
        verbs: ["get", "list", "create", "delete", "watch"]
      - apiGroups: [""]
        resources:
        - namespaces
        verbs: ["get", "list", "create"]

- name: Create ClusterRoleBinding for the Service Account
  k8s:
    kind: ClusterRoleBinding
    definition:
      apiVersion: rbac.authorization.k8s.io/v1
      metadata:
        name: openshift-backup
        labels:
          app: openshift-backup
      subjects:
```

```
      - kind: ServiceAccount
        name: openshift-backup
        namespace: ocp-etcd-backup
      roleRef:
        apiGroup: rbac.authorization.k8s.io
        kind: ClusterRole
        name: cluster-etcd-backup

- name: Create Backup CronJob
  k8s:
    namespace: ocp-etcd-backup
    definition:
      apiVersion: batch/v1
      kind: CronJob
      metadata:
        name: openshift-backup
        labels:
          app: openshift-backup
      spec:
        schedule: "* * * * *"
        concurrencyPolicy: Forbid
        successfulJobsHistoryLimit: 5
        failedJobsHistoryLimit: 5
        jobTemplate:
          spec:
            template:
              spec:
                containers:
                - name: backup
                  image: "registry.redhat.io/openshift4/ose-cli"
                  command:
                  - "/bin/bash"
                  - "-c"
                  - |
```

```
                        oc get no -l node-role.kubernetes.io/master --no-
headers -o name | head -n 1 | xargs -I {} -- oc debug {}  --to-
namespace=ocp-etcd-backup -- bash -c 'chroot /host rm -rf /home/core/backup
&& chroot /host mkdir /home/core/backup && chroot /host sudo -E mount -t nfs
<nfs-server-IP>:<shared-path> /home/core/backup && chroot /host sudo -E
/usr/local/bin/cluster-backup.sh /home/core/backup && chroot /host sudo -E
find /home/core/backup/ -type f -mmin +1 -delete'

                    restartPolicy: "Never"

                    serviceAccountName: openshift-backup
```

Here is a step-by-step explanation of the provided playbook:

1. Create a new project for **etcd backup**:
   - This task uses the **k8s** module to create a **new namespace** (project in OpenShift terms) called **ocp-etcd-backup**.
   - A label and annotation are attached to this **namespace** for identification and description purposes.
2. Create **Service Account** for **etcd backup**:
   - Within the namespace **ocp-etcd-backup**, a **Service Account** named **openshift-backup** is created.
   - Service accounts provide identity for processes that run in a **Pod**.
3. Create **ClusterRole** for **etcd backup**:
   - A **ClusterRole** is a way to group permissions (**verbs**) on resources. In this task, a **ClusterRole** named **cluster-etcd-backup** is defined.
   - This **ClusterRole** has permissions to:
     - **Get** and **list** **nodes**.
     - **Get**, **list**, **create**, **delete**, and **watch** **pods**, **pods/log**, and **pods/attach**.
     - **Get**, **list**, and **create** **namespaces**.
4. Create **ClusterRoleBinding** for the **Service Account**:
   - **ClusterRoleBindings** associate **ClusterRoles** with **Service Accounts**. It ensures that the permissions defined in the **ClusterRole** are granted to the specified **Service Account**.
   - In this task, the **openshift-backup Service Account** in the **ocp-etcd-backup** namespace is associated with the **cluster-etcd-backup ClusterRole**.
5. Create **backup CronJob**:
   - This task sets up a **CronJob** in the **ocp-etcd-backup** namespace.
   - A **CronJob** is a Kubernetes object that runs a job periodically on a given schedule, similar to the **cron** utility in Linux.
   - The **CronJob** is set to run every minute (**schedule: "* * * * *"**).
   - The job uses the **registry.redhat.io/openshift4/ose-cli** image.

- It fetches the name of the master node, runs a debug command on it, and performs several operations, such as clearing the `backup` directory, mounting an NFS share, running a backup script, and deleting files older than a minute.
- The `restartPolicy` is set to `Never`, meaning if the container fails, it would not be restarted.
- The job runs with the `openshift-backup` Service Account created earlier, ensuring it has the permissions defined in the associated `ClusterRole`.

The entire playbook automates the setup process for `etcd` backup in an OpenShift environment. It organizes resources, defines roles and permissions, and sets up a scheduled job to carry out the backup process. Using Ansible's `k8s` module, these tasks are carried out in an idempotent manner, meaning running the playbook multiple times would not have any adverse effects or duplicate the resources.

Conclusion

Ansible is a powerful, agentless, open-source automation tool that is extensively used in managing and automating IT infrastructures, including Kubernetes and OpenShift clusters. It provides a simple, human-readable language to define complex tasks and is extensible with hundreds of modules available to automate almost anything.

In the context of Kubernetes and OpenShift, Ansible shines in various areas:

Operator framework and Ansible operators: The operator framework allows developers to build extendable APIs into Kubernetes, effectively customizing their automation logic at the platform level. Ansible's operator SDK provides a streamlined, productive way to build these operators. This can lead to faster time-to-market and improved quality of custom resources as teams can focus on the actual logic rather than boilerplate code.

Throughout our discussion, we explored how you can create a `Memcached` operator using Ansible. The process involved initializing a new operator project, generating APIs and controllers, and defining the Memcached deployment's desired state. This shows how Ansible can be used to encapsulate and manage the entire application lifecycle right within the Kubernetes/OpenShift environment.

Cluster management: Managing an `OpenShift` or `Kubernetes cluster` includes various tasks like upgrades, scaling, and backup/restore operations. We have seen how Ansible can handle these operations with a high degree of automation. The provided Ansible playbook example for upgrading an `OpenShift 4 cluster` illustrates this clearly, making the upgrade process repeatable, less error-prone, and potentially more maintainable.

Configuration management: Ansible is an excellent tool for managing configurations across a cluster. It ensures that the systems in the infrastructure are consistent and in the desired state. Using Ansible for configuration management in a Kubernetes or OpenShift environment can greatly simplify the process of managing and deploying complex configurations across a large number of nodes.

Monitoring and debugging: Ansible also aids in monitoring and debugging. With its detailed execution information and idempotent nature, Ansible provides valuable insights into the system. This can be leveraged for troubleshooting issues, making it a valuable tool in the Kubernetes/OpenShift environment.

Integration: Ansible can be seamlessly integrated with other CI/CD tools in the Kubernetes and OpenShift ecosystem. For example, integration with Jenkins for continuous integration and continuous deployment pipelines can lead to fully automated deployment processes.

In conclusion, Ansible brings automation, consistency, and simplicity to the Kubernetes and OpenShift world, making it an indispensable tool for managing and operating these complex systems. Whether creating custom operators or managing the cluster’s lifecycle, Ansible has proven its mettle. The examples discussed are just the tip of the iceberg, and the possibilities with Ansible in the Kubernetes and OpenShift universe are vast and continuously evolving.

Looking forward to our next chapter, we will explore the Ansible Automation Platform, also known as Ansible Tower. This will cover how we can use Ansible at an enterprise level, manage complex workflows, maintain inventory in a dynamic environment, and use the built-in RBAC capabilities to safely manage and automate tasks at scale. Stay tuned!

Join our book’s Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 10

Using Ansible Automation Platform in Multi-Cloud

Introduction

Ansible Automation Platform (**AAP**), previously known as **Ansible Tower**, takes us beyond Ansible's command-line interface to a web user interface. It offers a more familiar environment for most customers, employers, and non-technical people. Tower allows sysadmins and DevOps/automation specialists to demonstrate the value and power of automation. It provides easier training and demos that lead to quicker acceptance and overall buy-in so that you can add this tool to your long-term automation strategy. Embedding AAP within an enterprise's architecture accrues multiple benefits, such as enhancing automation and business outcomes, optimizing collaboration and orchestration among various teams, and promoting innovation and growth at scale. The components such as Ansible core, Ansible Content Collections, Ansible content navigator, automation execution environments, and automation controller collectively facilitate these advantages, playing pivotal roles in the comprehensive automation solutions provided by AAP.

Each of these components, designed to optimize the automation initiatives of organizations, will be discussed in more detail, offering a clearer and more profound understanding of their significance in automation endeavors as we proceed further in this chapter. So, basically, by adopting AAP into the architecture, an enterprise can see the following benefits:

- Automation and acceleration in business outcomes
- Automation in collaboration and orchestration across the teams
- Automation in growth and innovation at scale.

All these benefits will become more clear and make more sense as we look further and deeper into AAP in this chapter.

Note:

- **This chapter is based on Ansible version 2.11, OpenShift cluster version 4.8 and Ansible Automation Platform operator version 2.1.**
- **Basic understanding of Kubernetes or OpenShift is required.**

Structure

In this chapter, we will learn the following topics:

- AAP components
- Understanding automation controller

- Reference architecture for AAP
- Installing the OpenShift operator on OpenShift 4 cluster
- Integrating AAP with source code management GIT repo
- Integrating AAP with **Red Hat Advance Cluster Management** (**RHACM**)
- Use cases
  - Implementing GitOps
  - Policy management

Objectives

The objective of this chapter is to understand how we can use Ansible beyond the configuration management tool. We will see how Ansible Automation Platform, aka **Ansible Tower,** takes Ansible to the next level to achieve GitOps, that too without changing our existing Ansible-based manifests and with very simple integrations. We will also see the power of Ansible using AAP, where existing automation can be triggered automatically if a certain condition occurs.

AAP components

AAP includes different components, which together provide a complete and integrated set of automation tools and resources. We will discuss each component as follows:

Embedding AAP within an enterprise's architecture accrues multiple benefits, like enhancing automation and business outcomes, optimizing collaboration and orchestration among various teams, and promoting innovation and growth at scale. The components such as Ansible core, Ansible content collections, Ansible content navigator, automation execution environments, and automation controller collectively facilitate these advantages, playing pivotal roles in the comprehensive automation solutions provided by AAP.

Each of these components, designed to optimize the automation initiatives of organizations, will be discussed in more detail, offering a clearer and more profound understanding of their significance in automation endeavors as we proceed further in this chapter.

In the following figure, the components of the automation controller are shown:

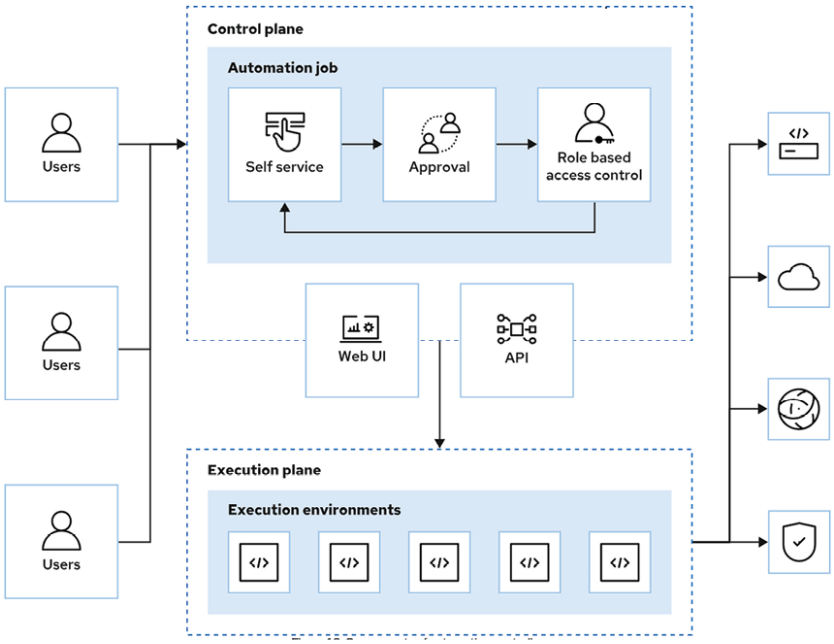

Figure 10.1: *Components of automation controller*

Ansible automation hub: It provides the capability to develop, manage and distribute the automation content. From Red Hat, we have public service at **console.redhat.com**, which provides access to Red Hat Ansible Certified Content Collections that we can download and use with `ansible-navigator` and `ansible-galaxy`.

Understanding automation controller

Automation controller: Red Hat AAP includes the component called `automation-controller`, which was previously known as `ansible-tower` and it provides a central hub that we can use to run our automation code.

Enterprise IT organizations need a way to define and embed automation workflows for other tools and processes. They also need reliable and scalable automation execution, and a centralized system that allows auditing. With the automation controller, companies automate with confidence and reduce automation drift and variance across the enterprise by standardizing how automation deploys in one centralized location.

Automation controller provides a framework for running and managing Ansible efficiently on an enterprise scale. It simplifies the administration involved with coordinating the execution of automation code. It also maintains organization security by introducing features such as a centralized web UI for playbook management, **role-based access control** (**RBAC**), and centralized logging and auditing. Its REST API ensures that the automation controller integrates easily with an enterprise's existing workflows and toolsets. The automation controller API and notification features make it particularly easy to associate Ansible Playbooks with other tools to enable continuous integration and deployment. It provides mechanisms to enable centralized use and control of machine credentials and other secrets without exposing them to end users of the automation controller.

Automation controller vs Ansible Tower

Till Red Hat Ansible Tower 3.8, it was hard to manage environment-specific dependencies on execution environments. Changes to the execution environment have to be made

directly to the Ansible Tower system node. If two playbooks need separate module versions, then we had to setup Python virtual environments (`venvs`) which created administration overhead and complexity. This challenge was solved by the automation controller. Here, we are not using Python installation or virtual environments, but instead we are using automation execution environments, which are containers that we can pull from the central container registry, install automation controller, and manage via web UI. Ansible developer `crs` can easily publish updates to the container registry and can create these exact execution environments and then use `ansible-navigator` to ensure it is compatible with their automation code.

Understanding automation controller: Automation controller, previously known as Ansible Tower, serves as the nucleus of the Ansible Automation Platform, providing the API and web UI for running and managing the automation jobs. It is where the control plane resides, orchestrating tasks and allocating resources, thus acting as the orchestrator of your automation scripts.

Control plane and execution plane separation: In the architectural realm of AAP, the separation of the control plane and execution plane is paramount. The control plane, housed within the automation controller, is responsible for managing tasks, policies, and scheduling, whereas the execution plane is where the tasks are actually carried out. This segregation is beneficial as it enables refined management and execution of tasks, allowing for better scalability and reducing the risk of system conflicts. It solves the challenge of running separate Python dependencies on the execution environment and default system by isolating the control node, which runs the web UI and API, and automation execution environments, which run as containers.

Automation execution environments: Automation execution environments are an integral component of the Ansible Automation Platform, providing a container image that encapsulates components like Ansible core, Ansible content collections, and so on. These environments aim to offer Ansible developers a consistent experience, mirroring that of Ansible 2.9. Developers also have the flexibility to use Ansible Builder to create their own execution environments.

Benefits over old architecture: The introduction of automation execution environments marked a significant leap from the old architecture. Unlike the previous setups where dependencies were managed at a system level, often leading to conflicts and inconsistencies, the new architecture ensures that each automation task runs within its isolated environment with its dependencies, eliminating conflicts and ensuring consistency and reliability across multiple runs. This encapsulation leads to enhanced reproducibility and reliability in automation, making it easier to manage and scale automation tasks across different environments.

Example workflow

In the evolved architecture of the Ansible Automation Platform, automation controller is crucial in maintaining a clear separation between the control and execution planes, providing a streamlined and conflict-free environment for automation tasks.

Imagine a scenario where you need to automate the deployment of a web application across different environments, each with its unique set of configurations and dependencies.

Control plane:

- Located within the automation controller, this is where you would define the tasks, policies, and scheduling.
- Here, you would create a job template, specifying the playbook to be run, the inventory to be used, and any required credentials and variables.

- The control plane ensures the correct orchestration of tasks and allocates resources as needed but does not execute the tasks itself.

Execution plane:

- Once the job is launched, the execution plane takes over. It is responsible for running the tasks defined in the control plane.
- In this isolated environment, the execution of the playbook occurs, deploying the web application with the specified configurations and dependencies.
- It interacts with the designated inventory, installing necessary software, configuring settings, and performing any other tasks defined in the playbook.

Benefits of decoupled planes:

- **Scalability**: The separation allows for easy scaling of each plane independently, accommodating growing needs without affecting the other plane's performance.
- **Conflict resolution**: The decoupled architecture prevents potential conflicts between system dependencies and those required by automation scripts, ensuring reliable and consistent execution.
- **Resource optimization**: Resources can be allocated and managed more effectively, ensuring optimal performance and reducing the risk of system overload.
- **Enhanced management**: It offers refined management and execution of tasks, enabling better organization and monitoring of automation jobs.

By maintaining a clear distinction between orchestrating tasks (Control plane) and executing them (Execution plane), the automation controller in Ansible Automation Platform offers a more reliable, scalable, and conflict-free environment for automation tasks, representing a significant advancement over the older, Ansible Tower architecture.

Figure 10.2 and *Figure 10.3* show the architectural differences between Ansible Tower and the automation controller. The main differences are as follows:

- Ansible Tower, as shown in *Figure 10.2*, is a centralized and monolithic application.
- *Figure 10.2* shows that Ansible Tower runs the control plane and execution plane on the same control node.
- Ansible Tower has comparatively poor scalability.
- Automation controller, as shown in *Figure 10.3*, is a decentralized and modular application.
- Control plane and execution plane for the automation controller are decupled, as shown in *Figure 10.3*. Also, it is a containerized environment.
- Automation controller is highly scalable, and one can request and create execution environments using container orchestrators.

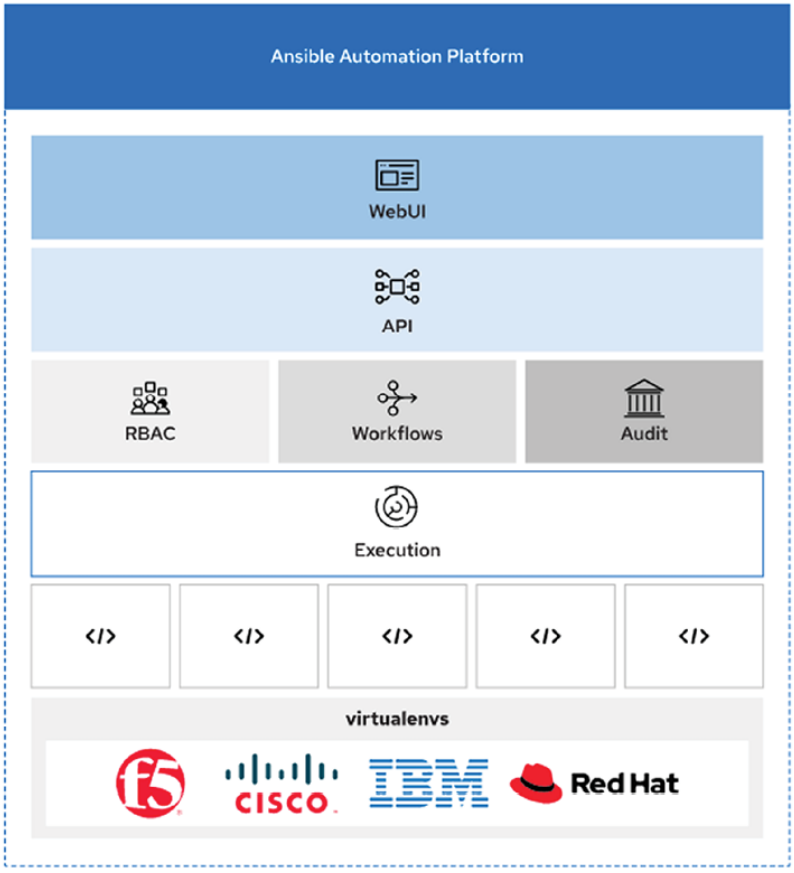

Figure 10.2: *Ansible Tower architecture*

Automation controller is shown in *Figure 10.3:*

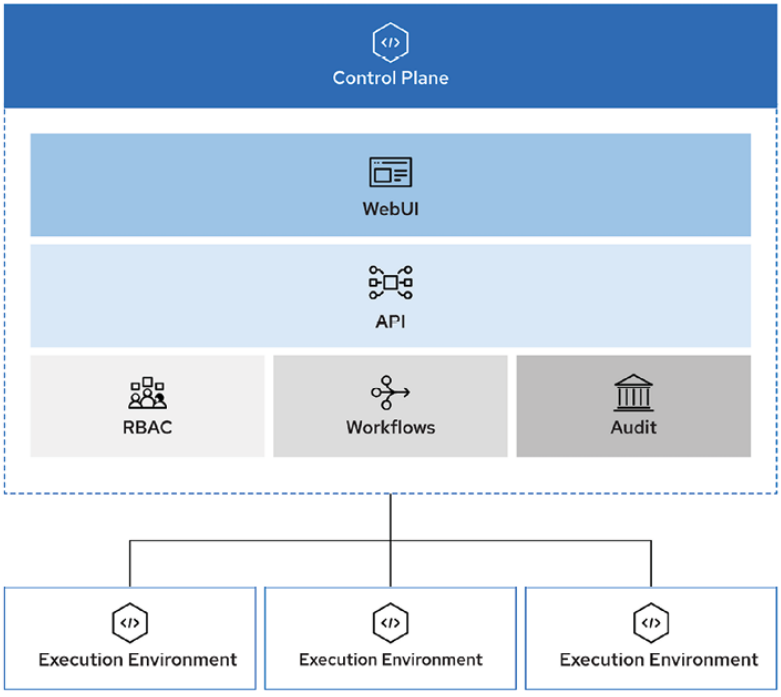

Figure 10.3: *Automation controller architecture*

Automation controller features

Below, certain features of automation controllers are discussed. These features are crucial for controlling, securing, and managing automation within an enterprise environment.

Visual dashboard: This is a webUI from the automation controller which provides a summary of the Ansible environment in the enterprise's environment. Ansible administrators can view and manage their host inventories and check the results of recent job executions. This UI provides better security, increases performance, and improves observability with new filtering capabilities and distinct views.

Role-based access control (RBAC): This system simplifies user access management to automation controller objects and resources like organizations, projects, inventories, jobs and so on.

Graphical inventory management: Automation controller user interface can be used by Administrators to manage hosts and inventories. You can also update inventories from external cloud services providers like AWS, IBM Cloud, Google Cloud, Azure or local virtualization environments or content in the GIT repo.

Job scheduling and task management: Playbook execution can be scheduled either one at a time or on a recurring basis. This allows the routine tasks to be executed unattended and is especially useful for tasks like backup tasks, which are run off hours.

Real-time and historical job reporting: On the automation controller web user interface, you can check the status of the current executed job and also you can go back and check historical data for previous jobs executed in the past.

User-triggered automation: Ansible simplifies IT automation by enabling self-service by using RBAC and web UI. Day-to-day routine tasks can be easily automated or can be handled by less privileged users itself.

Credential management: Automation controller centrally manages authentication credentials. This means that you can run Ansible plays on managed hosts, synchronize information from dynamic inventory sources, and import Ansible project content from version control systems. Automation controller encrypts the passwords or keys provided so that automation controller users cannot retrieve them. Users can be granted the ability to use or replace these credentials without exposing the secrets in the current credentials to the user.

Centralized logging and audit: Automation controller logs all Ansible Playbook and remote command execution logs. This provides the ability to audit the job executed and who executed it.

Integrated notifications: Ansible automation controller can deliver notifications to email, Grafana, Slack, RocketChat, Twilio, WebHooks and so on.

All these features we discussed above will be used in this chapter in later sections.

Reference architecture of AAP

In this section, we will discuss and see how AAP can be used for providing GitOps functionality on next generation platforms like Kubernetes or OpenShift clusters. This is an advance topic and it will be good to have an understanding of some basic components and functionalities which we will be discussing before we discuss reference architecture:

GitOps: It is a branch of DevOps that focuses on Git repositories and how they can be used to manage infrastructure and platform code deployments. Here Git is the only source of truth. We will leverage GitOps to provide end-to-end automation of **Infrastructure as a**

Code (**IaaC**) where as soon as platform code is checked into the Git repo, it will be deployed automatically to target K8s or OpenShift cluster.

Red Hat Advance Cluster Management (RHACM): If you are running Kubernetes or OpenShift clusters in different environments or different clouds, then it is very hard for administrators or site reliability engineers to manage these clusters and configurations running on these clusters. This is where RHACM comes in and helps administrators and site reliability engineers by providing them full visibility of clusters across different clouds with a single view and the ability to manage configurations running on these clusters easily from a single web UI. In RHACM, we have two types of clusters – hub and managed and we are discussing them as follows:

HUB cluster: This is an OpenShift cluster which acts as a central server to manage the configurations, days 2 ops code, and upgrades on managed clusters. It also runs the webUI and APIs of RHACM. Also, we can run observability on the hub cluster to collect the metrics from all clusters managed by HUB. Normally, we do not run any business application on the hub cluster.

Managed clusters: These are OpenShift clusters which run real business applications. Configurations, upgrades and so on of managed clusters are managed by the hub cluster it is attached to. So, for example if you have to upgrade the managed cluster, it can be easily initiated from hub cluster API or webUi.

Channel: Channel is **custom resource definition** (**CRD**) in RHACM which defines what source code repositories hub is pointing to.

Example: We can have channels for Git repositories, Helm chart repositories and so on.

Subscription: Subscription is CRD within RHACM that identifies the Kubernetes resources within the channel and then places the Kubernetes resources to target the managed cluster.

Placement rule: A rule that describes or defines the target clusters where a subscription has to be delivered.

Reference architecture is shown in *Figure 10.4*:

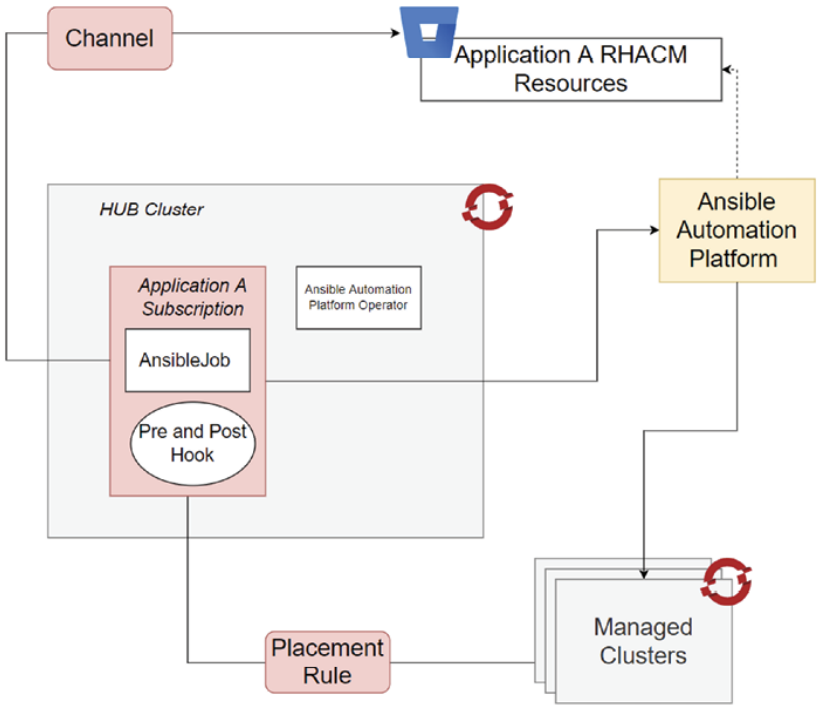

Figure 10.4: *AAP reference architecture when used for GitOps in OpenShift clusters*

Now, let us discuss the reference architecture in *Figure 10.4*.

We have the following components as a part of this architecture:

- **HUB cluster**: OpenShift 4 cluster with AAP and RHACM operators.
- **Managed clusters**: OpenShift 4 clusters which are actual target clusters where the desired code is targeted by hub.
- **Ansible Automation Platform**: This can be outside or within the OpenShift cluster.
- **Source code repo**: Bitbucket or any Git-based repo. This is defined as the RHACM channel.

The following are architectural and integration details:

- We are using AAP which is outside OpenShift cluster. Ideally you can have AAP on the same OpenShift cluster, or different OpenShift cluster, or even VM based hardware.
- We are using Bitbucket or any Git based source code repo. This repo will contain all our Ansible Playbooks, OpenShift configurations and so on.
- We have created a project on AAP which maps to the source code repo. We will discuss this in the chapter on how we setup and configure this project.
- We have OpenShift HUB cluster setup where we have **Advanced Cluster Management** (**ACM**) installed and configured. The following are ACM-related objects used in this architecture:
  - We have a channel which maps ACM with the source code repo. If any code update is made and is committed to the repo, then it is automatically deployed to target managed clusters. In other words, we have GitOps enabled.
  - We are using the placement rule which maps the ACM application to target desired managed clusters.

- We have the Ansible Automation Platform operator deployed on the hub cluster. This operator allows us to create the following objects on the hub cluster:
  - **`AnsibleJob`** objects, which is nothing but a template we can use to call the required Ansible Playbook on AAP.
  - Pre and post hooks, which are **`AnsibleJob`** objects and are called before or after certain conditions or if the criteria have been met respectively. We will discuss this further in more detail in this chapter.

Installing AAP in OpenShift cluster

In this section, we will see how AAP can be installed easily using the Red Hat operator on OpenShift clusters. Red Hat OpenShift operators automate the creation, configuration, and management of instances of Kubernetes-native applications. Operators provide automation at every level of the stack—from managing the parts that make up the platform all the way to applications that are provided as a managed service.

Red Hat OpenShift uses the power of operators to run the entire platform in an autonomous fashion while exposing configuration natively through Kubernetes objects, allowing for quick installation and frequent, robust updates. In addition to the automation advantages of operators for managing the platform, Red Hat OpenShift makes it easier to find, install, and manage operators running on your clusters.

Installing operator using command line interface.

AAP operator can be installed using a command line interface or via OpenShift web console. Following are the steps for CLI where we are creating:

1. You can save the content of the following **`yaml`** file in file **`aap-operator.yaml`** or any name and run **`oc apply -f aap-operator.yaml`**. This file will create below CRDs on the OpenShift cluster:
   - Namespace called **`ansible-automation-platform`**
   - Operator group
   - Subscription
   - AAP controller

```
---
apiVersion: v1
kind: Namespace
metadata:
  labels:
    openshift.io/cluster-monitoring: "true"
  name: ansible-automation-platform


---
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
```

```
metadata:
  name: ansible-automation-platform
  namespace: ansible-automation-platform
spec:
  targetNamespaces:
  - ansible-automation-platform

---
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: ansible-automation-platform-operator
  namespace: ansible-automation-platform
spec:
  channel: stable-2.2-cluster-scoped
  installPlanApproval: Automatic
  name: ansible-automation-platform-operator
  source: redhat-operators
  sourceNamespace: openshift-marketplace
---
apiVersion: automationcontroller.ansible.com/v1beta1
kind: AutomationController
metadata:
  name: example
  namespace: ansible-automation-platform
spec:
  create_preload_data: true
  route_tls_termination_mechanism: Edge
  garbage_collect_secrets: false
  loadbalancer_port: 80
  image_pull_policy: IfNotPresent
  projects_storage_size: 8Gi
  task_privileged: false
  projects_storage_access_mode: ReadWriteMany
```

```
  projects_persistence: false
  replicas: 1
  admin_user: admin
  loadbalancer_protocol: http
  nodeport_port: 30080
```

Following is an explanation of the above `yaml` file:

- **Namespace definition**:
  - A Kubernetes `namespace` named `ansible-automation-platform` is being created.
  - It has a label that indicates this `namespace` is enabled for cluster monitoring by OpenShift.
- **OperatorGroup definition**:
  - An `OperatorGroup` named `ansible-automation-platform` is defined within the `ansible-automation-platform` namespace.
  - This `OperatorGroup` specifies that the operators deployed in it will only manage the resources within the `ansible-automation-platform` namespace.
- **Subscription definition**:
  - A subscription for the `ansible-automation-platform-operator` is being set up in the `ansible-automation-platform` namespace.
  - This subscription is set to automatically install updates from the `stable-2.2-cluster-scoped` channel.
  - The operator is sourced from `redhat-operators` available in the `openshift-marketplace` namespace.
- **AutomationController definition**:
  - An instance of the `AutomationController` (from the Ansible Automation Platform) is being created with the name `example` within the `ansible-automation-platform` namespace.
  - Key configurations for this controller include.
  - Enabling preload data.
  - Setting up TLS termination with the `Edge` mechanism.
  - Configurations related to load balancing, project storage size, access modes, and more.
  - The controller will run a single replica with administrative privileges given to the user named `admin`.

This YAML essentially lays the groundwork for deploying Ansible Automation Platform components in a dedicated namespace within an OpenShift cluster.

2. Once the operator is installed you can verify the `pods` by running `oc get pods -n anisble-automation-platform` and you should see results like:

```
[redhat@choplnlvdev4 ~]$ oc get pods
NAME                                                     READY     STATUS    RESTARTS   AGE
automation-controller-operator-controller-manager-54c865f9m9hm   2/2       Running   1 (3d11h ago)
automation-hub-operator-controller-manager-565cf5d98-tqjnv       2/2       Running   1 (3d11h ago)
example-5bcf6bbdfd-wvmxb                                 4/4       Running   2 (9d ago)     9
example-postgres-13-0                                    1/1       Running   0
resource-operator-controller-manager-8556ff9b8b-4p2pr    2/2       Running   2 (3d11h ago)
[redhat@choplnlvdev4 ~]$
```

3. You can also verify the installed operator on the OpenShift console by going to **Operators** | **Installed Operator** and you will see results like shown in *Figure 10.5*:

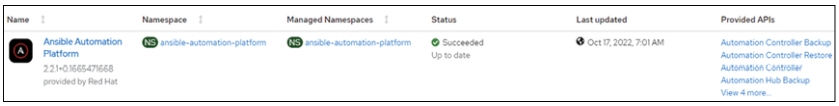

Figure 10.5: *AAP operator status on OpenShift cluster console*

4. Now, once you have the AAP operator installed and `AutomationController` CRD created, you should be able to access the user interface for AAP. For that, first, you will need to fetch the route.

   Run `oc get routes -n ansible-automation-platform` to get the route. Go to that URL in the browser and you will be asked for a basic authentication challenge.

5. In order to login into the AAP console with the `admin` user, you will need to fetch the password for automation controller. Use the following `oc` command to fetch the password. Run `oc get secret/example-admin-password -o yaml`.

   Note: Here, 'example' is the name of the controller we used in the yaml file while creating the AutomationController.

   Once you are logged into the AAP console, you will need to activate the subscription. This actually depends on what kind of license you or your company have with Red Hat. For training, you can request 60 days trial subscriptions here **https://access.redhat.com/management/subscription_allocations**.

   You will need to have an account on **access.redhat.com**. Upon successful generation and download of a subscription manifest, the subsequent steps illustrated in *Figures 10.8* and *10.9* may be followed for import.

   Figure 10.6 displays the **Ansible Automation Platform** console, a platform where the downloaded subscription may be browsed:

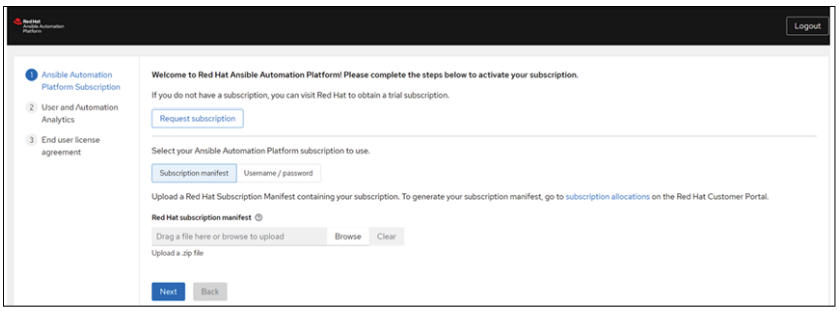

Figure 10.6: *Automation controller user interface asking for AAP subscription*

Figure 10.7, on the other hand, demonstrates the subsequent step: Once the **subscription** is selected, clicking **Next** enables the subscription:

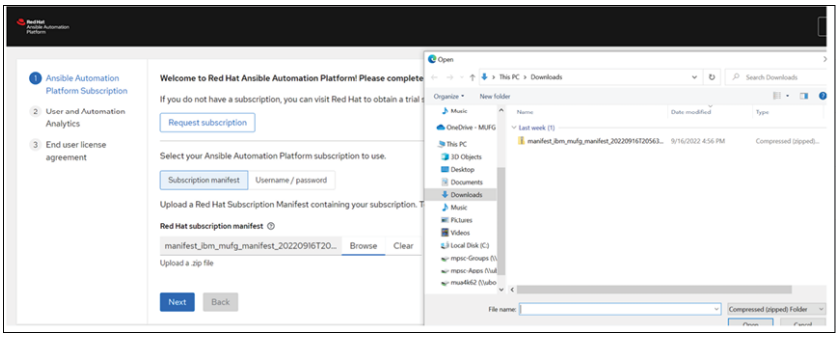

Figure 10.7: *Downloaded AAP subscription on the local machine*

Figure 10.8 demonstrates the requirement of entering Red Hat credentials:

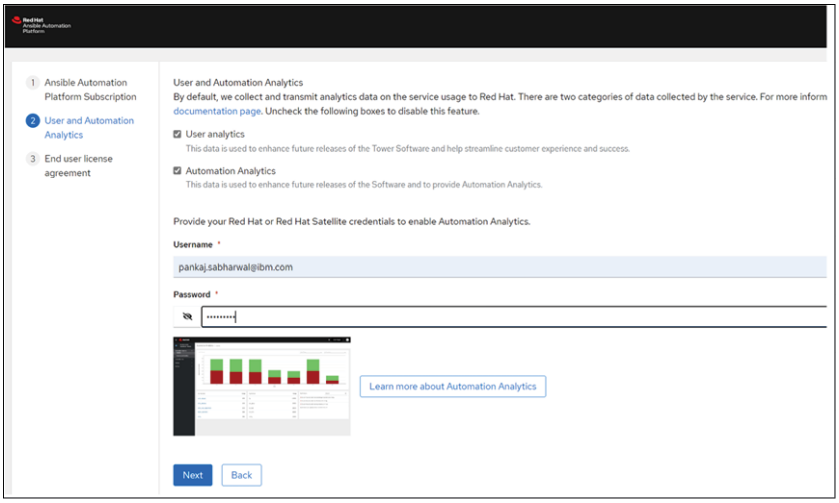

Figure 10.8: *Automation controller asking for Red Hat account user and password*

6. *Figure 10.9* shows that you are accepting the Red Hat agreement for using AAP. Once you click **Submit**, you will be taken to the **Dashboard** as shown in *Figure 10.9*. Now your setup for AAP is complete and you can make it work as per your requirements:

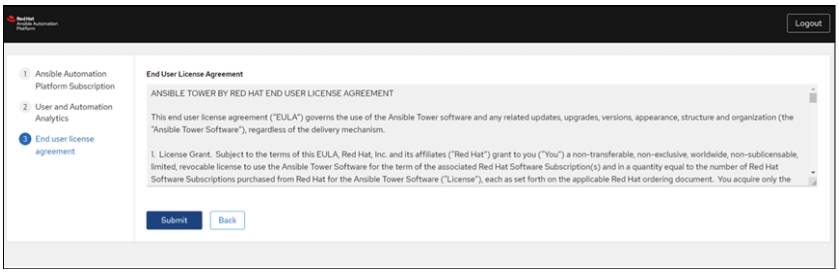

Figure 10.9: *Ansible automation control user interface for submitting the request for AAP install after providing a subscription*

Figure 10.10 as follows, demonstrates that your subscription has been enabled and you are ready to start with AAP:

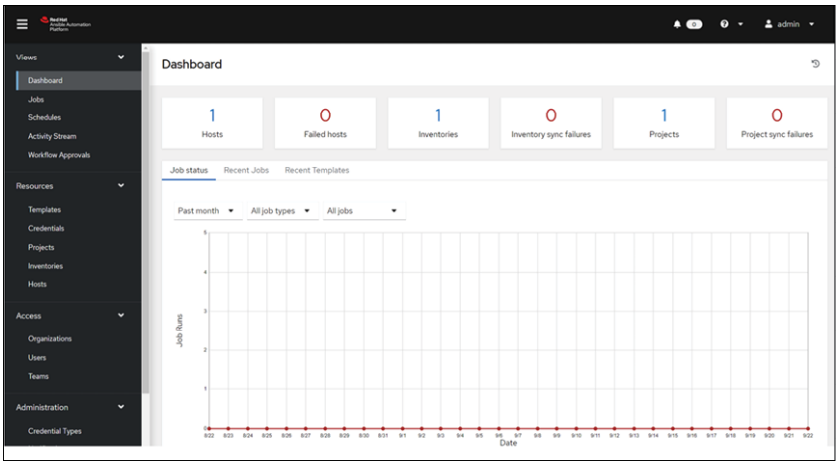

Figure 10.10: *Automation control dashboard after the subscription is applied*

Integrating AAP with source code management Git repo

In this section, we will discuss why and how we will integrate AAP with the source code repo. Here, we will use Bitbucket as a source code repo example.

This is the integration you need in order to allow AAP to use your code, like Ansible playbooks, in Bitbucket. Follow the steps in order to set this up:

1. Generate the **Personal access token** for AAP from Bitbucket as shown in *Figure 10.11*. Make a note of this token and store it carefully as it would not be available after this. You can name the token as your name; in this case we are naming it as **`pankaj-acm`** so that we can recognize the token easily.

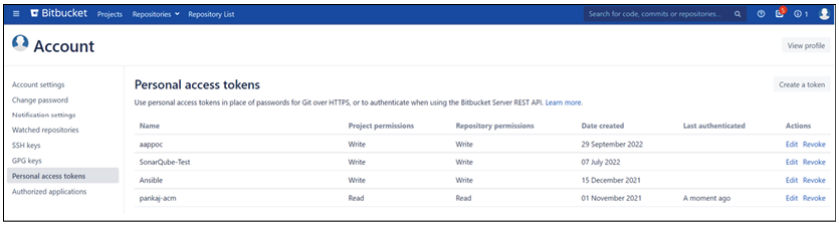

Figure 10.11: *Access token generation on Git account*

2. On the AAP side, you create a credential by logging into the **AAP console** and creating **Credentials,** which is an available option under **Resources** on left hand side. This is shown in *Figure 10.12*. Carefully look at the settings as shown. Here, we will be naming it as **scm** which you can name as per your need. On **Credential Type** we will select **scm** and then under **Username** we will type my Bitbucket user and password, we will type the **Personal access token** we created in one previous step.

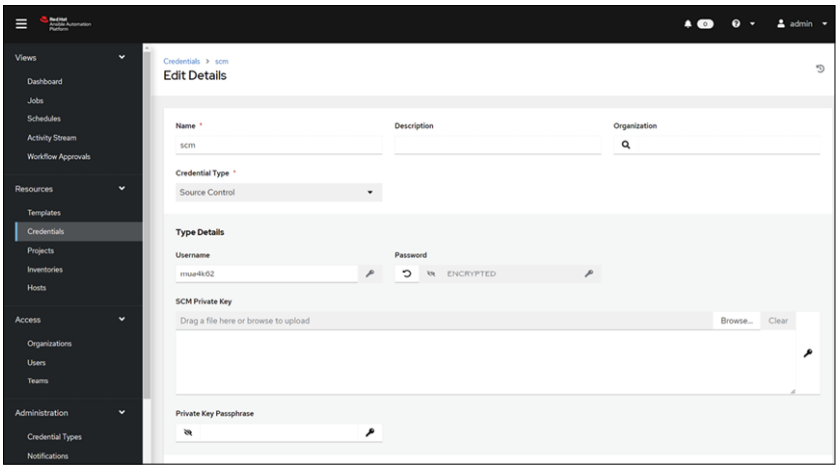

Figure 10.12: *Credential creation on automation control user interface*

3. Create a project of AAP for the Bitbucket. This is how you can sync the source code on Bitbucket on the AAP side. You have credentials ready for Bitbucket which you created in the previous step hence AAP can authentication to Bitbucket easily. Please check the settings in *Figure 10.13* where we specified the URL of the Bitbucket repo, branch name and credentials. After adding these settings click **Save**.

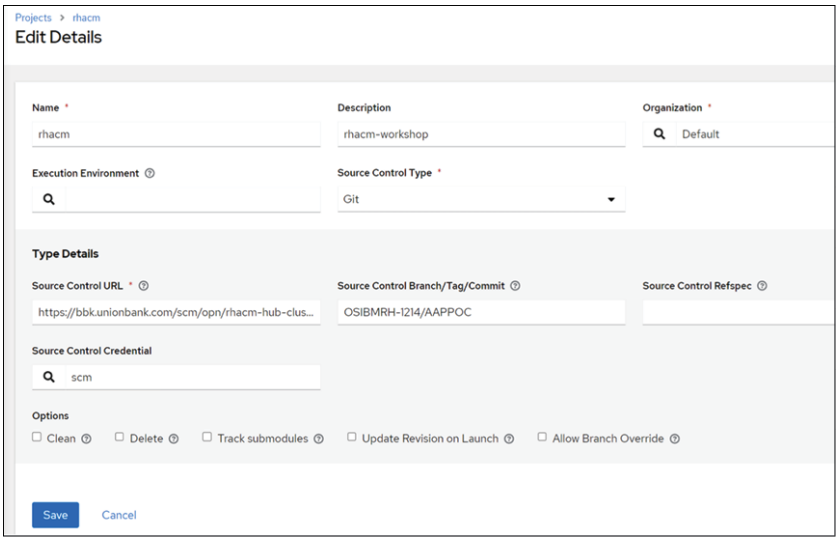

Figure 10.13: *Project creation on automation control dashboard*

4. If everything in the previous steps goes as expected then you should be able to sync the Bitbucket project easily on the AAP side. As shown in *Figure 10.14*, we see the project **rhacm** is successfully synced with the repo on Bitbucket.

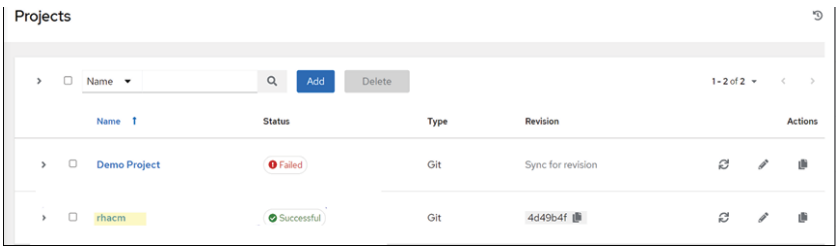

Figure 10.14: *Automation control dashboard showing AAP project in sync with Git repo*

Now you can say that AAP can access the Bitbucket successfully. Let us move to the next section, where we will discuss integrating with Red Hat Advance Cluster Management.

Example workflow

Let us consider a scenario where a team is working on automating the deployment of an application. They use Git for version control and Ansible Automation Platform for automation.

Code commit:

- A team member develops an Ansible playbook for the deployment and commits the code to a Git repository.
- The playbook contains tasks to configure servers, deploy the application, and validate the deployment.

Sync with AAP:

- The Ansible Automation Platform is configured to sync with the specific Git repository.
- Once a commit is detected, AAP automatically syncs the latest playbooks from the Git repository.
- This ensures that the automation platform is always using the most recent and accurate automation code.

Run job:

- After the sync, a job is automatically triggered in the Ansible Automation Platform using the latest playbook.
- The job executes the tasks defined in the playbook, deploying the application as specified.
- The user can monitor the progress of the job, review logs, and verify the deployment, all from within the AAP interface.

Validation:

- Upon job completion, the team validates the deployment by reviewing the logs in AAP or directly accessing the application.
- Any discrepancies or issues found can be corrected by updating the playbook and committing the changes to Git, which triggers the workflow again.

This integration with Git enables seamless synchronization and execution of automation tasks in the Ansible Automation Platform, ensuring consistent deployments and efficient workflow management. The team can focus on developing automation code, with the confidence that it will be correctly applied by the automation platform upon each commit, fostering a streamlined, automated, and error-free deployment process.

Integrating AAP with RHACM

Red Hat Advance Cluster Management is the GitOps solution from Red Hat where you manage multiple target Red Hat OpenShift clusters using subscriptions. Basically, if you want to deploy or manage any configuration on multiple OpenShift clusters, you can do that via RHACM by adding each OpenShift cluster as a managed cluster on RHACM. Now, there might be a case where you want to run a certain task after or before the code is deployed on the OpenShift cluster. In cases like this we can make use of **pre-hook** and **post-hook** options. These hooks are nothing but `AnsibleJob` CRD, which can call the Ansible playbook from the source code repo. A very simple example is deploying or ensuring that a

database is available when you deploy a Java application. In this case, when code for RHACM subscription for Java is checked into the Bitbucket, it will call the configured pre-hook which is nothing but an `AnsibleJob` object which calls Ansible Playbook to check if the database is deployed and if it is not deployed then the playbook will deploy it. Once the pre-hook step is completed, the RHACM subscription will be deployed to the target cluster.

For all this to work, we need to integrate AAP with RHACM. For this, complete the mentioned steps:

1. Create credentials on the AAP side for AAP. For this, you go to the **Credentials** section on AAP and add a username. Here, we added the username **admin** and provided roles **Admin** and **System Administrator,** as shown in *Figure 10.15*. But you can select the **Username** and **Roles** as per your need. Take note of the token you generated, as that will be needed in the next step:

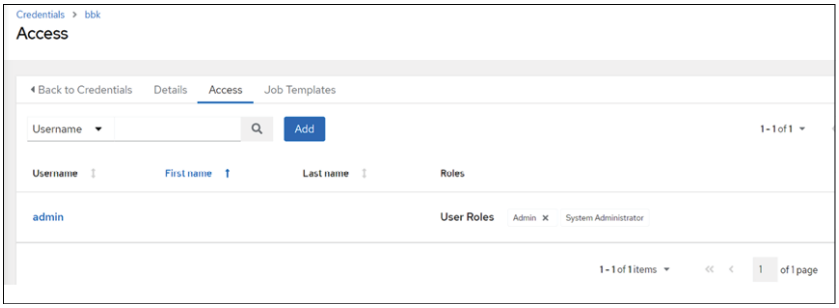

Figure 10.15: *Ansible automation control UI showing user creation and role mapping*

2. Now you have a username created on AAP in the previous step. Next you go to **Credential** on RHACM and click **Add credential**. Here you select the **Credential type** as **Red Hat Ansible Automation Platform**. Give credentials to any name like **aappoc** here and specify the **Namespace**. In the next screen, you can provide the AAP URL. This is shown in *Figure 10.16*:

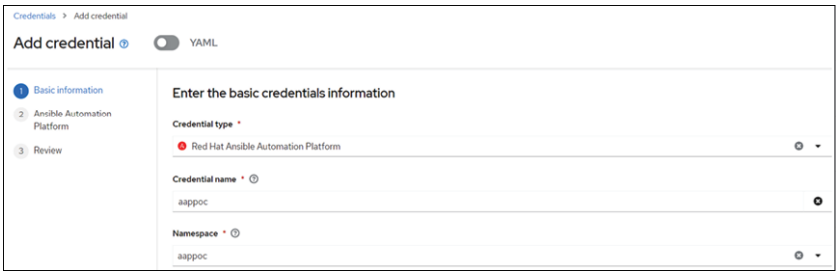

Figure 10.16: *Ansible automation control UI showing credential creation*

Now after completing the above, you have AAP which is configured with RHACM. Basically, now you can manage your OpenShift/K8s clusters easily by adding them on RHACM. Setting AAP with RHACM here provides you with complete end-to-end GitOps solutions where you do not have to run anything manually.

Use cases

Following are the two important use cases where we can use AAP for managing OpenShift/K8s clusters:

End to end GitOps solutions

In this use case we are managing RHACM applications or subscriptions which need some kind of Ansible task to be triggered before the application is deployed or after the application code is deployed. There are some automations today in our code which are in Ansible role form but need manual help to trigger, once the code is checked into the **Bitbucket** (**BBK**). What we can do is remove the manual step here to ansible trigger and let the trigger happen automatically once the code is checked in to BBK. Example: There are many updates we are doing on Advanced Cluster Security aka ACS using the Ansible role for adding RBAC on ACS. To deploy RBAC rules we have to run the ACS role manually. But if we have a subscription for ACS on RHACM and this subscription has post-hook `AnsibleJob` to run the ACS ansible role then this role will be triggered automatically once we commit the code in BBK.

The other example can be configuring custom certificates on OpenShift clusters after the OpenShift cluster is deployed. Normally what happens is that OpenShift clusters are deployed with self-signed certificates after which using Ansible playbook or manual steps, the custom certificates are configured on OpenShift clusters. With AAP, we can implement full end-to-end GitOps, which can trigger the `AnsibleJob` for requesting the custom certs and deploying the custom certs after OpenShift clusters are provisioned.

This will be more clear if you look at in workflow format, as follows:

Scenario 1

Automating RBAC Updates in Advanced Cluster Security

Commit to Bitbucket:

- The required updates or additions to RBAC rules in Advanced Cluster Security are made and committed to the `Bitbucket` repository.

RHACM subscription trigger:

- The commit in Bitbucket triggers a pre-defined subscription in RHACM, associated with a post-hook Ansible job configured in AAP to run the corresponding Ansible role for **Advanced Cluster Security** (**ACS**).

Ansible automation execution:

- AAP automatically executes the assigned Ansible role, applying the committed RBAC rules to ACS without any manual intervention.

Validation and monitoring:

- The applied changes are verified, and the status of ACS is continually monitored to ensure adherence to the newly applied RBAC rules.

Let us take another scenario here:

Scenario 2

Automated custom certificate configuration on OpenShift clusters

The workflow is addressing the automation of custom certificate configuration on deployed OpenShift clusters using a combination of Bitbucket, Red Hat Advanced Cluster Management, and Ansible Automation Platform.

OpenShift cluster deployment:

- OpenShift clusters are initially deployed with self-signed certificates.

Commit to Bitbucket:

- A commit is made in Bitbucket, representing the necessity to configure custom certificates on the deployed OpenShift clusters.

RHACM subscription and Ansible job trigger:

- The commit triggers a specific RHACM subscription which has a post-hook Ansible job in AAP for requesting and deploying the custom certificates.

Request and deployment of custom certificates:

- The Ansible job is executed automatically, requesting the required custom certificates and configuring them on the OpenShift clusters without any manual steps.

Validation and assurance:

- The configuration of custom certificates is verified, and the clusters are monitored to ensure the secure and proper functioning of the services with the newly applied certificates.

By leveraging Bitbucket, RHACM, and Ansible Automation Platform in Tandem, the manual steps traditionally involved in executing Ansible roles and tasks can be automated. This setup enables a more streamlined, efficient, and error-free process, reinforcing the principles of GitOps by automating responses and actions in real-time, following code commits in Bitbucket. Whether it is updating RBAC in Advanced Cluster Security or configuring custom certificates on OpenShift clusters, the integration provides an end-to-end automated approach, enhancing operational efficiency and reducing the scope of manual errors.

Policy management

We can configure RHACM with AAP to setup policy violation automations. There are multiple policies we deploy via RHACM, which makes sure all managed clusters are adhering those policies. Now if there are clusters where the policy is violated then we have to remediate the vulnerability manually. This process can be automated by creating a `PolicyAutomation` object where we map policy to `AnsibleJob`, which calls Ansible role to remediate the vulnerabilities and hence manual intervention is removed.

Let us understand this with workflow as follows:

Configuration of policies via RHACM:

- Define and deploy policies through RHACM to ensure all managed clusters adhere to these policies. These could be security policies, compliance policies, or configuration policies to maintain consistency across clusters.

Monitoring for policy violations:

- RHACM continually monitors all managed clusters to identify any violations of the configured policies.

Triggering automated remediation:

- When a policy violation is detected, a `PolicyAutomation` object is created. This object maps the violated policy to a specific `AnsibleJob` in AAP.
- The `PolicyAutomation` object automatically triggers the associated `AnsibleJob` without the need for manual intervention.

Execution of Ansible jobs:

- The `AnsibleJob` calls the corresponding Ansible role designed to remediate the specific policy violation.
- The role executes the necessary tasks to bring the non-compliant cluster back into compliance with the violated policy, addressing the identified vulnerabilities or configuration discrepancies.

Validation and logging:

- Once the remediation is complete, RHACM verifies if the cluster is now compliant with the policy.
- All actions, violations, and remediations are logged for auditability and traceability.

Continuous enforcement and monitoring:

- RHACM continues to enforce policies and monitor clusters, and AAP continues to remediate any new violations, ensuring ongoing compliance and security across all clusters.

Outcome:

By integrating RHACM with AAP and utilizing `PolicyAutomation` objects, the process of identifying and remediating policy violations becomes seamless and automated. This approach not only removes the need for manual intervention and reduces the risk of human errors but also ensures quicker response times to policy violations, maintaining the security, compliance, and consistency of the clusters effectively.

Impact of AAP in multi-cloud environments

With the prolific growth of cloud services, managing resources across various cloud service providers has become crucial. AAP's multi-cloud management capabilities ensure that organizations can efficiently handle resources, regardless of the cloud service provider, allowing seamless integration and management of resources across different cloud environments. This multi-cloud approach facilitated by AAP ensures optimal resource utilization, cost management, and enhanced operational flexibility across diverse cloud ecosystems.

DevOps and GitOps best practices

The automation capabilities of AAP align perfectly with DevOps and GitOps best practices, providing a controlled, versioned approach to manage configuration changes through Git. This leads to more structured, traceable, and efficient management of the desired state of OpenShift clusters, stored securely in a Git repository, and any modifications made are automatically reflected in the clusters.

Key benefits

Following are some key benefits:

Enhanced efficiency:

- AAP's automation and orchestration tools significantly streamline code and policy management processes, reducing manual intervention and mitigating the risk of errors.

Unparalleled automation:

- The platform empowers users with advanced automation features, ensuring seamless policy enforcement and immediate remediation across OpenShift clusters without human intervention.

Consistency and reproducibility:

- AAP ensures uniformity in operations and configurations across diverse environments. The automation of tasks enhances the reproducibility of processes and deployments, making it an indispensable tool for maintaining system stability.

Operational excellence:

- Incorporating AAP is not merely an upgrade but a substantial stride towards achieving operational superiority amidst escalating technological complexities.

Enhanced security:

- Automated policy management and enforcement ensure that all clusters are secure and comply with defined policies, reducing vulnerabilities and enhancing overall system security.

Multi-cloud management:

- AAP's comprehensive capabilities extend to managing diverse cloud environments, offering a singular, unified platform to orchestrate multi-cloud operations seamlessly.

Conclusion

This chapter has elucidated the profound capabilities of the Ansible Automation Platform for orchestrating GitOps and fostering efficient Day2Ops code management in OpenShift clusters. AAP emerges as a paragon of robustness and adaptability, poised to undertake intricate automation requisites, elevate productivity, and fortify system resilience.

In essence, the integration of AAP into an organization's strategy signifies a leap toward modern, efficient, and optimized operational methodologies. The benefits reaped from utilizing AAP transcend beyond mere functionality, impacting productivity, security, and comprehensive system optimization positively.

Here is a consolidation of the key benefits and capabilities discussed:

Streamlined workflow orchestration: AAP provides a streamlined and coherent approach to managing workflows, ensuring that all tasks are performed in a synchronized and orderly manner.

Enhanced collaboration: By integrating with Git repositories, AAP enables team members to collaborate on code development and deployment efficiently, ensuring that everyone is working with the most up-to-date versions.

Automated governance: The use of AAP with Red Hat Advanced Cluster Management for Kubernetes (RHACM) facilitates policy-driven governance, enabling consistent security and compliance across multiple clusters.

Flexibility and scalability: AAP's modular design allows for flexibility in scaling operations up or down according to the organization's evolving needs.

Cost-effective operations: With AAP's ability to automate repetitive tasks, organizations can reduce operational costs and redirect their focus towards innovation and strategic initiatives.

Simplified Day2Ops: AAP's tools and features are designed to handle the complexities of Day2 operations, such as updates, scaling, and self-healing of systems, thus minimizing downtime and maintenance overhead.

GitOps implementation: Through AAP, organizations can adopt a GitOps model, enabling them to manage their infrastructure and application configurations using Git as the single source of truth.

Policy management: AAP enables precise policy management, allowing organizations to enforce policies and stay compliant with industry standards.

In conclusion, the Ansible Automation Platform stands as a transformative force, empowering organizations to harness the full potential of automation in their pursuit of operational excellence. It lays the foundation for resilient, secure, and highly efficient infrastructure and application management across diverse environments. As we look

towards the future, AAP’s role in shaping the landscape of IT operations becomes increasingly indispensable, with its commitment to innovation and continuous improvement. In the next chapter, we will learn about Ansible for deep learning.

Join our book’s Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 11
Red Hat Ansible for Deep Learning

Introduction

Deep learning represents a profound layer of machine learning centered on the orchestration and deployment of artificial neural networks to tackle advanced tasks like image recognition, natural language processing, and speech recognition. With its capacity to surpass traditional machine learning models, deep learning has risen in prominence across diverse domains. However, setting up the necessary hardware and software for deep learning can be intricate and time-consuming.

Enter Ansible. Employing Ansible version 2.11 on Red Hat Enterprise Linux (RHEL 8.3), this chapter delves into how this groundbreaking automation tool streamlines the process. Ansible not only simplifies the installation and configuration of key deep learning frameworks like TensorFlow, PyTorch, and Keras but also automates the deployment of essential components such as CUDA, cuDNN, and NVIDIA drivers. Its adeptness in managing deep learning workflows across distributed environments makes

Ansible an invaluable ally for those in deep learning aiming to enhance their processes.

Beyond deployment, Ansible enhances reproducibility and collaboration in deep learning projects. Reproducibility is critical, as it provides a robust foundation for specialists to verify outcomes and expand upon previous work with assurance. Ansible facilitates the automation of environment setup and configuration, ensuring consistent and comprehensively documented development environments are established for team members. This approach significantly reduces the ‘it works on my machine’ syndrome, promoting uniformity in model development, testing, and deployment processes. Moreover, Ansible’s ability to automate platforms like Jupyter Notebook delivers a unified space for data analysis, model training, and the sharing of insights, further enabling collaborative progress in deep learning initiatives.

This chapter provides a deep dive into Ansible’s prowess in the deep learning landscape, highlighting its significant role in simplifying deployments, nurturing collaboration, and ensuring reproducibility. Readers will come away with valuable insights on harnessing Ansible for more streamlined and efficient workflows in their deep learning endeavors.

Structure

In this chapter, we will go over the following topics:

- How Ansible can help in the area of deep learning
- Using Ansible to install deep learning components
- Installing other deep learning components

Objectives

In this chapter, we explore how Red Hat Ansible acts as a comprehensive automation tool specifically for deploying environments catered to deep learning, focusing on how it facilitates the automation of installation, configuration, and management integral to deep learning workflows. This chapter is crafted to serve as a valuable guide for those in fields such as data science and machine learning, aiming to harness the power of Ansible for more streamlined, efficient, and productive development workflows in deep learning. By understanding and applying the capabilities of Ansible explored in this chapter, readers can accelerate project timelines, achieve higher reproducibility, and optimize the deployment of AI-based solutions.

Use of Ansible in deep learning

Ansible can help in the area of deep learning in the following ways:

- **Streamlining software installation and configuration**:

  Ansible Playbooks automate the installation and configuration of essential deep learning frameworks and components, ensuring consistency across different machines and managing various software versions effectively.

- **Managing distributed computing environments**:

  Given the extensive computing power deep learning necessitates, Ansible automates deployment and management in distributed computing environments like clusters or cloud platforms, enhancing the manageability of large-scale computing resources.

- **Enabling reproducibility**:

Ansible's automation of software environment setups and dependencies is crucial for reproducibility in deep learning research, ensuring consistent results across various environments and machines.

- **Simplifying infrastructure management**:

  Deep learning projects often demand intricate infrastructure management, involving tasks like managing virtual machines, storage, and networking, which are streamlined with Ansible's automation capabilities.

- **Facilitating model deployment to production**:

  Once models are trained and ready, deploying them into production environments is the next pivotal step, and here, Ansible shines again. It can automate the deployment process, ensuring that models are correctly installed, configured, and optimized for the production environment. This includes setting up the necessary software and hardware resources, adjusting network configurations, and managing dependencies, ensuring a smooth transition from development to production. For example, if a trained model is to be deployed on a Kubernetes cluster, Ansible can automate the entire deployment process, creating necessary pods, managing resources, and configuring network access, thus reducing manual intervention and mitigating the risk of deployment errors.

- **Further applications of Ansible in deep learning**:

  Ansible proves to be indispensable in various use cases in deep learning, automating tasks from installing dependencies and libraries, configuring network settings, setting up virtual environments, configuring IDEs and editors, creating and configuring Docker containers.

In essence, Ansible streamlines the processes involved in setting up and managing the requisite software and infrastructure for deep learning, allowing researchers and developers to concentrate more on the research and development aspects of their projects. It ensures the deep learning infrastructure remains consistent, reliable, and manageable, regardless of the scale, saving time and minimizing risks associated with errors and misconfigurations.

Following are some important use cases where Ansible has proven to be a great help in deep learning areas:

- **Installing dependencies and libraries**: Ansible can be used to automate the installation of dependencies and libraries required for deep learning frameworks such as TensorFlow, PyTorch, and Keras. This can include installing system-level packages like `CUDA`, `cuDNN`, and `NCCL`, as well as Python packages like `NumPy`, `SciPy`, and `Matplotlib`.
- **Configuring network settings**: Ansible can be used to automate the configuration of network settings required for deep learning, such as setting up SSH, configuring firewalls, and enabling remote access to the servers.
- **Setting up virtual environments**: Ansible can be used to automate the setup of virtual environments for deep learning, such as using virtualenv or Anaconda to create isolated Python environments.
- **Configuring IDEs and editors**: Ansible can be used to automate the configuration of **integrated development environments** (**IDE**s) and editors for deep learning, such as PyCharm, Visual Studio Code, or Jupyter Notebook.

- **Creating and configuring Docker containers:** Ansible can be used to automate the creation and configuration of Docker containers for deep learning, which can help simplify the deployment and management of deep learning environments.

Overall, Ansible can help streamline the process of setting up and managing the software and infrastructure required for deep learning, making it easier to focus on the actual research or application development. By using Ansible for provisioning and managing deep learning environments, you can ensure that your deep learning infrastructure is consistent, reliable, and easy to manage, regardless of the number of servers or cloud instances you need to configure. This can help save time and reduce the risk of errors and misconfigurations, allowing you to focus on the actual deep learning tasks and experiments.

The utility of Ansible in the realm of deep learning is demonstrated through its ability to:

- Streamline deployment of pivotal deep learning frameworks like TensorFlow and Keras, ensuring environments are optimized and consistent.
- Enable GPU acceleration by automating the setup of CUDA and NVIDIA GPU drivers, simplifying manual installation processes.
- Facilitate collaborative data exploration by deploying Jupyter Notebook, providing an interactive environment for various stages of deep learning projects.
- Enhance distributed model training by facilitating Horovod installation, optimizing performance and scalability across multiple GPUs and nodes.
- Integrate computer vision by automating OpenCV installation and integration, streamlining image

processing tasks.

- Manage version control in machine learning projects through the setup and management of **data version control** (**DVC**).

Using Ansible to install deep learning components

The Ansible Playbooks provided below are a set of tools designed to automate the installation and configuration of deep learning frameworks and other essential components. Deep learning involves training and using artificial neural networks to solve complex problems such as image recognition, natural language processing, and speech recognition. However, setting up the necessary software and infrastructure for deep learning can be a time-consuming and error-prone process. The Ansible Playbooks aim to simplify this process by providing a modular and flexible way to install and configure popular deep learning frameworks such as TensorFlow, PyTorch, and Keras, as well as additional components such as CUDA, cuDNN, and NVIDIA drivers. This can help to streamline the process of getting started with deep learning, making it easier for researchers and practitioners to focus on developing and training their models.

Setting up TensorFlow using Ansible

Refer to the following code:

```
---

- hosts: all

  become: yes
```

```
  tasks:
    - name: Update package cache
      apt:
update_cache: yes

    - name: Install required packages
      apt:
        name:
          - python3-dev
          - python3-pip
          - python3-venv
          - build-essential
          - libssl-dev
          - libffi-dev
        state: present

    - name: Create virtual environment
      command: python3 -m venv ~/venv/tf
args:
        creates: ~/venv/tf

    - name: Activate virtual environment
      command: source ~/venv/tf/bin/activate
```

```
    - name: Install TensorFlow
      pip:
        name: tensorflow
virtualenv: ~/venv/tf
```

This playbook does the following:

- Updates the package cache to ensure that the latest packages are available.
- Installs the required packages for TensorFlow, including **`Python 3`**, **`pip`**, **`venv`**, and build tools.
- Creates a virtual environment for TensorFlow in the **`~/venv/tf`** directory.
- Activates the virtual environment.
- Installs TensorFlow using **`pip`**.

To use this playbook, you can save it to a file (for example, **`tensorflow.yaml`**) and run it using the **`ansible-playbook`** command, specifying the target host(s) using the **`-i`** flag:

```
$ ansible-playbook -i inventory.ini tensorflow.yaml
```

Note that you will need to replace **`inventory.ini`** with your own **`inventory`** file containing the IP addresses or hostnames of the target server(s). Moreover, depending on your specific environment and requirements, you may need to customize this playbook or add additional tasks to meet your needs.

Setting up CUDA drivers

If you want to use TensorFlow with GPU acceleration, you will need to install CUDA and cuDNN separately before installing TensorFlow. Here is an example of how you can

modify the playbook to install CUDA and cuDNN on an Ubuntu-based server:

```
---
- hosts: all
  become: yes

  vars:
cuda_version: "11.5"
cudnn_version: "8.2.4.15"
tf_version: "2.6.0"

  tasks:
    - name: Update package cache
      apt:
update_cache: yes

    - name: Install required packages
      apt:
        name:
          - python3-dev
          - python3-pip
          - python3-venv
          - build-essential
```

```
          - libssl-dev
          - libffi-dev
          - wget
        state: present


    - name: Download CUDA installer
get_url:
        url:
"https://developer.download.nvidia.com/compute/cud
a/repos/ubuntu2004/x86_64/cuda-{{ cuda_version
}}_amd64.deb"
dest: "/tmp/cuda.deb"


    - name: Install CUDA
      apt:
        deb: "/tmp/cuda.deb"
        state: present


    - name: Download cuDNN archive
get_url:
        url:
"https://developer.download.nvidia.com/compute/cud
a/repos/ubuntu2004/x86_64/libcudnn8_{{
cudnn_version }}-1+cuda{{ cuda_version
}}_amd64.deb"
```

```
dest: "/tmp/cudnn.deb"

    - name: Install cuDNN
      apt:
        deb: "/tmp/cudnn.deb"
        state: present

    - name: Create virtual environment
      command: python3 -m venv ~/venv/tf
args:
        creates: ~/venv/tf

    - name: Activate virtual environment
      command: source ~/venv/tf/bin/activate

    - name: Install TensorFlow with GPU support
      pip:
        name: "tensorflow-gpu=={{ tf_version }}"
virtualenv: ~/venv/tf
```

This modified playbook does the following:

- Updates the package cache and installs the required packages for TensorFlow, as before.
- Downloads the CUDA installer package and the cuDNN archive from the official NVIDIA website.

- Installs CUDA and cuDNN using the downloaded packages.
- Creates a virtual environment for TensorFlow, as before.
- Activates the virtual environment, as before.
- Installs TensorFlow with GPU support using `pip`.

Note that the playbook assumes that you are using Ubuntu 20.04 as the target operating system. If you are using a different version of Ubuntu or a different Linux distribution, you may need to modify the URLs of the `CUDA` and `cuDNN` packages accordingly. Moreover, you may need to customize the versions of CUDA, cuDNN, and TensorFlow to match your specific requirements.

Installing NVIDIA drivers

To use TensorFlow with GPU acceleration, you will need to install the NVIDIA driver separately before installing CUDA and cuDNN. Here is an example of how you can modify the playbook to install the NVIDIA driver on an Ubuntu-based server.

NVIDIA is needed for running TensorFlow with GPU acceleration. TensorFlow is a popular open-source machine learning library that can be used for a variety of applications, including deep learning. TensorFlow supports both CPU and GPU acceleration, but using a GPU can greatly speed up the training of deep learning models, especially for large datasets.

To use TensorFlow with GPU acceleration, you need a GPU that supports CUDA, which is a parallel computing platform and programming model developed by NVIDIA. In addition, you need to install the NVIDIA driver, which is a software component that allows the operating system to communicate with the GPU hardware.

The NVIDIA driver provides the low-level interface between the operating system and the GPU, and it includes the necessary libraries and utilities for CUDA to work properly. CUDA is a software platform that enables developers to write high-performance GPU-accelerated applications using popular programming languages such as C, C++, and Python. TensorFlow uses CUDA to communicate with the GPU and to offload the computationally intensive tasks to the GPU.

In summary, NVIDIA is needed for running TensorFlow with GPU acceleration because it provides the necessary hardware and software components, including the GPU hardware, the CUDA software platform, and the NVIDIA driver to enable TensorFlow to run on the GPU.

Refer to the following code:

```
---
- hosts: all
  become: yes


  vars:
cuda_version: "11.5"
cudnn_version: "8.2.4.15"
tf_version: "2.6.0"
nvidia_driver_version: "470"


  tasks:
    - name: Update package cache
```

```
      apt:
update_cache: yes

    - name: Install required packages
      apt:
        name:
          - python3-dev
          - python3-pip
          - python3-venv
          - build-essential
          - libssl-dev
          - libffi-dev
          - wget
        state: present

    - name: Add NVIDIA package repository
apt_repository:
        repo: "deb
http://us.download.nvidia.com/tesla/{{
nvidia_driver_version }}/pool/main/n/nvidia-
driver/ nvidia-driver-{{ nvidia_driver_version }}"
        state: present
        filename: "nvidia-driver-{{
nvidia_driver_version }}.list"
```

```
    - name: Install NVIDIA driver
      apt:
        name: nvidia-driver
        state: present

    - name: Download CUDA installer
get_url:
        url:
"https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2004/x86_64/cuda-{{ cuda_version }}_amd64.deb"
dest: "/tmp/cuda.deb"

    - name: Install CUDA
      apt:
        deb: "/tmp/cuda.deb"
        state: present

    - name: Download cuDNN archive
get_url:
        url:
"https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2004/x86_64/libcudnn8_{{
```

```
cudnn_version }}-1+cuda{{ cuda_version
}}_amd64.deb"
dest: "/tmp/cudnn.deb"


    - name: Install cuDNN
      apt:
        deb: "/tmp/cudnn.deb"
        state: present


    - name: Create virtual environment
      command: python3 -m venv ~/venv/tf
args:
        creates: ~/venv/tf


    - name: Activate virtual environment
      command: source ~/venv/tf/bin/activate


    - name: Install TensorFlow with GPU support
      pip:
        name: "tensorflow-gpu=={{ tf_version }}"
virtualenv: ~/venv/tf
```

This modified playbook does the following:

- Updates the package cache and installs the required packages for TensorFlow, as before.

- Adds the official `NVIDIA package repository` to the system.
- Installs the NVIDIA driver using `apt`.
- Downloads the CUDA installer package and the cuDNN archive from the official NVIDIA website, as before.
- Installs CUDA and cuDNN using the downloaded packages, as before.
- Creates a virtual environment for TensorFlow, as before.
- Activates the virtual environment, as before.
- Installs TensorFlow with GPU support using `pip`, as before.

Note that the playbook assumes that you are using Ubuntu 20.04 as the target operating system, and the NVIDIA driver version is set to 470. If you are using a different version of Ubuntu or a different NVIDIA driver version, you may need to modify the repository URL and the driver package name accordingly.

Installing Jupyter Notebook

Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations, and narrative text. Jupyter Notebook supports over 40 programming languages, including Python, R, Julia, and MATLAB.

Jupyter Notebook is commonly used in the fields of data science, machine learning, and scientific computing. Here are some use cases of Jupyter Notebook:

- **Data exploration and analysis:** Jupyter Notebook provides an interactive environment for exploring and

analyzing data, allowing data scientists to quickly iterate over their analysis and visualization code.

- **Machine learning and deep learning:** Jupyter Notebook is often used to develop and train machine learning and deep learning models, allowing data scientists to experiment with different algorithms, hyperparameters, and data preprocessing steps.
- **Education and research:** Jupyter Notebook is widely used in education and research, allowing students and researchers to share their code and results in a reproducible and interactive way.
- **Data visualization:** Jupyter Notebook provides built-in support for creating interactive visualizations, allowing data scientists to explore and communicate their data insights more effectively.

Refer to the following code:

```
- name: Install Jupyter Notebook
  hosts: all
  become: yes

  tasks:
    - name: Install pip
      apt:
        name: python3-pip
        state: present

    - name: Install Jupyter Notebook
```

```
      pip:
        name: jupyter
        state: present


    - name: Generate Jupyter Notebook
configuration file
      command: jupyter notebook --generate-config


    - name: Check if password is already set
      command: "grep -q 'c.NotebookApp.password'
/home/{{ ansible_user }}/notebook_config.py"
      register: password_check
      failed_when: false
      changed_when: false


    - name: Set Jupyter Notebook password
      command: "echo 'c.NotebookApp.password = \"
{{ jupyter_password | password_hash('sha256')
}}\"' >> /home/{{ ansible_user
}}/notebook_config.py"
      when: password_check.rc != 0


    - name: Start Jupyter Notebook service
      command: "jupyter notebook --no-browser --
ip='0.0.0.0' --port=8888 --config=/home/{{
```

```
ansible_user }}/notebook_config.py"
      async: 30
      poll: 0
```

Here is a brief breakdown of what the playbook is doing:

1. **Installing pip**: The playbook starts by installing the `python3-pip` package using the `apt` module. This ensures that pip (the Python package manager) is available on the target system.
2. **Installing Jupyter Notebook**: With pip installed, the playbook proceeds to install Jupyter Notebook using the `pip` module.
3. **Generating a Jupyter Notebook configuration file**: The playbook generates a default Jupyter Notebook configuration file by running the `jupyter notebook --generate-config` command. This file is generally required for customized settings.
4. **Setting the Jupyter Notebook password**: The playbook sets a password for the Jupyter Notebook by appending the hashed password configuration to the previously generated configuration file. This ensures that users need a password to access the Jupyter Notebook when it runs.
5. **Starting the Jupyter Notebook service**: Finally, the playbook starts the Jupyter Notebook service with specified settings, including no browser option, binding to all IP addresses (`0.0.0.0`), using port `8888`, and referring to the generated configuration file.

This playbook essentially automates the process of setting up and launching a Jupyter Notebook instance on the target host(s).

You can run this playbook using the command **`ansible-playbook jupyter.yaml`**.

Managing ML experiments with Ansible, DVC, MLflow:

In conjunction with tools like DVC and MLflow, Ansible can be leveraged to manage and track ML experiments, ensuring that the ML lifecycle is more reproducible, manageable, and transparent.

Example:

Using Ansible, ML practitioners can automate the deployment of DVC and MLflow, manage their configurations, and subsequently, automate experiment tracking processes:

```
- name: Install DVC and MLflow
  hosts: all
  become: yes
  tasks:
    - name: Install DVC and MLflow
      pip:
        name:
          - dvc
          - mlflow
        state: present
```

This playbook automates the installation of DVC and MLflow, allowing data scientists to version their data and models and track experiments seamlessly.

Automating hyperparameter tuning and model training using Ansible

In the context of deep learning, the precision and efficiency of model training and hyperparameter tuning are crucial. Ansible can automate these critical aspects of the deep learning pipeline, reducing the time and effort required to find the optimal model parameters and accelerate the training process.

Integration with HyperOpt/Ray Tune

Hyperparameter tuning tools like HyperOpt and Ray Tune are pivotal in refining model parameters to enhance performance. Integrating Ansible with these tools can streamline the process of hyperparameter optimization and model training. Here is a brief example of how such an integration could look:

1. **Automating HyperOpt with Ansible:**

   Ansible can be utilized to deploy and configure environments where HyperOpt runs, ensuring the necessary dependencies and packages are installed and set up correctly, and run HyperOpt tuning jobs on configured environments automatically.

```
- name: Setup Hyperopt Environment
 hosts: hyperopt_nodes
  tasks:
    - name: Install Necessary Packages
      apt:
        name: ["python3", "python3-pip"]
        state: present
```

```
    - name: Install Hyperopt
      pip:
        name: hyperopt
```

This Ansible Playbook sets up HyperOpt on the designated nodes, ensuring that the environment is ready for running hyperparameter tuning jobs.

2. **Leveraging Ray Tune with Ansible:**

   Similar to HyperOpt, Ansible can automate the deployment and configuration of Ray Tune in the desired environments, manage dependencies, and initiate Ray Tune jobs for model training and hyperparameter tuning.

```
- name: Setup Ray Tune Environment
  hosts: ray_tune_nodes
  tasks:
    - name: Install Necessary Packages
      apt:
        name: ["python3", "python3-pip"]
        state: present

    - name: Install Ray Tune
      pip:
        name: ray[tune]
```

This playbook ensures that Ray Tune is installed and configured correctly on the specified nodes.

Ansible's versatility extends to automating the entire model training workflow, ensuring consistent, repeatable, and optimized training processes. For example, once the hyperparameter tuning is complete, Ansible can automate the deployment of the trained model to the training environments, manage the necessary resources, and monitor the training process.

Benefits if integration with HyperOpt/Ray Tune

The integration of Ansible with hyperparameter tuning tools like HyperOpt and Ray Tune not only simplifies the workflow but also ensures that the model training and hyperparameter tuning is executed in a controlled, optimized, and reproducible manner, thus minimizing errors and improving the overall efficiency of deep learning projects.

Installing other deep learning components

The Ansible Playbooks covered some of the common components required for deep learning, such as Python, TensorFlow, CUDA, and Jupyter Notebook. However, there are other components and tools that are often used in deep learning workflows that are not covered in those playbooks. Here are some examples:

- **PyTorch**: PyTorch is an open-source machine learning library developed by Facebook's AI research team. It provides a tensor computation library similar to NumPy and supports both CPU and GPU acceleration. You can install PyTorch using `pip` or `conda`.
- **Keras**: Keras is a high-level neural networks API written in Python. It provides a simple and consistent interface for building and training neural networks, and supports both TensorFlow and Theano as the backend. You can install Keras using `pip` or `conda`.

- **Horovod**: Horovod is a distributed deep learning framework developed by Uber. It supports TensorFlow, PyTorch, and Apache MXNet, and allows you to scale your deep learning training across multiple GPUs and machines. You can install Horovod using `pip`.
- **OpenCV**: OpenCV is an open-source computer vision library that provides tools for image and video processing. It is often used in deep learning for tasks such as data preprocessing, image augmentation, and visualization. You can install OpenCV using `pip` or `conda`.
- **DVC**: **Data version control** (**DVC**) is an open-source version control system for data science and machine learning projects. It allows you to track changes to your data and models, collaborate with others, and reproduce experiments. You can install DVC using `pip`.

In summary, there are many components and tools that are often used in deep learning workflows. The Ansible Playbooks provided here are just a starting point. Depending on your specific needs and workflows, you may need to install additional components and tools using Ansible or other automation tools.

Let us see how these components can be covered by Ansible.

Installing PyTorch

Refer to the following code:

```
- name: Install PyTorch
  hosts: all
  become: yes
```

```
  tasks:
    - name: Install dependencies
      apt:
        name: libomp-dev
        state: present

    - name: Install PyTorch
      pip:
        name: torch
        state: present
```

This playbook installs the **`libomp-dev`** package, which is a dependency for PyTorch, and then installs PyTorch using **`pip`**.

Installing Keras

Refer to the following code:

```
- name: Install Keras
  hosts: all
  become: yes

  tasks:
    - name: Install Keras
      pip:
        name: keras
        state: present
```

This playbook installs Keras using **pip**.

Installing Horovod

Refer to the following code:

```
- name: Install Horovod
  hosts: all
  become: yes

  tasks:
    - name: Install dependencies
      apt:
        name: build-essential
        state: present

    - name: Install Horovod
      pip:
        name: horovod
        state: present
```

This playbook installs the **build-essential** package, which is a dependency for Horovod, and then installs Horovod using **pip**.

Installing OpenCV

Refer to the following code:

```
- name: Install OpenCV
```

```
  hosts: all
  become: yes

  tasks:
    - name: Install dependencies
      apt:
        name: libsm6 libxext6 libxrender-dev
        state: present

    - name: Install OpenCV
      pip:
        name: opencv-python-headless
        state: present
```

This playbook installs the **libsm6**, **libxext6**, and **libxrender-dev** packages, which are dependencies for OpenCV, and then installs OpenCV using **pip**.

Installing DVC

Refer to the following code:

```
- name: Install DVC
  hosts: all
  become: yes

  tasks:
```

```
  - name: Install dependencies
    apt:
      name: git
      state: present

  - name: Install DVC
    pip:
      name: dvc
      state: present
```

This playbook installs the `git` package, which is a dependency for DVC, and then installs DVC using `pip`.

Note that these playbooks assume that you are using a Ubuntu-based machine. You may need to modify the playbooks for other Linux distributions. Additionally, these playbooks only install the dependencies and packages required for each component and do not configure or tune the components. You may need to perform additional configuration or tuning depending on your use case and requirements.

Conclusion

Deep learning frameworks such as TensorFlow, PyTorch, and Keras require complex software environments, including system libraries, drivers, and other dependencies, that can be challenging to set up and configure manually. Ansible provides a way to automate this process, allowing users to define a set of tasks or playbooks that can be executed on one or more machines. By using Ansible to automate the installation and configuration of deep learning frameworks and other essential components, researchers and

practitioners can save time and reduce the risk of errors or inconsistencies.

Moreover, Ansible Playbooks are modular and customizable, meaning that users can adapt them to their specific needs. For example, users can choose which versions of a framework to install or whether to include additional components such as TensorBoard or Jupyter. This flexibility allows users to set up their deep learning environment in a way that best suits their research or development needs. Another benefit of using Ansible is the ability to easily reproduce software environments across multiple machines or environments. By using the same set of Ansible Playbooks to set up deep learning environments on multiple machines, users can ensure consistency and reproducibility in their research or development work. This can be particularly useful when working on large-scale projects or collaborating with others.

Overall, Ansible is a powerful tool for anyone working in the field of deep learning, providing a way to automate and standardize the installation and configuration of deep learning frameworks and other essential components. By removing the manual effort required to set up a deep learning environment, Ansible can help researchers and practitioners focus on what they do best, that is, developing and training deep learning models to solve complex problems.

Join our book’s Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

V

W