Automating the Provisioning of Kubernetes Cluster on Bare Metal Servers in Public Cloud

BARE METAL CLOUD

By Seow Lim, Sergio Muriana

April, 2021

With the continued evolution of Public Cloud technology and platforms, more and more companies are leveraging their elasticity and flexibility to meet their compute workload requirements. While virtual machine (VM)-based cloud platforms are sufficient for many types of workloads, there are still demands for the performance and isolations provided by bare metal servers.

According to Gartner, as companies adopt Cloud Native technology, one of the top emerging trends is bare metal containers. In a bare metal containers setup, container platforms such as Kubernetes are deployed to run directly on bare metal servers instead of VMs. This helps remove the resource and performance overhead of VMs without losing the benefits of operational simplicity and security.

In the context of accelerated container adoption, which is increasingly used for production environments as suggested in **the 2019 CNCF Survey**, bare metal platform gains even more importance. The ability to leverage dedicated resources with the flexibility of the public cloud helps overcome some of the most common challenges of organizations looking to adopt agile business models and go to market faster.

Addressing the Challenges of Adopting Cloud-Native Infrastructure

Adopting the agile methodologies has become a priority for organizations of different sizes, helping improve project quality and efficiency. Some of the most common challenges in this effort, however, are related to optimizing the infrastructure to support demanding and highly dynamic workloads. While cultural changes with development teams and the lack of training remain the biggest issue for most respondents in the CNCF survey (41% and 40%), infrastructure security, storage, and networking follow closely behind:

- Security (38%)
- Monitoring (34%)
- Storage (30%)
- Networking (30%)

Building a cloud-native platform that provides all these capabilities is a key to success. While traditional dedicated servers provide advanced resources to overcome these challenges, their limited scaling options cannot always meet the needs for instant deployment, scalability, and flexibility.

The phoenixNAP **Bare Metal Cloud (BMC)** offers the elasticity and flexibility of Public Cloud, while at the same time provides performance and isolation of bare metal servers. Powered by the latest generation **Intel® Xeon® Scalable processors**, BMC instances provide a platform for even the most demanding workloads such as high-traffic web servers, SaaS app hosting, database workloads, high performance computing (HPC), machine learning (ML), etc. Launching instances based on the latest 3rd Gen Intel Xeon Scalable processors **(codenamed Ice Lake)** in 2021, BMC expanded the deployment options, enabling access to advanced features of the new generation CPUs.

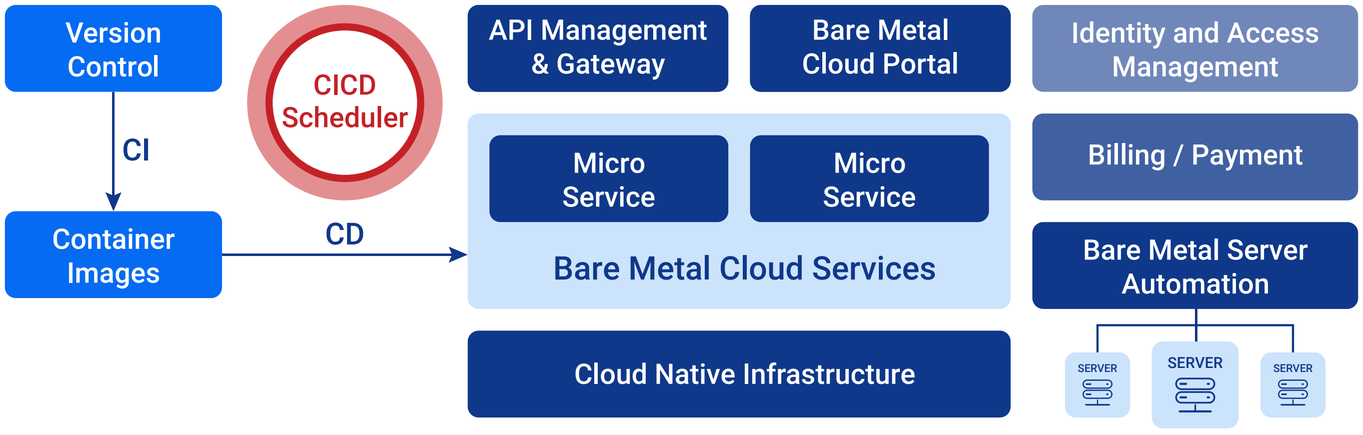
In this article, we discuss the high-level architecture of Bare Metal Cloud, focusing on the APIs that are built on top of **Canonical Metal-as-a-Service (MaaS)** to provide public-cloud-like capabilities and speeds for provisioning and managing of bare metal servers. To demonstrate the capabilities of the system, this article provides an example (with Python code) on how to leverage the BMC API to automate the provisioning of Kubernetes cluster and WordPress application on Bare Metal Cloud.

Architecture Overview

Following the fast-changing cloud market conditions, phoenixNAP adopted microservices architecture, cloud-native infrastructure, and continuous integration/continuous delivery (CI/CD) practice to develop and deliver Bare Metal Cloud in an agile manner. These microservices integrate with third-party solutions to provide an API-based multi-tenant bare metal server platform.

The major components of the phoenixNAP Bare Metal Cloud system include:

- Bare metal automation system
- API management and gateway
- Identity and access management
- Bare Metal Cloud microservices
- Bare Metal Cloud portal
- Version Control and CICD pipeline
- Cloud native infrastructure



Bare Metal Automation System

The heart of the phoenixNAP Bare Metal Cloud is a bare metal automation system that automatically discovers newly racked servers and PXE boot into them to prepare for provisioning. The system is responsible for performing the actual provisioning, de-provisioning, power controls, OS install, and configurations operations. As a scalable and mature bare metal automation solution, Canonical Metal-as-a-Service (MaaS) provides BMC with critical capabilities for making this possible.

MaaS provides a command line interface (CLI), a web user interface (web UI), and a REST API that enables developers to control and query the system. The operations that can be performed through the MaaS API include querying the properties and status of servers, deploying operating systems, initiating power actions on servers (e.g. reboot), and running custom scripts. By leveraging MaaS and its API, we saved significant development efforts as we did not have to build a bare metal automation system from scratch.

Since MaaS is designed and built for enterprises, it does not support multi-tenancy. Thus, we dedicated a big portion of our architecture and development efforts around extending and securing the MaaS API, while providing business capabilities to support multi-tenancy, such as billing and network segregation.

API Management and Gateway

To protect and secure the Bare Metal Cloud API, we utilize Google Apigee, which is an industry-leading API management and gateway solution [2]. We leverage the API gateway functionalities of Apigee, such as rate limiting and OAuth integration to properly protect the API from security threats. For instance, we implement the OAuth 2.0 client credentials flow, which involves using unique client_id and client_secret to generate temporary access token to authenticate and authorize an API session.

In addition, we publish the **BMC API documentation** to the Apigee integrated developer portal, which also provide sandbox capabilities for developers to try out the API directly from the portal.

Identity and Access Management

To integrate phoenixNAP Control Panel (PNCP) authentication and authorization s ystem with ApiGee, we leverage the open source KeyCloak identity and access management solution to facilitate access control to the Bare Metal Cloud portal and API. Specifically, we develop a custom Keycloak service provider interface (SPI), which allows phoenixNAP clients to use their PNCP credentials to access Bare Metal Cloud while leveraging the standard authentication and authorization mechanism provided by Keycloak.

Bare Metal Cloud Microservices

The core business logic of the Bare Metal Cloud system is embedded within a couple microservices. Following is a list of capabilities provided by these microservices:

- Server inventory
- Server provisioning and de-provisioning
- Server power controls
- Network automation
- Billing
- Telemetry

The billing and telemetry microservices keep track of client servers and bandwidth usages. They integrate with an external billing/payment system to handle invoices and payments for clients.

Bare Metal Cloud Portal

By leveraging the BMC API, the portal allows phoenixNAP clients to access the capabilities of Bare Metal Cloud through a modern web interface, which is built with Angular framework and NG Zorro UI widgets. Users of the portal, after authenticating against the Keycloak identity and access management system, can provision servers, deprovision servers, view the status of their bare metal servers, and manage the OAuth client credentials of the API.

Responsive design is incorporated into the interface to optimize the user experience from different types of devices, including desktop/laptop computers, tablets, and smart phones. In addition, the portal is built as a progressive web application (PWA), which can be added to home screens of devices or browsers that has native support for it. Being a PWA enables the BMC Portal to provide the user experience of a traditional application or native mobile application.

Version Control and CICD Pipeline

To facilitate agile software development and delivery, we utilize GitLab version control and CI/CD system to provide continuous integration, testing, and automated deployment of microservices to the cloud native infrastructure. The microservices are built and packaged into Docker containers for deployment.

Cloud Native Infrastructure

The cloud native infrastructure that hosts the microservices consists of Kubernetes (k8s) clusters and Rancher k8s cluster management system. The k8s clusters run on top of VMs deployed on phoenixNAP Data Security Cloud, which is built on top of VMware vCloud platform. Due to the containerization and microservices architecture, these services can be deployed or scaled out to other k8s-compatible cloud if needed.

EXAMPLE: Automating the Provisioning of Kubernetes Cluster

The phoenixNAP Bare Metal Cloud exposes a RESTFul API interface which enable developers to automate the creation of bare metal servers. Detailed information of the API interface is documented at **Bare Metal Cloud API Portal**. Once the bare metal servers are provisioned, additional platform or applications can be provisioned in an automated manner to run on the servers. In this section, we will walk through and highlight a subset of the Python code segments which leverage the BMC API and shell commands to:

1. Create bare metal server instances running **Ubuntu** operating systems
2. Provision a Kubernetes cluster that run on the bare metal servers
3. Install a WordPress application that run on the Kubernetes cluster
4. Install Kubernetes dashboard

The entire code example can be found in the phoenixNAP GitHub **bare-metal-cloud-demo-scripts** repository, where the k8s-demo.py file contains the main program flow.

Get Access Token

Before invoking the BMC API, we need to obtain an OAuth access token using the client_id and client_secret registered in BMC Portal. Steps on how to register for the client_id and client_secret are documented in **Bare Metal Cloud API Portal**.

This is the Python function that gets the access token for the API.

```
def get_access_token(client_id: str, client_secret: str) -> str:
    """Retrieves an access token from BMC auth by using the client ID and the client
Secret."""
    credentials = "%s:%s" % (client_id, client_secret)
    basic_auth = standard_b64encode(credentials.encode("utf-8"))
    response = requests.post('https://api.phoenixnap.com/bmc/v0/servers',
        headers={
            'Content-Type': 'application/x-www-form-urlencoded',
            'Authorization': 'Basic %s' % basic_auth.decode("utf-8")},
        data={'grant_type': 'client_credentials'})

    if response.status_code != 200:
        raise Exception('Error: {}. {}'.format(response.status_code, response.json()))
    return response.json()['access_token']
```

Create Bare Metal Server Instances

Bare metal server instances can be created by making the POST /servers REST API calls and specifying parameters, such as data center location, type of server etc.

This is the Python function that makes a call to the BMC API to create a bare metal server.

```
def __do_create_server(session, server):
    response = session.post('https://api.phoenixnap.com/bmc/v0/servers'),
                            data=json.dumps(server))
    if response.status_code != 200:
        print("Error creating server: {}".format(json.dumps(response.json())))
    else:
        print("{}".format(json.dumps(response.json())))
        return response.json()
```

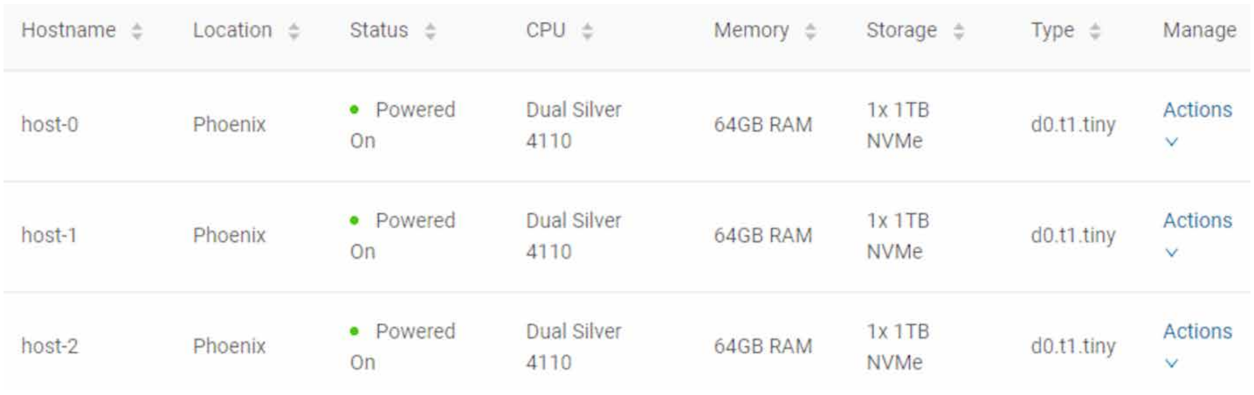
In this example, three bare metal servers of type “d0.t1.tiny” are created, as specified in the server-settings.conf file.

```
{
"ssh-key" : "ssh-rsa xxxxxx== username",
"servers _ quantity" : 3,
"server _ type" : "d0.t1.tiny"
}
```

The output from the Python scripts as a result of generating the token and creating the three bare metal servers are as followed:

```
Retrieving token
Successfully retrieved API token
Creating servers...
{
  “public”: true,
  “id”: “5e848339ed8dd52b946f0386”,
  “status”: “creating”,
  “hostname”: “host-2”,
  “description”: “host-2”,
  “os”: “ubuntu/bionic”,
  “type”: “d0.t1.tiny”,
  “location”: “PHX”,
  “cpu”: “Dual Silver 4110”,
  “ram”: “64GB RAM”,
  “storage”: “1x 1TB NVMe”,
  “privateIpAddresses”: [
    “10.0.3.1”
  ],
  “publicIpAddresses”: [
    “198.15.65.30”,
    “198.15.65.29”,
    “198.15.65.28”,
    “198.15.65.27”,
    “198.15.65.26”
  ]
}
Server created, provisioning host-2...
{
  “public”: true,
  “id”: “5e848339ed8dd52b946f0388”,
  “status”: “creating”,
  “hostname”: “host-1”,
  “description”: “host-1”,
  “os”: “ubuntu/bionic”,
  “type”: “d0.t1.tiny”,
  “location”: “PHX”,
  “cpu”: “Dual Silver 4110”,
  “ram”: “64GB RAM”,
  “storage”: “1x 1TB NVMe”,
  “privateIpAddresses”: [
    “10.0.1.1”
  ],
  “publicIpAddresses”: [
    “198.15.65.14”,
    “198.15.65.13”,
    “198.15.65.12”,
    “198.15.65.11”,
    “198.15.65.10”
  ]
}
Server created, provisioning host-1...
{
  “public”: true,
  “id”: “5e848339ed8dd52b946f0387”,
  “status”: “creating”,
  “hostname”: “host-0”,
  “description”: “host-0”,
  “os”: “ubuntu/bionic”,
  “type”: “d0.t1.tiny”,
  “location”: “PHX”,
  “cpu”: “Dual Silver 4110”,
  “ram”: “64GB RAM”,
  “storage”: “1x 1TB NVMe”,
  “privateIpAddresses”: [
    “10.0.7.1”
  ],
  “publicIpAddresses”: [
    “198.15.65.62”,
    “198.15.65.61”,
    “198.15.65.60”,
    “198.15.65.59”,
    “198.15.65.58”
  ]
}
Server created, provisioning host-0...
Waiting for servers to be provisioned...
```

The following screenshot from Bare Metal Cloud portal lists the three bare metal servers that are created:



Hostname	Location	Status	CPU	Memory	Storage	Type	Manage
host-0	Phoenix	● Powered On	Dual Silver 4110	64GB RAM	1x 1TB NVMe	d0.t1.tiny	Actions
host-1	Phoenix	● Powered On	Dual Silver 4110	64GB RAM	1x 1TB NVMe	d0.t1.tiny	Actions
host-2	Phoenix	● Powered On	Dual Silver 4110	64GB RAM	1x 1TB NVMe	d0.t1.tiny	Actions

Provisioning Kubernetes Cluster

Once the three bare metal servers are created, the script poll the BMC API to check for the server status until the provisioning is completed and the server is powered on. The first server to be provisioned is assigned as the Kubernetes master node. The following Python function performs these steps.

```
def wait_server_ready(function_scheduler, server_data):
    json_server = bmc_api.get_server(REQUEST, server_data['id'])
    if json_server['status'] == "creating":
            main_scheduler.enter(2, 1, wait_server_ready, (function_scheduler,
server_data))
   elif json_server['status'] == "powered-on" and not data['has_a_master_server']:
        server_data['status'] = json_server['status']
        server_data['master'] = True
        server_data['joined'] = True
        data['has_a_master_server'] = True
        data['master_ip'] = json_server['publicIpAddresses'][0]
        data['master_hostname'] = json_server['hostname']
        print("ASSIGNED MASTER SERVER: {}".format(data['master_hostname']))
    else:
        server_data['status'] = json_server['status']
```

Once the servers are provisioned, the Python script make an SSH connection to the servers using its public address to start the installation of **Canonical Microk8s** Kubernetes cluster by invoking the following command line:

```
sudo snap install microk8s --classic --channel=1.17/stable
```

For the master node, the script configures local Kubernetes CLI to connect to the node, and install Kubernetes add_ons, such as DNS and nginx ingress controllers. This is the Python function that installs the add-ons.

```
def setup_k8s_addons(master_ip: str):
   print("Installing k8s add-ons in master server: {}".format(data['master_hostname']))
       run_shell_command([ssh + 'ubuntu@{} \'sudo microk8s.enable dns ingress\''.
format(master_ip)])
```

After the master node is set up, the other two nodes are joined to the Kubernetes cluster. This is the Python function that issues the shell command to perform the join operation:

```
def join_to_cluster(worker_ip: str, master_ip: str, token: str) -> str:
       return run_shell_command([ssh + 'ubuntu@{} sudo microk8s.join {}:25000/{}'.
format(worker_ip, master_ip, token)])
```

At this point, the provisioning of the Kubernetes cluster is completed. Below is the output from the Python script:

```
ASSIGNED MASTER SERVER: host-1
Server provisioned host-0
Installing kubernetes in host-0
Server provisioned host-2
Installing kubernetes in host-2
Server provisioned host-1
Installing kubernetes in host-1
2020-04-01T12:05:20Z INFO Waiting for restart...
microk8s (1.17/stable) v1.17.4 from Canonical* installed

Kubernetes installed host-0
2020-04-01T12:05:31Z INFO Waiting for restart...
microk8s (1.17/stable) v1.17.4 from Canonical* installed

Kubernetes installed host-2
2020-04-01T12:05:26Z INFO Waiting for restart...
microk8s (1.17/stable) v1.17.4 from Canonical* installed

Kubernetes installed host-1
Configure local kubernetes cli for connect to master
Local kubernetes cli configured to connect to master
Installing k8s add-ons in master server: host-1
Enabling DNS
Applying manifest
serviceaccount/coredns created
configmap/coredns created
deployment.apps/coredns created
service/kube-dns created
clusterrole.rbac.authorization.k8s.io/coredns created
clusterrolebinding.rbac.authorization.k8s.io/coredns created
Restarting kubelet
DNS is enabled
Enabling Ingress
```

```
namespace/ingress created
serviceaccount/nginx-ingress-microk8s-serviceaccount created
clusterrole.rbac.authorization.k8s.io/nginx-ingress-microk8s-clusterrole created
role.rbac.authorization.k8s.io/nginx-ingress-microk8s-role created
clusterrolebinding.rbac.authorization.k8s.io/nginx-ingress-microk8s created
rolebinding.rbac.authorization.k8s.io/nginx-ingress-microk8s created
configmap/nginx-load-balancer-microk8s-conf created
daemonset.apps/nginx-ingress-microk8s-controller created
Ingress is enabled

Add-ons installed, the master node is ready to use
Setup servers done
Adding node
Join node with: microk8s join 198.15.65.10:25000/umZAYUifCsbTWUJBNcaJNohFVBNMmtlq

If the node you are adding is not reachable through the default interface you can use one of the following:
 microk8s join 192.168.100.102:25000/umZAYUifCsbTWUJBNcaJNohFVBNMmtlq
 microk8s join 10.0.1.1:25000/umZAYUifCsbTWUJBNcaJNohFVBNMmtlq
 microk8s join 198.15.65.10:25000/umZAYUifCsbTWUJBNcaJNohFVBNMmtlq
 microk8s join 198.15.65.11:25000/umZAYUifCsbTWUJBNcaJNohFVBNMmtlq
 microk8s join 198.15.65.12:25000/umZAYUifCsbTWUJBNcaJNohFVBNMmtlq
 microk8s join 198.15.65.13:25000/umZAYUifCsbTWUJBNcaJNohFVBNMmtlq
 microk8s join 198.15.65.14:25000/umZAYUifCsbTWUJBNcaJNohFVBNMmtlq
 microk8s join 10.1.2.0:25000/umZAYUifCsbTWUJBNcaJNohFVBNMmtlq

umZAYUifCsbTWUJBNcaJNohFVBNMmtlq

Joining node host-2 to master node
Finished: ssh -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no -o LogLevel=ERROR ubuntu@198.15.65.30 sudo
microk8s.join 198.15.65.14:25000/umZAYUifCsbTWUJBNcaJNohFVBNMmtlq

Setup servers done
Adding node
Join node with: microk8s join 198.15.65.10:25000/xBmcEzciEQBGleGisjRaBhrYzmIrmvNb

If the node you are adding is not reachable through the default interface you can use one of the following:
 microk8s join 192.168.100.102:25000/xBmcEzciEQBGleGisjRaBhrYzmIrmvNb
 microk8s join 10.0.1.1:25000/xBmcEzciEQBGleGisjRaBhrYzmIrmvNb
 microk8s join 198.15.65.10:25000/xBmcEzciEQBGleGisjRaBhrYzmIrmvNb
 microk8s join 198.15.65.11:25000/xBmcEzciEQBGleGisjRaBhrYzmIrmvNb
 microk8s join 198.15.65.12:25000/xBmcEzciEQBGleGisjRaBhrYzmIrmvNb
 microk8s join 198.15.65.13:25000/xBmcEzciEQBGleGisjRaBhrYzmIrmvNb
 microk8s join 198.15.65.14:25000/xBmcEzciEQBGleGisjRaBhrYzmIrmvNb
 microk8s join 10.1.2.0:25000/xBmcEzciEQBGleGisjRaBhrYzmIrmvNb

xBmcEzciEQBGleGisjRaBhrYzmIrmvNb

Joining node host-0 to master node
Finished: ssh -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no -o LogLevel=ERROR ubuntu@198.15.65.62 sudo
microk8s.join 198.15.65.14:25000/xBmcEzciEQBGleGisjRaBhrYzmIrmvNb
```

Installing WordPress Application

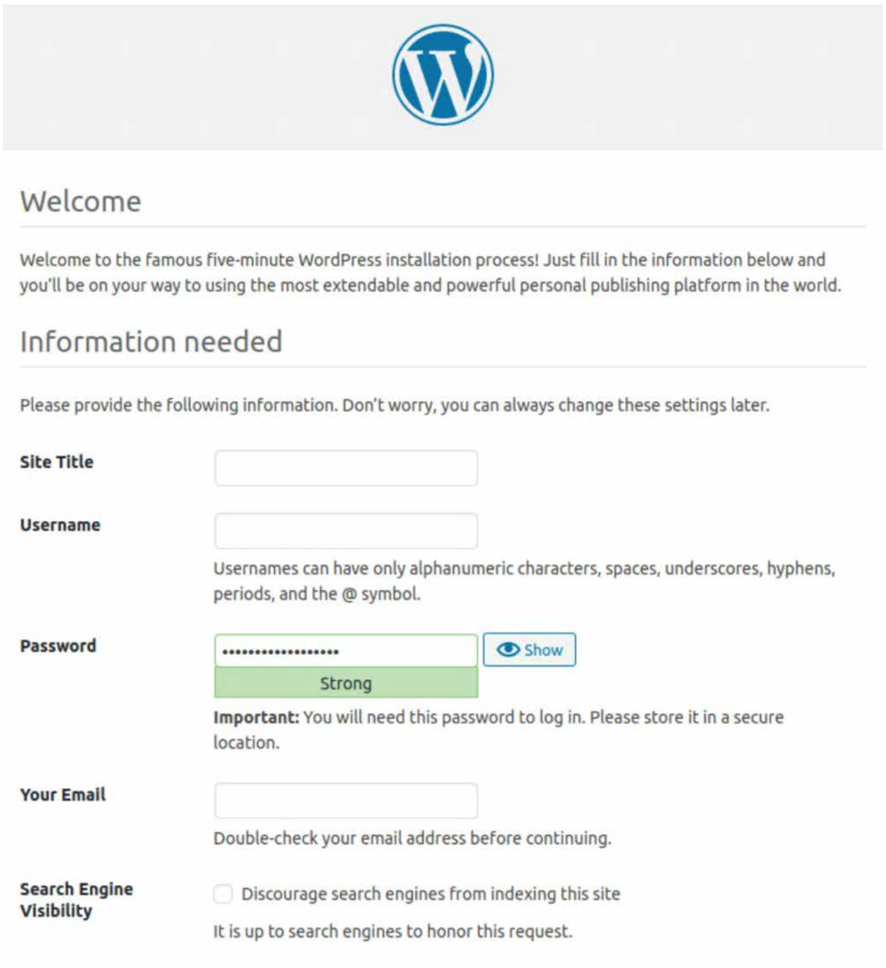
Once the Kubernetes cluster is provisioned, the Python script installs a WordPress application on the cluster. The installation information for the application, such as version number, is specified in the wordpress.yaml file. This is the Python function that invoke the shell commands to install WordPress:

```
def install _ wordpress():
    print("Installing wordpress")
    run _ shell _ command(['kubectl create namespace wordpress')
    run _ shell _ command(['kubectl apply -f ./wordpress.yaml -nwordpress'])
```

Following is the output from running the WordPress application:

```
Installing wordpress
namespace/wordpress created

service/mysql created
deployment.apps/mysql created
service/wordpress created
deployment.apps/wordpress created
ingress.networking.k8s.io/wordpress created
persistentvolume/mysql-pv-volume created
persistentvolumeclaim/mysql-pv-claim created

Wordpress installed
```

Following is the screenshot that displays the welcome screen from the WordPress application that is created:



Welcome

Welcome to the famous five-minute WordPress installation process! Just fill in the information below and you'll be on your way to using the most extendable and powerful personal publishing platform in the world.

Information needed

Please provide the following information. Don't worry, you can always change these settings later.

Site Title

Username

Usernames can have only alphanumeric characters, spaces, underscores, hyphens, periods, and the @ symbol.

Password •••••••••••••••••• Show

Strong

Important: You will need this password to log in. Please store it in a secure location.

Your Email

Double-check your email address before continuing.

Search Engine Visibility ☐ Discourage search engines from indexing this site

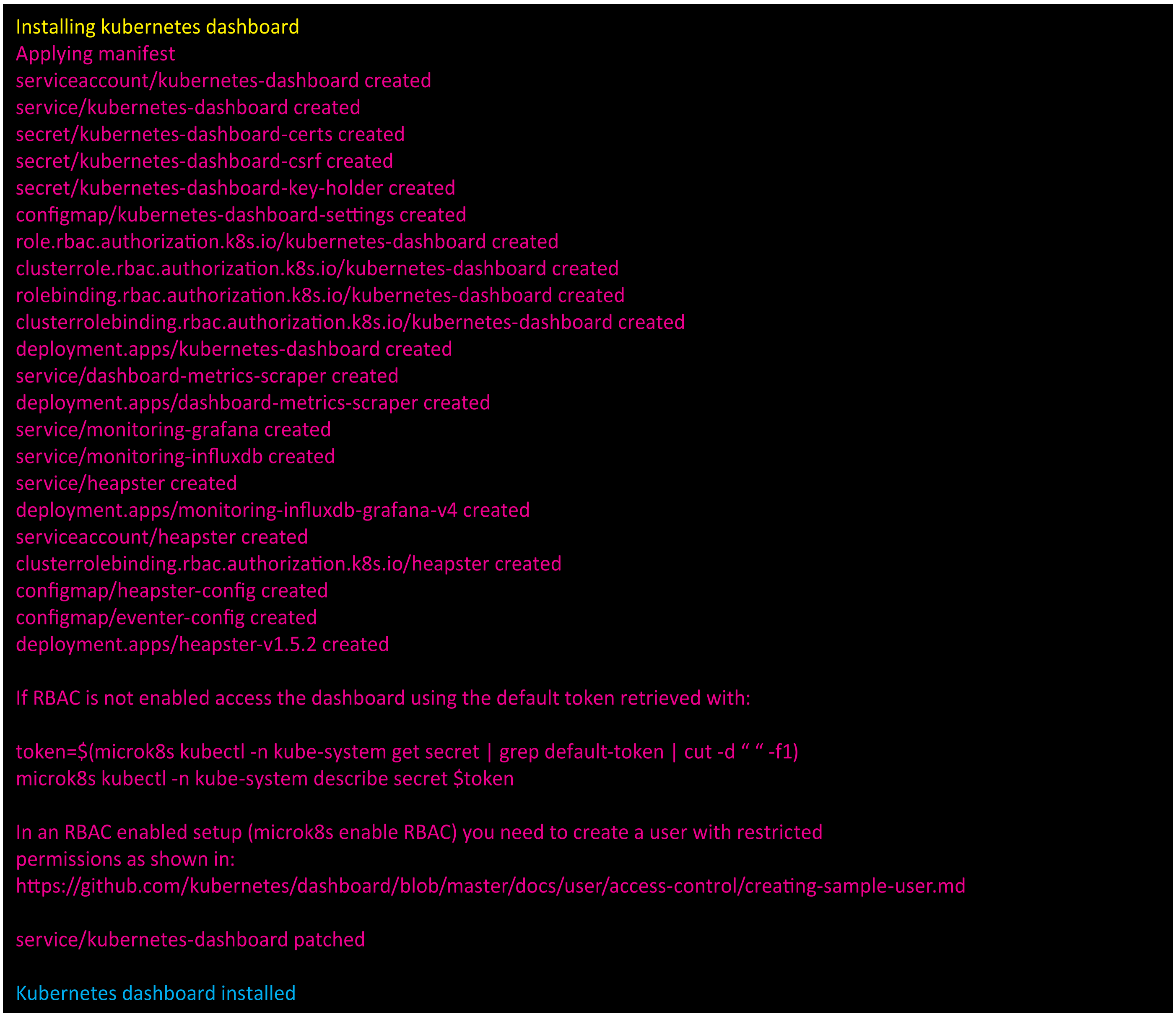
It is up to search engines to honor this request.

Install Kubernetes Dashboard

The last installation the Python script performs is Kubernetes dashboard. The script issues the following shell commands to install Kubernetes dashboard.

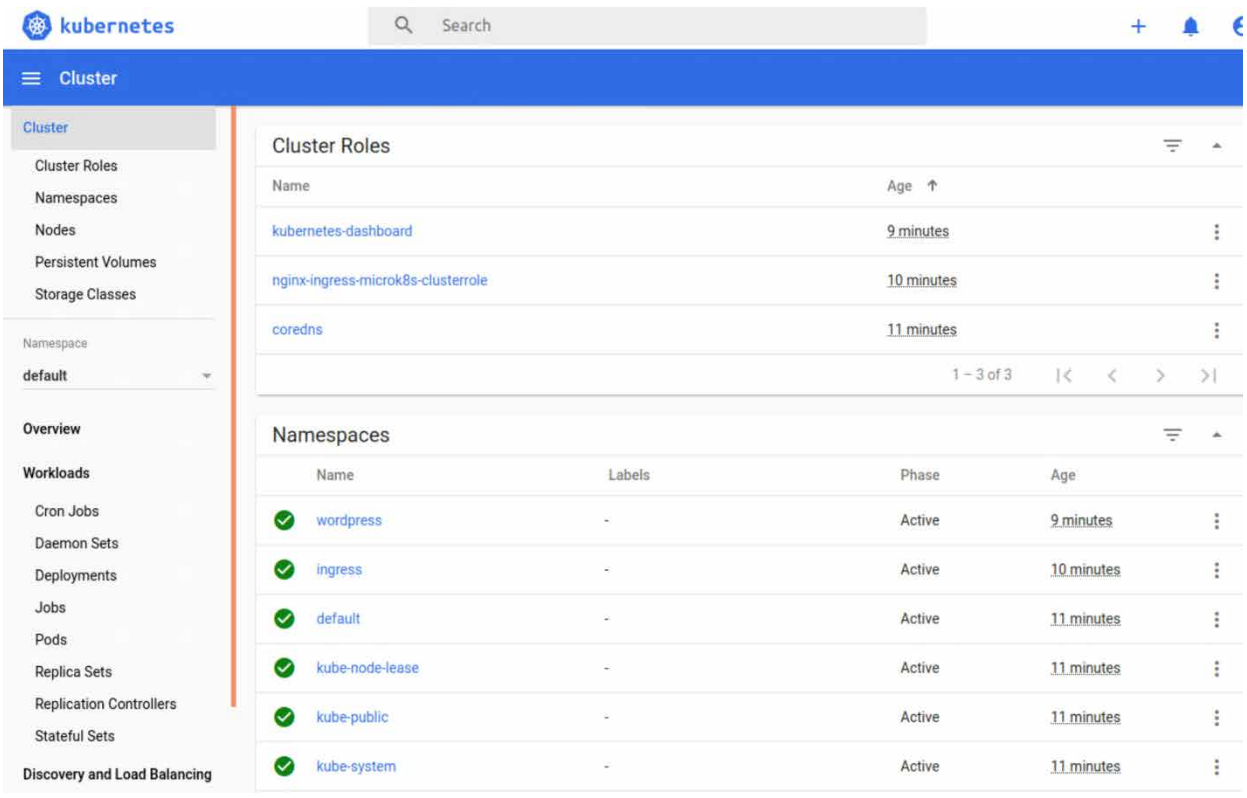
```
sudo microk8s.enable dashboard
sudo microk8s.kubectl -n kube-system patch service kubernetes-dashboard --patch
'{"spec": {"type":"NodePort"}}'
```

The output from running of the commands are:



```
Installing kubernetes dashboard
Applying manifest
serviceaccount/kubernetes-dashboard created
service/kubernetes-dashboard created
secret/kubernetes-dashboard-certs created
secret/kubernetes-dashboard-csrf created
secret/kubernetes-dashboard-key-holder created
configmap/kubernetes-dashboard-settings created
role.rbac.authorization.k8s.io/kubernetes-dashboard created
clusterrole.rbac.authorization.k8s.io/kubernetes-dashboard created
rolebinding.rbac.authorization.k8s.io/kubernetes-dashboard created
clusterrolebinding.rbac.authorization.k8s.io/kubernetes-dashboard created
deployment.apps/kubernetes-dashboard created
service/dashboard-metrics-scraper created
deployment.apps/dashboard-metrics-scraper created
service/monitoring-grafana created
service/monitoring-influxdb created
service/heapster created
deployment.apps/monitoring-influxdb-grafana-v4 created
serviceaccount/heapster created
clusterrolebinding.rbac.authorization.k8s.io/heapster created
configmap/heapster-config created
configmap/eventer-config created
deployment.apps/heapster-v1.5.2 created

If RBAC is not enabled access the dashboard using the default token retrieved with:

token=$(microk8s kubectl -n kube-system get secret | grep default-token | cut -d “ “ -f1)
microk8s kubectl -n kube-system describe secret $token

In an RBAC enabled setup (microk8s enable RBAC) you need to create a user with restricted
permissions as shown in:
https://github.com/kubernetes/dashboard/blob/master/docs/user/access-control/creating-sample-user.md

service/kubernetes-dashboard patched

Kubernetes dashboard installed
```

Below is the screenshot from the Kubernetes dashboard that shows the cluster roles and namespaces of the Kubernetes that the script created:



Cluster Roles

Name	Age ↑
kubernetes-dashboard	9 minutes
nginx-ingress-microk8s-clusterrole	10 minutes
coredns	11 minutes

1 – 3 of 3

Namespaces

Name	Labels	Phase	Age
wordpress	-	Active	9 minutes
ingress	-	Active	10 minutes
default	-	Active	11 minutes
kube-node-lease	-	Active	11 minutes
kube-public	-	Active	11 minutes
kube-system	-	Active	11 minutes

REFERENCES

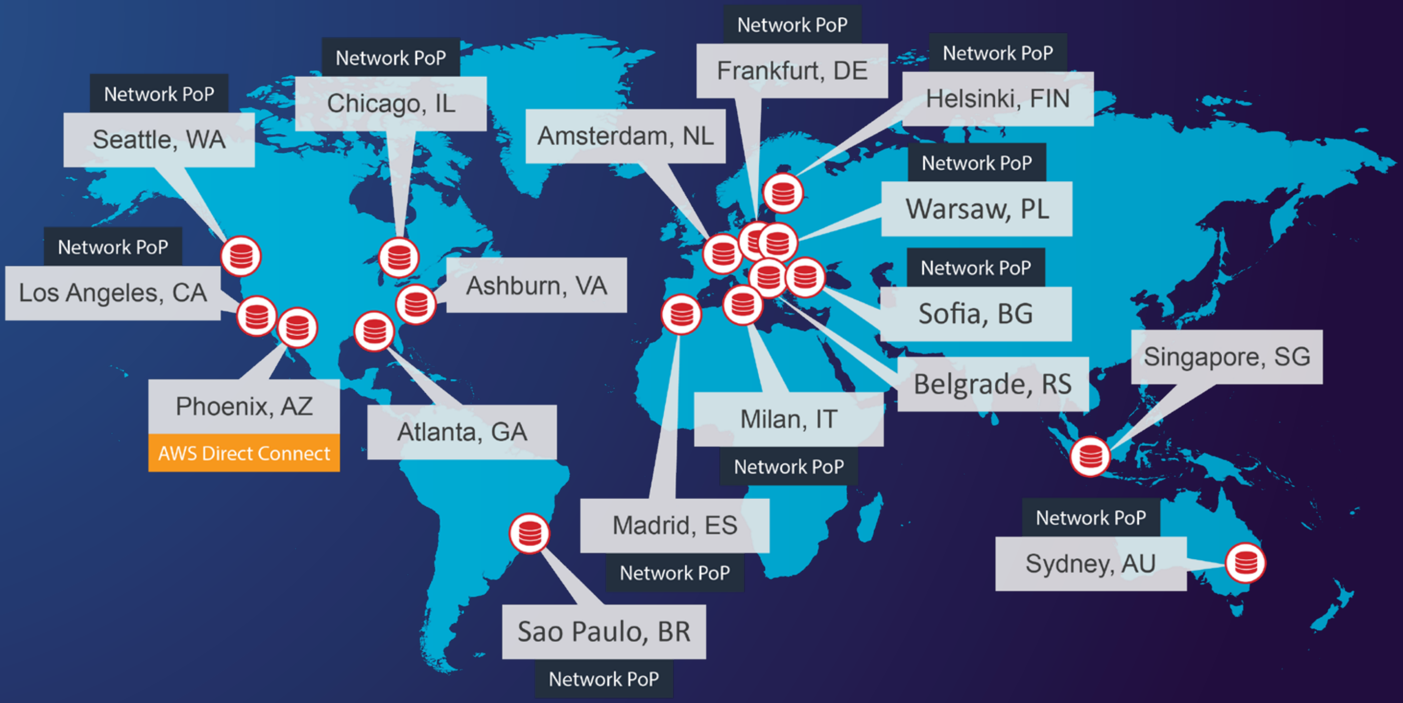
[1] Gartner - Top Emerging Trends in Cloud-Native Infrastructure, May 28, 2019, Arun Chandrasekaran and Wataru Katsurashima.

[2] Gartner – Magic Quadrant for Full Life Cycle API Management, October 9, 2019, Paolo Malinverno, Mark O’Neill, Aashish Gupta, Kimihiko Iijima.

GLOBALLY CONNECTED. LOCALLY AVAILABLE.

2.35 Tbps Bandwidth Capacity | 20,000+ Servers Available Worldwide

100% Network Uptime with World-Class Carrier Blend



April, 2021