Available online www.ejaet.com

European Journal of Advances in Engineering and Technology, 2023, 10(12):35-39

Research Article ISSN: 2394 - 658X

Kubernetes Cluster for Automating Software Production Environment

Naveen Muppa

10494 Red Stone Dr Collierville, Tennessee

ABSTRACT

Microservices, Continuous Integration and Delivery, Docker, DevOps, Infrastructure as Code—these are the current trends and buzzwords in the technological world of 2020. A popular tool which can facilitate the deployment and maintenance of microservices is Kubernetes. Kubernetes is a platform for running containerized applications, for example microservices. There are two main questions which answer was important for us: how to deploy Kubernetes itself and how to ensure that the deployment fulfils the needs of a production environment. Our research concentrates on the analysis and evaluation of Kubernetes cluster as the software production environment. However, firstly it is necessary to determine and evaluate the requirements of production environment. The paper presents the determination and analysis of such requirements and their evaluation in the case of Kubernetes cluster. Next, the paper compares two methods of deploying a Kubernetes cluster: kops and eksctl. Both of the methods concern the AWS cloud, which was chosen mainly because of its wide popularity and the range of provided services. Besides the two chosen methods of deployment, there are many more, including the DIY method and deploying on-premises. Achieves best latency and

Key words: software production environment, production environment, Kubernetes, Amazon Web Services (AWS), Amazon Elastic Kubernetes Service (EKS), operations automation.

INTRODUCTION

Microservices, Continuous Integration and Delivery, Docker, DevOps, and Infrastructure as Code—these are the current trends and buzzwords in the technological world of 2020. A popular tool which can facilitate the deployment and maintenance of microservices is Kubernetes. It is a platform for running containerized applications, for example microservices. Kubernetes provides mechanisms for maintaining, deploying, healing and scaling containerized microservices. Thanks to that, Kubernetes hides the complexity of microservices orchestration. It is much easier to satisfy non-functional requirements of microservices deployment by using Kubernetes. An example of such requirements may be: availability, healing, redundancy or autoscaling. There are two main questions which answer was important for us: how to deploy Kubernetes itself and how to ensure that the deployment fulfills the needs of a production environment.

There are plenty methods of Kubernetes cluster deployment. The methods differ in relation to: how much customization they offer, which clouds they support, how much they cost. Some of the methods has existed since the Kubernetes was created, the other ones, such as AWS EKS, were invented later. There already exist comparisons between different deployment methods of a Kubernetes cluster. However, to the best of our knowledge, these comparisons either: are based on theoretical information (e.g., technical documentation) or are created through a informal, non-academic process and presented as a blog post or they consider only a vanilla cluster.

There exist sources which compare several methods of Kubernetes cluster deployment. However, they are either informal sources (e.g., blog posts or Internet tutorials) or they do not compare the two methods selected in our

research or they do not consider the production environment. Therefore, this paper discussing the comparison of two chosen methods offers the aspects not presented before.

Our goal was to apply a practical approach (by really deploying a Kubernetes cluster, not just reading how to do it), to do it in an automated manner (so that the experiment can be easily reproduced) and to ensure that the cluster is ready to be used in a production environment. Thus, the paper aims to compare two methods of deploying a Kubernetes cluster: kops and eksctl. Both of the methods concern the AWS cloud, which was chosen mainly because of its wide popularity and the range of provided services. Besides the two chosen methods of deployment, there are many more, including the DIY method and deploying on-premises.

Any application may comprise several components, for example: a backend server, a frontend server, database. It is common knowledge that deploying an application to a production environment, should obey a set of guidelines. It should be easy to view all the log messages generated by each of the components of the application, the application should be reachable for its end users and also, it would be useful if in a case of any component failure—that component should be available despite the failure. Kubernetes facilitates satisfying such requirements. However, our aim is to ensure such requirements for Kubernetes itself. A set of requirements were selected. Then, several ideas were provided on how to meet each of the chosen requirements. This makes us believe that the way in which we deployed the Kubernetes clusters was a formal, documented way. Of the proposed CQNS.

PRODUCTION DEPLOYMENT REQUIREMENTS BACKGROUND AND RELATED WORKS

A Kubernetes cluster may be defined as a collection of storage and networking resources which are used by Kubernetes to run various workloads. Another definition states that a Kubernetes cluster is a single unit of computers which are connected to work together and which are provisioned with Kubernetes components. A cluster consists of two kinds of instances: masters and nodes. An instance can be a virtual machine or a physical computer.

Typically, there are two reasons for which multiple environments are in use: to support a release delivery process and to run multiple production instances of the system. The first reason allows having a particular build of an application (e.g., a git commit or a specified version of code) well tested. Such a build has to go through many different environments, e.g., testing, staging and production. When a build does not pass all the stages in the former environments, it will not be promoted to the production environment.

The second reason for multiple environments is that they are used in order to ensure fault-tolerance (when one environment fails, the other can take over), scalability (the work can be spread among many clusters), segregation (it may be decided to handle a group of customers using one environment and the other group with the other environment, e.g., for latency purposes). Well-known examples of running multiple production deployments can be Blue-green deployments or Canary deployments.

The authors of discuss the set of capabilities required for container orchestration platform for design principles. It also provides a guide to identifying and implementing the key mechanisms required in a container orchestration platform.

Throughout this work, a production deployment means such a deployment which targets the production environment. A list of requirements for a production deployment, gathered through the literature, is as follows:

- Central Monitoring—this is helpful when troubleshooting a cluster.
- Central Logging—this is a fundamental requirement for any cluster with number of nodes or pods or containers greater than a couple.
- Audit—to show who was responsible for which action.
- High Availability—authors of go even further and state that the cluster should be tested for being reliable and highly available before it is deployed into production.
- Live cluster upgrades—it is not affordable for large Kubernetes clusters with many users to be offline for maintenance.
- Backup, Disaster Recovery—cluster state is represented by deployed containerized applications and workloads, their associated network and disk resources—it is recommended to have a backup plan for this data.
- Security, secrets management, image scanning—security at many levels is needed (node, image, pod and container, etc.).

- Passing tests, a healthy cluster–'if you don't test it, assume it doesn't work'.
- Automation and Infrastructure as Code—in production environment a versioned, auditable, and repeatable way to manage the infrastructure is needed.

KUBERNETES CLUSTER PRODUCTION DEPLOYMENT REQUIREMENTS AND METHODS

The practical planning and designing the production deployment consider the following activities: capacity planning, choosing which requirements to satisfy and taking different deployment and infrastructure related decisions.

There are numerous requirements for a production deployment of a Kubernetes cluster. In general, the companies, which deploy Kubernetes and similar systems, obey some set of best practices, dedicated to these companies only. Thus, the requirements presented in this work do not exhaust the topic.

In the empirical part of our study, the most important requirements were attempted to be satisfied:

- healthy cluster
- automated operations
- central logging
- central monitoring
- central audit
- backup
- high availability
- autoscaling
- security

The set entails nine requirements. We determined the acceptance criteria needed to satisfy each of the requirements. Based on this, a plan for Kubernetes cluster deployment was created.

AVAILABLE KUBERNETES CLUSTER DEPLOYMENT METHODS

Kubernetes is not trivial to deploy. In order to deploy a usable cluster, there are at least two machines needed: one master and one node. On each of the machines, several components must be installed.

Fortunately, Kubernetes is a popular tool and many methods of deploying it are already described in the literature. The available methods may be divided into three categories:

- self-hosted solutions, on-premises
- deployment in a cloud, but not using Managed Services
- deployment in a cloud, using Managed Services

According to the authors of [5], the best solution is to use Managed Services. The authors argue that, thanks to this method, one can get a fully working, secure, highly available, production-grade cluster in a few minutes and for a little price. Managed Services are certainly a good way to try Kubernetes out. Then, if one wants to do something non-standard or to experiment with the cluster, then one could choose a custom or a hybrid solution. The self-hosted, deployed on-premises way is recommended if the following qualities are of a great importance: price, low-network latency, regulatory requirements, and total control over hardware.

Managed Services offer many features, such as built-in autoscaling, security, high availability, having serverless options. However, they may be more expensive (for example when using AWS EKS, one has to pay $0.10 per hour for each Amazon EKS cluster [46]) and less customizable. Custom solutions allow the Kubernetes administrator to broaden their knowledge and grasp the deep understanding on what is going on under the Kubernetes hood (i.e., one can customize how the Kubernetes control plane is deployed or set up a custom network topology). There is no one right answer which fits all the use cases. It is always advised to do one's own research and to try and experiment with the existing methods.

Furthermore, different categorization may be applied. For instance, the deployment methods may be categorized by the tools:using web interface of a particular cloud, e.g., AWS Management Console (supported by AWS),using command-line tools officially supported by a particular cloud, e.g., awscli or eksctl (supported by AWS),using command-line tools designed exactly to deploy a Kubernetes cluster, but not limited to one particular cloud, e.g., kops,using command-line tools, designed for managing computer infrastructure resources, e.g., Terraform, Salt Stack.

However, our focus is on the two particular methods, which were evaluated for the production environment of software deployment: deploying on AWS, using AWS Managed Service (AWS EKS), using eksctl which is a AWS supported official tool, deploying on AWS, not using any Managed Service, using kops which is a command-line tool, not officially supported by any cloud, but designed exactly to deploy a Kubernetes cluster.

Figure 1: Schema presenting the stages of working with AWS EKS.

COMPARISON OF THE USED METHODS OF KUBERNETES CLUSTER DEPLOYMENT

Several comparison criteria are used in order to compare the two chosen methods of a Kubernetes cluster deployment. Even though the two methods: using kops and using eksctl, help to deploy a Kubernetes cluster, they have some differences. For example: kops works for many clouds (e.g., AWS, GCP), whereas eksctl supports only AWS. Another difference is the default AMI(Amazon Machine Image). kops chooses Debian Operating System, while eksctl uses Amazon Linux 2. These AMIs are, however, only defaults, one can always choose another AMI.

There were nine production environment requirements which were supposed to be met by both tested methods. All of them were met. This means that both of the methods can be applied in production use cases.Provides a brief explanation how each of the requirements was satisfied for each method. Many requirements were handled in the same way for both methods. Subjectively, the hardest requirement to meet was: security, using eksctl. There was a problem with finding a working YAML configuration and also ensuring that cluster was healthy. Eksctl provides a more automated approach. For example the AWS CloudWatch LogGroup, needed for central logging, was automatically created when using eksctl, but not when using kops. Also, eksctl provides the cluster which is already Highly Available. On the other hand, kops provides more flexibility. The master node can be accessed with SSH when using kops and not when using eksctl. When using kops, it is possible.

Requirement	Using *kops* method	Using *eksctl* method
A healthy cluster	The same approach was used. *Bats-core* was chosen as a test framework	
Automated operations	The same approach was used, a *Bash* file *tasks* was used	
Central Monitoring	It was provided by AWS upfront, thanks to *AWS CloudWatch*	
Central Logging	Two *Helm* charts were deployed	It was easy to configure with *YAML*
Central Audit	It was provided by AWS upfront, thanks to *AWS CloudTrail*	
Backup	The same approach was chosen, *Velero* was used with an S3 bucket	
High Availability	It was easy to configure either with *YAML* or *CLI*	It was provided by *AWS EKS* upfront
Autoscaling	The same approach was chosen, ClusterAutoscaler was deployed	
Security	It was easy to set in *YAML*	Finding a working *YAML* configuration was more demanding

Figure 2: Comparison of how each production requirement was satisfied using kops and eksctl.

In production deployment using eksctl, there were one command used to generate cluster configuration (eksctl create cluster) and to deploy the cluster. This command did not return (meaning: it waited) for the cluster to be ready. First, the control plane was deployed, then the worker nodes. For each of these two tasks (deploying control plane and deploying worker nodes) a separate CloudFormation stack was used. It was done automatically by AWS EKS. The control plane is entirely managed by AWS and it is run across three AWS availability zones in order to ensure high availability. The end user has even no access to the control plane, meaning: there is no EC2 instance with master node visible and when listing all Kubernetes nodes, only the worker nodes are visible.

To make the cluster more secure, it was decided to restrict the SSH access and kubectl access. Searching through eksctl configuration, there was no setting which allowed to whitelist the IP addresses which are allowed to ssh login to worker nodes. However, blocking all the SSH access was available and it was chosen. This means now that a cluster administrator has only kubectl access to cluster and no ssh access at all (because ssh access to worker nodes is blocked and access to master nodes is not given by design of AWS EKS). It is acknowledged that sometimes, not being able to ssh to the node may make troubleshooting more difficult, but generally system and EKS logs contain enough information for diagnosing problems. The second security

requirement was to restrict the access to cluster through kubectl. By default, eksctl exposes the Kubernetes API server publicly but not directly from within the VPC subnets. This means that the public endpoint is by default set to true and the private endpoint to false.

REFERENCES

[1]. https://aws.amazon.com/containers/?gclid=CjwKCAjwydSzBhBOEiwAj0XN4CgpkvD386mV84-PtgHFX1CCcO5VWms6fubduXGgyq3ISxdHxTxJ-RoCBhQQAvD_BwE&trk=0f01809b-4c0c-4861-bdfb-d8fb615f8170&sc_channel=ps&ef_id=CjwKCAjwydSzBhBOEiwAj0XN4CgpkvD386mV84-PtgHFX1CCcO5VWms6fubduXGgyq3ISxdHxTxJ-RoCBhQQAvD_BwE:G:s&s_kwcid=AL!4422!3!641418343893!p!!g!!managed%20kubernetes!19256168360!141344069981

[2]. https://kubernetes.io/docs/setup/production-environment/