KubeCOM: An implementation of a non-containerized software management system based on Kubernetes

Guangyin Shi*, Weiwei Cai, Jiawei Zhang, Chuanji Gao, Siqing Sun, Yong Zhang, Yan Jiang
Department of Cloud Platform Research and Development, Inspur Cloud, Jinan, Shandong, China

ABSTRACT

The unified deployment and management of non-containerized software is the prerequisite and key to achieve efficient management of cloud computing platforms. With the rapid development of information technology, Kubernetes technology is widely employed in container orchestration and cluster deployment. As a novel generation of containerized application cluster platform, Kubernetes has the advantages of automatic resource scheduling mechanism and elastic scaleup in a single cloud platform. With the development of hybrid clouds such as private cloud, government cloud, and local cloud, traditional non-containerized software management methods are stretched to the limit. In this paper, we present a declarative non-containerized software management approach based on Kubernetes, and elaborates the system management and deployment software in detail. The management system takes full use of the advantages of the Kubernetes platform, and combines the characteristics of automatic management, Operator and CRD technology, etc., to provide a stable, reliable and powerful guarantee for the automatic deployment and management of hybrid cloud platform resources.

Keywords: Non-containerized software, cloud component, deployment, model, controller, actuator

1. INTRODUCTION

Modern clusters generally consist of thousands of machines, which are used to provide stable, high-efficiency, and automatically scalable network services, such as scientific computing[1], data services[2], and e-commerce. Regarding of researches on modern clusters, most of them primarily focus on how to optimize algorithms to improve scheduling efficiency, while ignoring other researching areas on cluster management and deployment[3]. With continuous iterative updates of complex business scenarios and business requirements, a portion of applications have successfully been containerized deployment and management based on technologies such as containers and micro-services. However, containerized applications could not involve all application scenarios. For example, some applications can only exist in the cluster as a unified whole, but cannot be designed as containerized applications, in which applications may carry inter-dependencies. Thus, the deployment of containerized applications is also difficult to achieve.

In general, there are still many challenges to deploy and manage non-containerized applications in a cluster. Traditional clusters based on Kubernetes cannot enable efficient deployment of non-containerized software, let alone unified management of software in a multi-cloud environment. At the usage level, users only need to focus on the final status of the software, but not the deployment process of the software. At the development level, there are higher management requirements for R&D personnel to implement the deployment and management of various resources in the cluster. At the operation and maintenance level, during the process of managing non-containerized software, the traditional management methods are too complicated to deploy and manage and cannot meet the various needs of customers, which is too dependent with human experience.

Several methods have been proposed to handle the above problems to some extent. The Ansible software proposed by Hochstein et al.[4] realizes convenient management and effective deployment of applications. However, it suffers from lower execution performance, and the resource consumption on nodes is too high to afford, which may even affect other running programs. In addition, its failed execution status does not support retry function and has no ability to effectively feedback operating status, which could not meet the requirements of large-scale cluster automation management. Zadka et al.[5] mentioned a configuration management system tool (named Salt) with systematically implementing software deployment. But it has shortcomings such as high learning cost and long familiarity time, and it is incompatible with Kubernetes and could not meet the requirements of unified management of cloud platforms. Therefore, in the process of

* 13906102@qq.com

Third International Conference on Computer Science and Communication Technology (ICCSCT 2022)
edited by Yingfa Lu, Changbo Cheng, Proc. of SPIE Vol. 12506, 1250654
 · 0277-786X · doi: 10.1117/12.2662548

deploying and managing large-scale non-container application environments, traditional application deployment management methods significantly reduce work efficiency[6].

The deployment and management of non-containerized applications in large-scale clusters have the above problems such as harsh requirements and complex management chains. It was depicted that a systematic and comprehensive study abstracts the non-containerized software in the cluster and encapsulates the data. Components are managed in the form of cloud components, and proposed a declarative cloud component management system (named KubeCOM), through which our contributions are summarized as follows.

(1) We propose an automatic method that implemented the full-life cycle management of multi-node cloud components.

(2) We design the declarative definition the final form of cloud components without focusing on the deployment process of cloud components.

(3) We propose an efficient scheduling algorithm for cloud component actions, which can filter nodes according to the configuration to manage cloud components.

(4) The cloud component management system is designed to feedback status by the automatic retry function after multiple components' actions fail.

(5) We show that the management system simplifies the operation of R&D personnel, reduces the time for managing cloud components, and improves the efficiency of cluster management and deployment.

2. SYSTEM DESIGN

The cloud component management system is depicted in Figure 1, and it aims to provide an efficient management system that is compatible with Kubernetes and has a unified syntax. In terms of data model and scheduling algorithm, the management system proposes cloud component resource models, scheduling algorithms and strategies for cloud component actions.

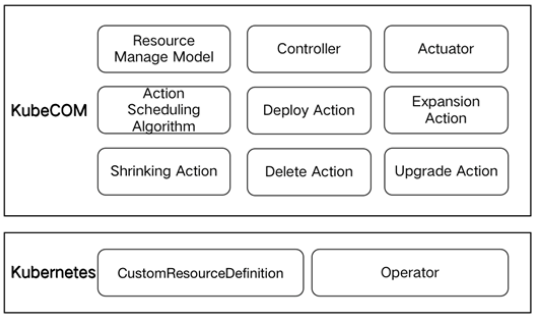


Figure 1. Schematic diagram of KubeCOM architecture.

The component management system consists of cloud component resource model, controller and actuator.

2.1 Cloud component resource model

Based on the characteristics of non-containerized software deployment and management, this paper abstracts and proposes the concept of data resource model. The data resource model includes cloud component definition resources, cloud component deployment resources, cloud component cluster resources, cloud component node resources, and cloud component upgrade resources. The data resource model is based on the CRD technology of Kubernetes, and abstracts and designs management actions of non-containerized software, which is difficult to manage effectively in traditional clusters[7].

As shown in Figure 2, cloud component definition resources include meta data such as component name, component version, component image, and desired state. The cloud component deployment resource refers to the cloud component definition resource, including the deployed component name, deployment component version, deployment component configuration parameters, the affinity node to which the deployment component points, and the deployment scheduling policy and other meta data. The cloud component cluster resource extracts important characteristics of the cluster,

including meta data such as cluster name, cluster version, cluster parameters, cluster image repository, current cluster status, and expected cluster status. Cloud component node resources contain meta data such as node name, node IP, current state, and desired state. Cloud component upgrade resources include meta data such as cloud component names and cloud component upgrade target versions. Cloud component cluster resources manage cloud component node resources; cloud component deployment resources filter appropriate nodes for component deployment based on cloud component node resources.

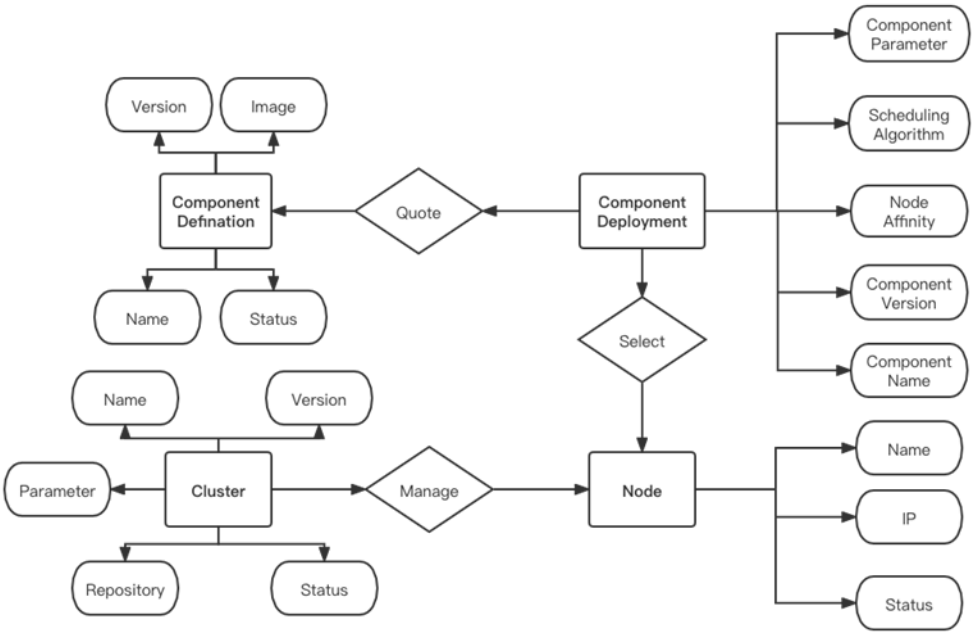


Figure 2. Schematic diagram of cloud component definition resources.

2.2 Scheduling algorithms and strategies for cloud component actions

The native scheduling algorithm of Kubernetes could not achieve effective scheduling of custom resources or support the scheduling strategy of cloud components' actions, which could be result from no consideration of the dependencies between cloud components. In general, the scheduling policies for cloud component actions cloud be divided into inter-component scheduling policies and single-component scheduling policies. Since the critical path of cloud components' actions plays a critical role in actions' scheduling, an optimal scheduling strategy was proposed for the critical path between components to achieve optimal deployment and upgrade of cloud component functions. The strategy analyzes the shortest path and completes the corresponding actions according to the shortest path. There are three dependencies (identical parent structure, V-shaped structure and sequential structure) between components in the cluster. The scheduling algorithm outputs the shortest path after logical calculation, which is the critical path optimal solution for inter-component scheduling.

Shortest Path Upgrade Algorithm

Step 1: Determine the total upgrade path according to the fromVersion and toVersion in the cluster upgrade resource group

Step 2: Check the upgrade resources in the cluster, and determine that the upgrade resources in the cluster exist and could complete the upgrade of the total path according to the fromVersion and toVersion inside each upgrade resource

Step 3: Traverse each upgrade resource, find the upgrade resource whose fromVersion is consistent with the fromVersion of the cluster, determine the upgrade resource with the longest initial upgrade path according to its corresponding toVersion, and use its toVersion as the next initial fromVersion

Step 4: Step 3 recursively, record all upgrade paths

Step5 : Output the shortest path where the toVersion of the upgrade resource group is consistent with the toVersion of the cluster

---

To accommodate various scheduling scenarios of a single cloud component, the scheduling policies within a single component are divided into the following three types:

(1) Action rolling execution strategy: In order to ensure the high availability of component functionality, this policy ensures the external responsiveness of the component during the action execution process. This is expressed as follows: action sequences are executed on each node, and the actions will be completed when the previous node entity completes its action. The action of the next entity is initiated.

(2) Delayed execution policy for destructive actions: To ensure the external availability of component functions, this policy ensures that the timing of triggering destructive actions needs to be manually confirmed by a specific person at a specific time. This is shown by the fact that when the action is triggered, the execution of the action is reduced during policy validation process. The weight ratio is the lowest.

(3) Action execution node affinity policy: To make sure that the component entity runs on the target node specified by the user, the policy deploys the component entity on the target node according to the declarative node selection function. The corresponding action will be performed on that node.

3. SYSTEM IMPLEMENTATION

The cloud component management system is implemented based on Kubernetes, which abstracts and proposes the concept of cloud components with non-containerized software as the management object, supports the resource management of cloud components, and promote rapid developments and iterations, elastic scaleup and intelligent operation and maintenance of non-containerized software on cloud platforms. The management system proposes a declarative cloud component management system, which is consists of the following parts: the cloud component resources and management model, the cluster-level cloud component management control unit controller, the node-level cloud component execution and monitoring unit actuators, and logical solutions for resolving declarative resources (including implementing management functions) and scheduling algorithms for cloud component actions.

3.1 Cloud component management and control at the cluster level

The management model includes architectural information, which is adapted to the management model. It is divided into a cluster-level controller from the server side and a node-level actuator from the agent side. The main functions of cloud component resources include deployment, scaleup, scaledown, upgrade, and deletion.

The server-side controller is responsible for parsing the resource files of cloud components and executing the corresponding processing logic. It also monitors the changes of cloud component resources on the control node, analyzes different parts of the resource changes through the internal policy clusters, and executes differentiated actions simultaneously. As shown in Figure 3, actions of the cloud component management part are: deploying components, expanding components, and deleting components. During the execution of the action, the progress and logs of the current cloud components can be conveniently visualized through commands.

3.2 Cloud component actuators and monitoring unit actuators at the node level

As shown in Figure 4, agent actuators are deployed on each child node to monitor resource changes of the cloud component node entity (also called CkeNode), and perform corresponding processing actions through logical processing of state changes of the cloud component entity.

The above operation logic is: The controller registers the custom CRD resource into Kubernetes cluster to monitors the creation and change of cloud component resources. After the successful initialization of controller, the corresponding resource types are resolved through the registration monitoring mechanism of Kube-Apiserver to handle the differentiated actions of the cloud components with different logical modes. In these components, the controller computes the optimal node and the optimal order through the scheduling algorithm according to the declarative node configuration and scheduling policy. In the individual component, the controller computes the optimal choice between multiple actions according to the declarative policy cluster, and obtains the optimal solution for the performing the differentiated actions. The controller will create a cluster-level cloud component template, and simultaneously instantiate

the node-level component entity on the child node specified by the cloud component. The cluster-level template of the cloud component is processed to trigger changes in the node-level component entities to achieve multiple logical processing.

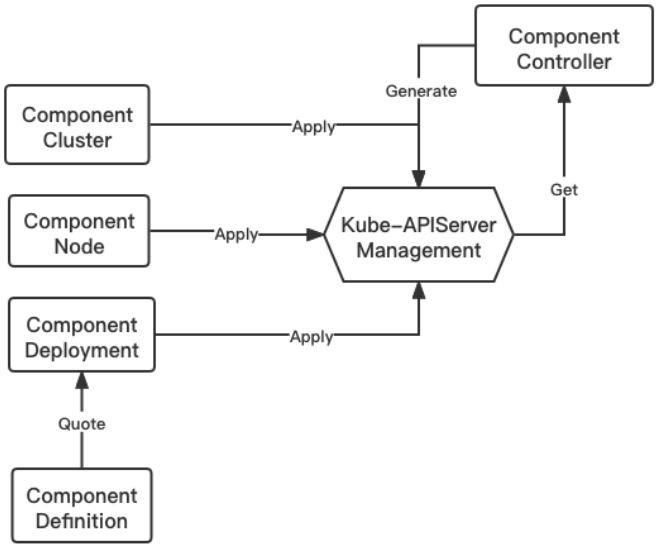


Figure 3. Actions supported by the cloud component framework on the server side, a flowchart of actions such as scaleup, scaledown, update, upgrade, deployment, and deletion.

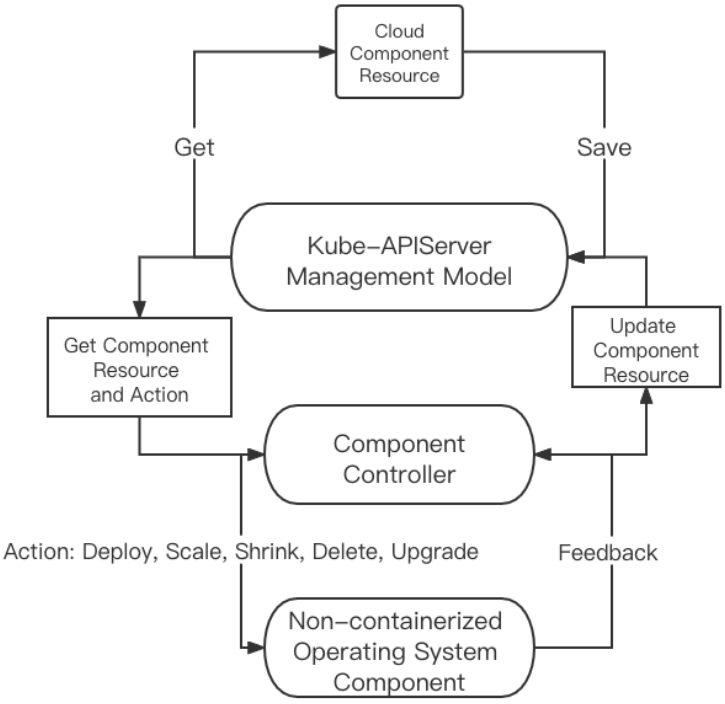


Figure 4. The flow chart of the scaleup, scaledown, update, upgrade, deployment and deletion of the agent-side cloud component framework.

3.3 Cloud components perform actions

Deployment means that when the controller monitors the creation of resources related to new components, it triggers the deployment logic and instantiates cluster-level component templates and node-level component entities in turn to complete the deployment of cloud components. Taking the CkeNode's data resource model as an example, its deployment file is as follows:

```
apiVersion: cke.inspur.com/v1alpha1
kind: CkeCluster
metadata:
  name: cke-v5-component
  namespace: kube-system
spec:
  version: 5.0.0-20210608_152428
  registry:
    ip: ${registry}
    domain: registry-jinan-lab.inspurcloud.cn
  cluster_vip:
    ip: ${cluster_vip}
  cluster_parameters:
    image_manifest_enabled: false
    kube_service_cidr: 10.105.0.0/16
    kube_pods_cidr: 10.101.0.0/16
    override_system_hostname: true
    apiserver_domain_name: vip-apiserver.inspur.com
---
apiVersion: cke.inspur.com/v1alpha1
kind: CkeNode
metadata:
  name: master1
  namespace: kube-system
  annotations:
    instanceID: 6f4f6c99-b6dd-4ed6-92c7-30aaca4ac197
    resources: 4C/8G/50G
    install_net_address: ${master1Ip}
  labels:
    cie.inspur.com/cluster: "true"
    node-role.kubernetes.io/master: "true"
```

```
      node-role.kubernetes.io/node: "true"
spec:
  address: ${master1Ip}
---
apiVersion: cke.inspur.com/v1alpha1
kind: CkeNode
metadata:
  name: master2
  namespace: kube-system
  annotations:
    instanceID: 6f4f6c99-4ed6-b6dd-92c7-30aaca4ac197
    resources: 4C/8G/50G
    install_net_address: ${master2Ip}
  labels:
    node-role.kubernetes.io/master: "true"
    node-role.kubernetes.io/node: "true"
spec:
  address: ${master2Ip}
---
apiVersion: cke.inspur.com/v1alpha1
kind: CkeNode
metadata:
  name: master3
  namespace: kube-system
  annotations:
    instanceID: 6f4fac197-b6dd-92c7-4ed6-30aaca4a6c99
    resources: 4C/8G/50G
    install_net_address: ${master3Ip}
  labels:
    node-role.kubernetes.io/master: "true"
    node-role.kubernetes.io/node: "true"
spec:
  address: ${master3Ip}
```

Scaleup indicates that when the controller monitors the increase in the number of node replicas pointed to by the component's node selector, it will trigger the scaleup logic by creating a new node component entity and modifying the

entity state to the state to be expanded. The scaleup will be completed by waiting for the actuator to complete the scaleup logic. An example of the expanded data resource model is as follows:

```
apiVersion: cke.inspur.com/v1alpha1
kind: CkeNode
metadata:
  name: node1
  namespace: kube-system
  annotations:
    instanceID: 30aa6c99-b6dd-4ed6-92c7-ca4ac1976f4f
    resources: 4C/8G/50G
    install_net_address: ${nodeIp}
    install_net_port: "6233"
  labels:
    node-role.kubernetes.io/node: "true"
spec:
  address: ${nodeIp}
```

The scaling, updating, and deleting actions of the cloud component are all done by the controller monitoring the node status and triggering the corresponding logic. Specifically, in the case of scaling, when the controller detects that the number of node replicas pointed to by the node selector of the component decreases, it triggers the scaling logic, and modifies the scaling node component entity to the state to be scaled down. The scaling of component is completed when the scaling logic is executed.

The monitoring unit actuator is deployed in each sub-node with high availability to monitor the resource change of the cloud component node entity and execute corresponding processing actions through logical processing of the cloud component entity state changes. The operation logic of the node actuator is as follows: the actuator runs as a resident process on a single node, the node runs and monitors the cloud component entity of the current node and when the component entity changes, it performs related processing through the internal action processing model.

The actuator determines the action categories according to the current node state of the cloud component: deploy, expand, delete, upgrade, and execute the script through the preset template. Through the declarative profile definition, the actuator has the ability to access to Api-server in different clusters and can monitor and change a variety of actions such as component custom resources. For the reason of efficiency, the actuator has a built-in retry mechanism. After the current action fails for some reason, the actuator will automatically trigger the retry mechanism when it detects the failure status of the action, and the retry trigger time will be weighted with the number of times to extend the execution time.

The management system proposed in this paper enables fine-grained control of cloud components. For a single cloud component, it includes deployment, scaleup, deletion, and upgrade operations of cloud component resources.

Taking the etcd component as an example, the example of its component deployment is as follows:

```
apiVersion: cie.inspur.com/v1alpha1
kind: Component
metadata:
   name: etcd-3.4.9-10
   namespace: kube-system
spec:
   deleteConfirm: true
   depends:  - ntp|chrony
   deployImage: library/cke/components/etcd:3.4.9-10
   images:
      etcd: library/cke/etcd/etcd:v3.4.9-2
   scaleUpdate: true
   version: 3.4.9-10
---
apiVersion: cie.inspur.com/v1alpha1
kind: Comdeploy
metadata:
   name: etcd-deploy
   namespace: kube-system
spec:
   component:
      name: etcd
      version: 3.4.9-13
   nodeSelector:
   - node-role.kubernetes.io/master=true
   parameters:
      data_dir: /var/lib/etcd
      election_timeout: "5000"
      heartbeat_interval: "1000"
   policy:
   - UpgradeOneByOne
   - ScaleOneByOne
```

Once the component is successfully deployed, the controller will listen to the individual cloud component. Once the template parameters of the component are modified, the corresponding logic is triggered. The controller modifies the child nodes of the cloud component to the corresponding state, and the actuator performs the corresponding component operation until the cloud component state matches the template definition.

4. EXPERIMENT AND VERIFICATION

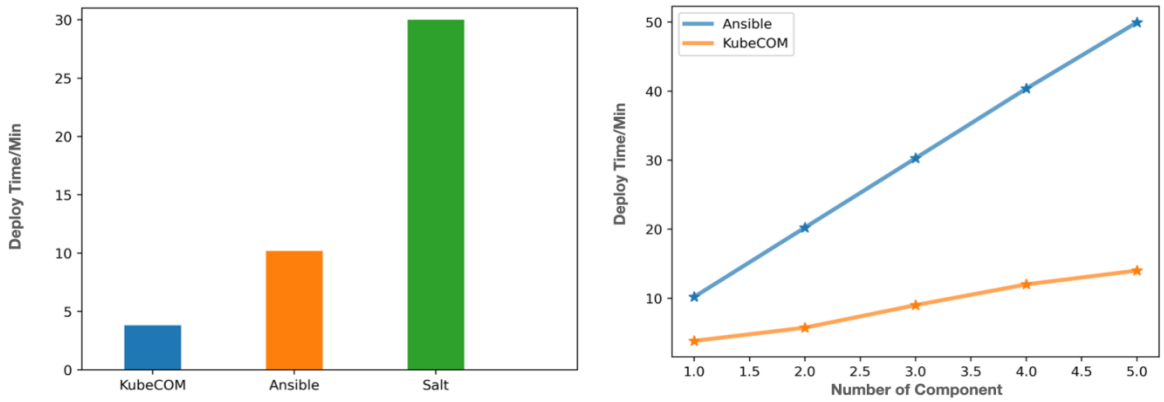
4.1 Experimental environment

To validate the resource deployment and management capabilities of the cloud component management system in a large-scale cluster, this paper builds a 5-master node (4CPU, 8GB RAM and 50GB disk memory) and 100 nodes (4CPU, 8GB RAM and 50GB disk memory) and integrates the developed cloud component management system into the above cluster.

4.2 Experimental results and analysis

Through the large-scale Kubernetes cluster of the five master nodes as describe above, this paper compares KubeCOM and other deployment tools (Salt and Ansible) in terms of ease of use and component deployment efficiency.

To demonstrate the ease of use of the cloud component framework and the advantages of unified management, we compare the deployment processes of KubeCOM, Ansible and Salt by deploying etcd services in a large-scale cluster of 100 nodes from an operational perspective. Both Ansible and Salt need to prepare multiple configuration files during deployment. The configuration process is complex and the threshold for use is high. Both of them need to distribute etcd zip files to each node and perform the compilation and deployment process. KubeCOM's deployment file is clean and simple as showed above in Section 3.3. It only requires the image file and image file of the component, which leverages the existing information of the cluster (nodes' information, etc.) and enables the deployment of the component conveniently.



(a) (b)

Figure 5. For (a) and (b), the deployment efficiency is measured by consuming time when etcd is successfully deployed: (a) Single component deployment efficiency comparison between Ansible, Salt and KubeCOM; (b) is the multi-component deployment efficiency comparison between Ansible and KubeCOM.

In the same cluster environment (100 nodes), the comparison of efficiency between KubeCOM, Ansible and Salt when deploying a non-containerized software (etcd). As shown in Figure 5a, for a 100-node cluster (shown in Table 1), the deployment of a single component takes a relatively high time-consuming for Salt, almost 30 minutes, while the Ansible deployment tool takes only about 10 minutes and KubeCOM takes 3 minutes. Therefore, Salt is meaningless for further comparison and the following tests of the deployment of components for KubeCOM and Ansible are employed to make a comprehensive comparison between them. During the test, multiple components are replaced by deploying multiple etcd components in order to ensure the test effect and deployment stability. In the same cluster environment (100 nodes), when deploying the same number of components, the comparison of deployment efficiency between KubeCOM and Ansible along with the number of components, where the solid line is the change trend of deployment efficiency with the scale of components, and the asterisk points represent the deployment time of different numbers of components. As shown in Figure 5b, it can be seen that although the single-component deployment time of Ansible is moderate (10min), its deployment efficiency is linearly related to the deployment scale, so Ansible is not suitable for multi-component deployment and application in large-scale clusters. It could be figure out that the efficiency of proposed cloud component management system KubeCOM is much higher than that of Ansible. The comparison in deployment efficiency if

particularly evident when the five components are deployed. The deployment efficiency of the cloud component management system is more than four times that of Ansible (Table 1).

Table 1. Efficiency comparison between cloud component management framework and Ansible deploying different numbers of components.

Number	Ansible	KubeCOM
1	10m11s	3m49s
2	20m13s	5m44s
3	30m17s	9m10s
4	40m21s	12m3s
5	49m57s	14m1s

Finally, the above results in this paper indicate that KubeCOM effectively achieves the automatic management of the full life cycle of multi-node cloud components and solves the problems of difficult operation and maintenance, large-scale cluster deployment, and slow upgrade. The declarative definition of the final status and form of cloud components tremendously simplifies the manual operations and promotes the efficiency of cloud component management. The management system also supports automatic retry of cloud components after multiple operation failures, which improves the high availability of the cluster. Its multiple declarative configurable policies enable the flexible choice of multiple complex scenarios.

5. SUMMARY

In view of the resource management problem in large-scale cloud computing scenarios, we build a declarative cloud component management system based on Kubernetes. It also provides customized functions for cluster deployment and management, cluster scaling-up and contraction based on real management application requirements. The cloud platform resource model is defined according to the Operator and CRD technologies and customizes the arrangement and scheduling of cloud platform resources through the unified yaml format, which provides the function of distributed concurrent execution of extended tasks. After a full testing on the large-scale cluster, the management system has achieved the deployment and upgrade of a 200-node cluster at the minute level, which significantly improves the performance and efficiency of cloud components deployment, implements a unified management plane for cloud platform resources, promotes operation and maintenance efficiency, unifies the cluster operation and maintenance language. Finally, the system enhances the continuous delivery capability of the cluster and the stability of application services.

REFERENCES

[1] Tirmazi, M., Barker, A., Deng, N., Haque, M. E., Qin, Z. J., Hand, S., Harchol-Balter, M. and Wilkes, J., "Borg: The next generation," Proc. of the Fifteenth European Conf. on Computer Systems, 1-14(2020).

[2] Burns, B., Grant, B., Oppenheimer, D., Brewer, E. and Wilkes, J., "Borg, omega, and Kubernetes," *Communications of the ACM,* 59(5), 50-57(2016).

[3] Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M. and Wilkes, J., "Omega: Flexible, scalable schedulers for large compute clusters," *Proc. of the 8th ACM European Conf. on Computer Systems,* 351-364(2013).

[4] Hochstein, L. and Moser, R., "*Ansible: Up and running: Automating configuration management and deployment the easy way,"* Roles: Scaling Up Your Playbooks, O'Reilly Media, Inc., Chapter 7, 127-141(2017).

[5] Zadka, M., [*DevOps in Python: Infrastructure as Python],* Apress, Berkeley, CA, Chapter 10 Salt Stack, 121-137(2019).

[6] Huang, Z., Wu, S., Jiang, S., and Jin, H., "FastBuild: Accelerating docker image building for efficient development and deployment of container," *Proc. of the 2019 35th Symp. on Mass Storage Systems and Technologies,* IEEE, Piscataway, 28(2019).

[7] Yilmaz, O., [*Extending the Kubernetes API*], Apress, Berkeley, CA, 99-141(2021).