HCA Operator: A Hybrid Cloud Auto-scaling Tooling for Microservice Workloads

***Abstract*—Elastic cloud platform, e.g. Kubernetes, enables dynamically scale in or out computing resources in accordance with the workloads fluctuation. As the cloud evolves to hybrid, where public and private clouds co-exist as the underline substrate, autoscaling applications within a hybrid cloud is no longer straightforward. The difficulty lies in all aspects, e.g. global load balancing, hybrid-cloud monitoring and alerting, storage sharing and replication, security and privacy, etc. However, it will significantly pay off if hybrid-cloud autoscaling is supported and boundless computing resources can be utilized per request. In this paper, we design Hybrid Cloud Autoscaler Operator (HCA Operator), a customized Kubernetes Controller that leverages the Kubernetes Custom Resource to auto-scale microservice applications across hybrid clouds. HCA Operator load balances across hybrid clouds, monitors metrics, and autoscales to destination clusters that exist in other clouds. We discuss the implementation details and perform experiments in a hybrid cloud environment. The experimental results demonstrate that if the workload changes quickly, our Operator can properly autoscale the microservice applications across hybrid cloud in order to meet the Service Level Agreement (SLA) requirements.**

***Index Terms*—Hybrid Cloud, Auto-scaling, Kubernetes Operators, Kubernetes, Microservices**

I. Introduction

With the continuous development of cloud-native and microservice architectures, hybrid cloud is becoming increasingly popular [1]. Hybrid cloud platform composes of multiple cloud providers, either private, public or a mix of them, be seamlessly federated as if it was a single cloud platform. In a hybrid cloud, each cloud can be managed independently in its own cloud, or used as a whole depending on the setting. The data and running applications can be shared among the clouds [2]. Hybrid cloud can prevent vendor lock-in by distributing workloads to different stand-alone cloud service providers. If one service on a private cloud fails, the workload could failover to a replica or a backup server running in the other public cloud. Hybrid clouds also allow enterprises to leverage the incremental computing resources provided by various cloud providers to deal with surges of workloads.

To better adapt to the cloud environment, more and more applications are shifting from large monolithic service architectures to a large number of loosely coupled microservice architectures [3]. Hybrid cloud microservices architectures can now be easily built with container orchestration platforms such as Kubernetes [4]. In contrast to monolithic systems that need to build, deploy, and run all components at once, microservice systems are often decoupled into multiple independent components and deployed as Kubernetes resources in hybrid cloud environments.

Although microservice architecture provides us many benefits in a hybrid cloud environment, auto-scaling resources within or across hybrid clouds is still difficult, as illustrated in Fig. 1. The auto-scaling in a hybrid cloud environment presents the following challenges:

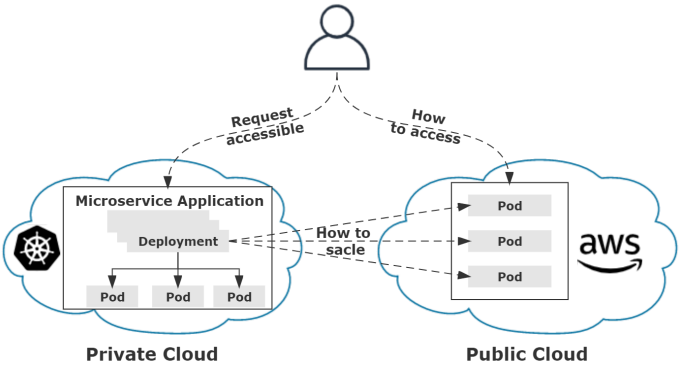


Fig. 1: With the workload in the private cloud suddenly increasing and the compute resource being exhausted, the Pod instances within Deployment will be auto-scaled to the public cloud, e.g., scaling three Pod replicas. Two major questions remain for hybrid-cloud auto-scaling. First, how to scale Pods to public clouds? Second, how to discover the new Pods in another cloud and route workload there?

- **Scaling Kubernetes resources** Scaling Kubernetes resources from one cloud to another entails more than the application itself, denoted as the Pods in Fig.1, but also nearly a copy of all other resources, e.g. the configuration resource, secrets, jobs, custom resource definitions, denoted by ConfigMap, Secret, Job and CRD respectively. How to extend these resources when scaling in another cloud challenges the hybrid cloud deployment.
- **Load balance across clouds** User requests usually target the services within one cloud. Hybrid-cloud deployment incurs a new challenge as to the global load balancing across multiple clouds and mechanisms to distribute the user requests efficiently and fairly.
- **Storage of stateful applications** Different cloud providers may use different storage classes. However, claiming the persistent volume (PV) by creating persistent volume claims (PVC) across clouds in order to efficiently deploy stateful applications also has a significant amount of challenges.

Kubernetes has in-built Horizontal Pod Autoscaler (HPA) that adjusts the replicas of the running Pods based on the user-

defined metrics, e.g. CPU or memory usage, in order to scale in need. Also it has Vertical Pod Autoscaler (VPA) that enables automatic CPU and memory request and limit adjustments based on historical resource usage measurements. We design Hybrid Cloud Autoscaler Operator (HCA Operator), a custom Kubernetes controller that uses Kubernetes CR to monitor and auto-scale microservice applications specifically in a hybrid cloud environment. HCA Operator has the following two components:

1) Admission Controller. It is used to inject a sidecar container into a microservice Pod. The injection process is not exposed to the microservice application.
2) Scheduler. It will monitor the status of the cloud cluster and create, update and delete various Kubernetes in-built and custom resources for auto-scaling across the hybrid cloud when appropriate.

We build a hybrid-cloud system with HCA Operator, Load Balancer, Monitor and Autoscaler for validation. In addition, we evaluate our Operator's feasibility and performance in a real case study. Experiments show that our HCA Operator not only scales microservice applications from private to different public clouds and vice versa, but also plays the role of automating the autoscaling itself.

This paper is structured as follows. Section II presents some related work. Section III presents the background related to Kubernetes and Kubernetes Operator. Section IV first introduces the general architecture of the hybrid-cloud system, then briefly describes the LoadBalancer, Monitor and Autoscaler in it, followed by details of the HCA Operator implementation. Section V presents some experiments to evaluate the capabilities of our Operator. Finally, Section 6 draws some conclusions of this paper.

II. Related Work

Various vendors including Dell, IBM, and HP offer hybrid cloud solutions [5], [6]. Several large companies are offering hybrid cloud solutions, often in conjunction with their existing product lines. Alibaba Cloud also offers a hybrid cloud solution called Apsara Stack, which provides an interface to manage the entire hybrid cloud platform [7].

In [8], the authors present a solution to create a hybrid cloud platform using Apache Mesos that weaves together heterogeneous clouds and geographical locations into a unified platform. Hélène Coullon et al. [9] also propose a specific constraint for hybrid Clouds. These new elements have been added within OptiPlace [10], which is an independent flexible framework to work on placement issues for data centers in hybrid cloud.

Fengtao Huang et al. [11] propose a comprehensive application deployment method based on hybrid cloud, which has been practiced in some server products. However, some issues remain unresolved, such as auto-scaling between private and public clouds, accessing databases, etc.

Chuanqi Kan [12] designs DoCloud, an elastic cloud platform that dynamically scales in or scales out the number of running containers according to the workload demand. In [13], the authors design an auto-scaling system for web applications in hybrid cloud based on Docker, that dynamically scales Docker containers based on workload demand. A scaling algorithm is developed that spawns new containers and attaches them to the load balancer. An underlying predictive model in the algorithm ensures to scale down the number of containers only if it predicts less number of containers for k successive periods.

Different from all the above work, we design a Kubernetes Operator called HCA Operator to extend the auto-scaling capability across hybrid clouds. As far as we know, no prior research or open-source tooling serves the purpose of auto-scaling across hybrid clouds. It also provides enterprises and end users with a simple, pluggable and performant solution.

III. Background

In this section, we provide an concept overview of the cloud orchestration framework Kubernetes in Subsection A and describe the Kubernetes Operator in Subsection B.

A. Kubernetes

Kubernetes is a portable, scalable, open source platform for managing containerized workloads and services at scale. In short, Kubernetes is all about managing resources - endpoints in the Kubernetes API that store a collection of API objects of different types. It also allows cluster users to define their own API custom resources (CRs) [14] by dynamically registering custom resource definition (CRD) [15]. Once a custom resource is installed, users can create and access its objects using kubectl, similarly to the in-built resource. Here is a list of Kubernetes resources that we will be using:

- **Pod** A group of one or more containers. Each container runs the applications that serves the users request. Therefore a Pod is the basic scheduling unit in a Kubernetes cluster.
- **Deployment** It maintains a fixed number of pods, enables rollout of updated Pod in a controlled manner, or roll back to an earlier deployment version if necessary.
- **Namespace** Divide a Kubernetes cluster into several virtual sub-clusters. It is useful when different teams or projects share a Kubernetes cluster when each team usually creates a seperate Namespace.
- **Configuration map (ConfigMap)** Store non-confidential data as key-value pairs that will be used in the Pods, e.g., the environment variables, parameters, etc.
- **Volume** An abstraction that persist the data created and used by Pods. This is necessary since Pods are ephemeral, meaning that data is deleted when the Pods themselves are terminated.
- **MutatingAdmissionWebhook [16]** An admission controller which intercepts Kubernetes API requests like an interceptor to modify the target resource, e.g., an incoming request may target Pods in cluster A, MutatingAdmissionWebhook can modify the request to land in cluster B in a hybrid cloud.

- **ServiceMonitor [17]** It collects various metrics by using label selectors to define the target services to monitor, the Namespaces to look for, and the port on which the metrics are exposed.
- **HorizontalPodAutoscaler (HPA) [18]** It automatically updates a workload resource (such as a Deployment), with the aim of scaling in and out the Pod replicas to serve users requests.
- **Application [19]** It can be composed by multiple components with attachable operational behaviors (traits), deployment policy and workflow. It is a higher level abstraction to a full application deployment.

B. Kubernetes Operators

In Kubernetes, controllers of the control plane implement control loops that repeatedly compare the desired state of the cluster to its actual state and reconcile any discrepancies. For example, when a Pod fails, the controller not only creates a new Pod, but also takes care of any prerequisite steps that need to be run, such as injecting a container into the newly created Pod, which may not be possible with the default reconciliation provided by Kubernetes.

A Kubernetes Operator is a software extension to allow for the creation of a custom controller that uses CRs to manage applications and their components [20]. To put it in a simple way, a CRD is equivalent to a user-defined class in object-oriented programming, e.g. a Person class with variables such as Name, Age, Height, Weight, etc. A CR is an instance of such a class, e.g. "Jack Lee", 28, 178cm, 180lb. In a Kubernetes cluster, user-defined CRD and CR can be used to create various resources. For example, if a website contains the front-end and back-end component. One can define their CRDs in the cluster first, and the Kubernetes Operator will create the website once it sees the follow-up CRs corresponding to the CRDs are created.

IV. System Design

As illustrated in Fig.2, the hybrid-cloud system consists of the HCA Operator, Load Balancer, Monitor and Autoscaler. First, the microservice application is deployed as Kubernetes resources, e.g. Deployment in the private cloud. It generates metrics for monitoring. The Load Balancer routes user requests to the private cloud. The Monitor is responsible for collecting runtime metrics from the Kubernetes Deployment in the private cloud. The runtime metrics are mainly conditions of the cloud and microservice system that affect performance and quality of service that reflect performance. These metrics are provided to the Autoscaler as a data source for auto-scaling within the private cloud. Meanwhile, the HCA Operator also collects cluster information within the private cloud and calls the Autoscaler to scale microservice resources, e.g. Pods from the private cloud to a specified public cloud when compute resources are about to be exhausted. When a Pod is auto-scaled to the public cloud, HCA Operator will apply the admission control so that user requests will eventually land to the new Pods.

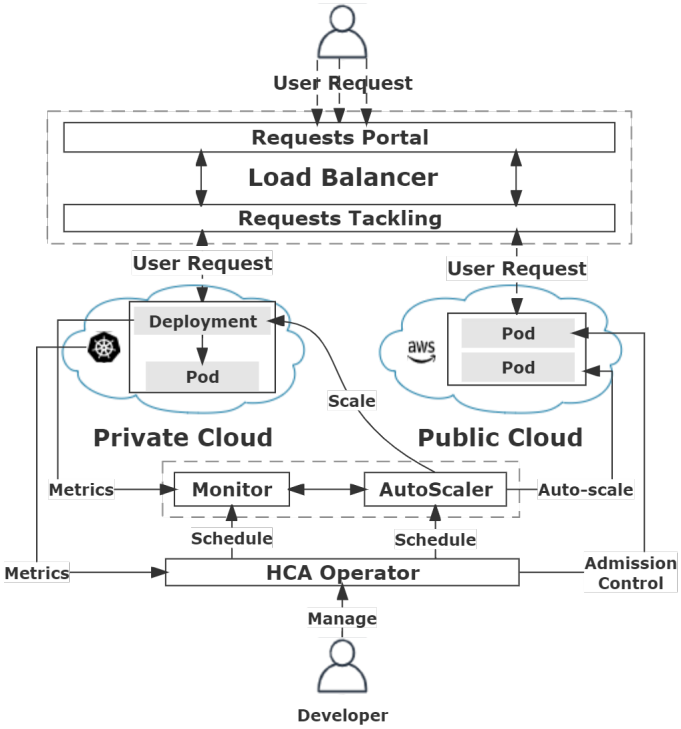


Fig. 2: System Overview.

A. Load Balancer

Load Balancer serves as the Gateway of the hybrid-cloud system. It receives all users' requests, routes them to Pods which have installed the microservice applications, and then sends back the responses to users.

We use Nginx as a load balancer due to its good performance and stability [21]. We also use the Consul [22] to dynamically update the configuration of the load balancer when a Pod is auto-scaled to the public cloud. The overall structure of the load balancer is shown in Fig.3.

Since the responsibility of the main container in a microservice Pod is to handle user requests, we design a sidecar container [23] named confSender. It is part of the microservice Pod, but is only used to communicate with the Consul service.

B. Monitor

The Monitor collects dynamic microservice runtime metrics, e.g. service requests per second (RPS), service request counts, and average, minimum and maximum response times for external user requests. These metrics are collected both on the private and the public cloud. Service RPS is averaged over a specified time period. The average, minimum and maximum response times for external user requests are calculated over any specified time period.

The Monitor in our hybrid-cloud system is implemented by Prometheus Operator [24]. Prometheus Operator is designed to orchestrate the deployment and management of Prometheus instances over a Kubernetes cluster. It provides essential functionality: a packaged set of jobs and exporter called

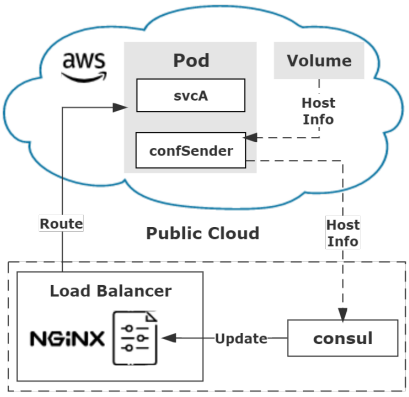


Fig. 3: Routing requests to a public cloud. The sidecar container named confSender will read the public cloud's configuration information, e.g. IP address from the Volume. It will then send the information to Consul. Consul receives the information and dynamically updates a Load Balancer Nginx's configuration file. Finally, Nginx will route to Pods in the public cloud

ServiceMonitor to gather data from the Kubernetes cluster, including per Node and Pod system metrics, i.e. CPU, memory usage, network, and Kubernetes custom metrics, i.e. the RPS, http requests, etc.

In this way, Prometheus can pull data through the metrics data interface provided by the ServiceMonitor. RPS, request counts and external user request response times are collected within the specified time period due to the capture interval setting of Prometheus. In our experiments, the period is set to be five seconds.

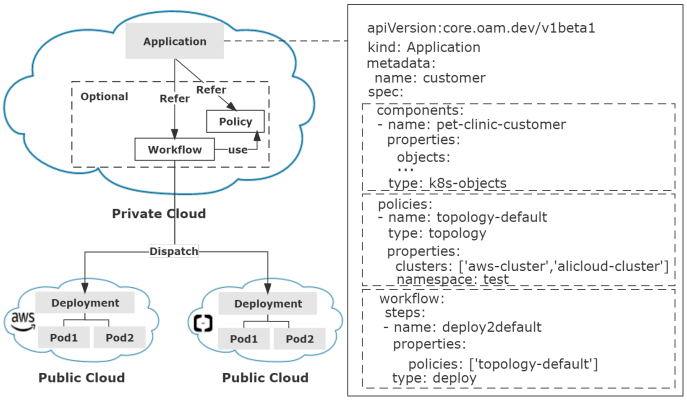


Fig. 4: KubeVela-specific configuration information (including Applications, Policies, and Workflows) live in the private cloud. The Kubernetes resources (like Pod, Deployment) will be dispatched to public clouds.

C. AutoScaler

The Autoscaler in our hybrid-cloud system is composed of two components: Kubernetes HPA and KubeVela [25].

- **Kubernetes HPA** This component scales the number of Pods of a microservice Deployment deployed in the private cloud based on the utilization of compute resources. It does not scale Pods to the other clouds. HPA receives user defined parameters related to resource utilization thresholds. To be able to utilize HPA, resource requests and limits should be allocated to the containers running in the Pods.
- **KubeVela** It is used as a control plane to deploy and operate applications across the hybrid cloud platforms. We use KubeVela's custom resource called Application to publish the microservice application to the public cloud cluster. Each Application deployment consists of three entities: Components, Policies, and Workflow. As shown in Fig.4, We apply Deployment to the different public clouds using Components and set the type of Components to k8s-objects. This type aims at deploying the Kubernetes in-built resource objects. We also define a Policy, topology shown in the yaml file shown in Fig.4. The topology tells KubeVela which cloud cluster we need to deploy the Components to. The Workflow is the real executor of the application delivery. The deployment steps in the Workflow will configure the Kubernetes resources differently based on the referenced Policy and then deploy them to the appropriate cloud clusters. Therefore, we use Workflow to target the public cloud platform, aws-cluster and alicloud-cluster.

D. HCA Operator

Fig.5 shows the structural view of the control and management flow within the HCA Operator. The HCA Operator auto-scales different microservice applications to designated cloud clusters. It is divided into two components: Admission Controller and Scheduler. For each microservice application, the Scheduler creates, deletes and updates the ServiceMonitor, HPA and Application resources associated with microservice. The Admission Controller performs admission control for each microservice application. The Scheduler and Admission Controller implementations target to the specified microservices and the cloud clusters that need to auto-scale to. Therefore, in order to support a new resource type, the only need would be implementing these two components (based on very simple rules imposed by the Operator). Our Operator is controlled by a CR named HCAJob, which defines the parameters for the execution details. The parameters include:

- The label of the microservice Pods which will be controlled by the Admission Controller (Line 7 and 8).
- The interval (s) of the HCA Operator to query the private cloud cluster state (Line 9).
- The name of the private and public cloud clusters that users want the auto-scale targets to (Line 10 and 11).
- The minimum and maximum number of replicas of Pods on the private cloud after Autoscaler auto-scaling (Line 13 and 14).
- Metrics used for auto-scaling Pods in the private cloud(Line 15-18).

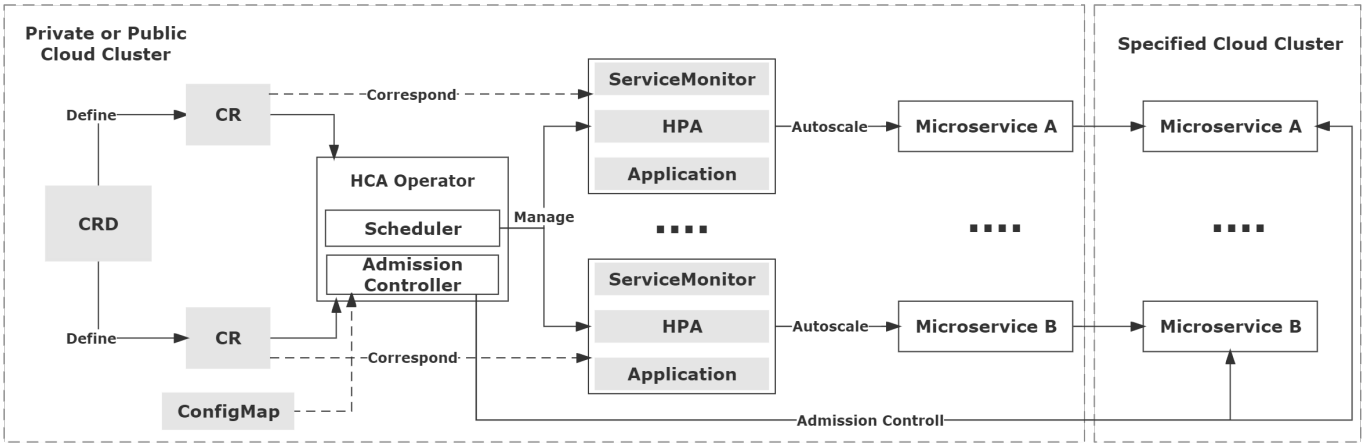


Fig. 5: A detailed view of our implementation and the relationship among the control, management and definition within the HCA Operator.

- Sub-modules designated to be monitored in the microservice system (Line 19-22).

A specific HCAJob can be deployed using a CR yaml file, see Fig.6 for example yaml.

1 Sample HCAJob yaml for auto-scaling to AWS and ACK cluster

```
1:  kind HCAJob
2:  apiVersion: HCAOperator.njtech.com/v1beta1
3:  metadata:
4:    name: hcajob-test
5:    namespace: test
6:  spec:
7:      matchLabels:
8:        app: custom
9:      updateInterval: 3
10:     clusterData:
11:       clusterName: ["AWS-cluster","ACK-cluster"]
12:     scaleData:
13:       maxReplicas: 6
14:       minReplicas: 2
15:       metrics:
16:       - type: Resource
17:         name: cpu
18:         targetAverageUtilization: 80
19:     monitorData:
20:       microserviceNamespace: test
21:     - svcLabel: custom
22:       svcPort: 8080
```

Fig. 6: An yaml example for auto-scaling to AWS and Alibaba Cloud Container Service for Kubernetes (ACK) cluster.

The detail of the two components is depicted as follows:

1) **Admission Controller** It will automatically inject a sidecar container named confSender into the microservice Pod as we mentioned in Load Balancer section. To successfully inject confSender into the target Pod, we use MutatingAdmissionWebhook to implement the Admission Controller. MutatingAdmissionWebhook is a management controller in the form of a plugin that can

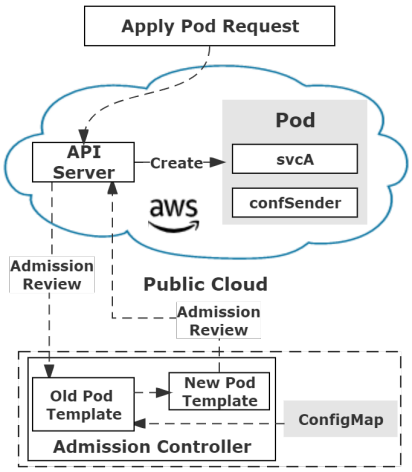


Fig. 7: The Admission Controller's workflow.

be configured based on custom requests. It will receive admission requests from the Kubernetes API Server and perform the appropriate endpoint actions based on those requests. As shown in Fig.6, we define the "matchLabels" field, whose value is a key-value pair. In this case, MutatingAdmissionWebhook will only inject confSender to the Pod with the label "app : custom". As shown in Fig.7, in the confSender auto-injection stage, the webhook follows the following steps to complete the process.

- Parse the webhook REST request and deserialize the raw data of the AdmissionReview.
- Parse the Pod and deserialize the AdmissionRequest in the AdmissionReview.
- Use the Pod to render confSender configuration template stored in the ConfigMap.
- Use the Pod and rendered template to create the Json Patch.

- Build the AdmissionResponse.
- Build the AdmissionReview and send it to the Kubernetes API Server.

2) **Scheduler** Once the CR is created, it will create the corresponding ServiceMonitor and HPA resources in turn based on the configuration provided in the CR yaml file. As shown in Fig.6, the developer can configure the "metrics", "maxReplicas" and "minReplicas" fields in the yaml file to define the metrics used by the HPA and the upper and lower limits for the number of replicas of the Pod. In addition, there are three subfields under the "monitorData" field, which are defined as follows.
   - microserviceNamespace: Namespace of the microservice system in the private cloud cluster.
   - svcLabel: The name of the sub-module in the microservice system.
   - svcPort: The port number of the sub-module in the microservice system.

2 Pseudocode for watchPod function

```
func (r *HCAJobReconciler) WatchPod(ctx context.Context, bc *bootstrap.Client) {
  for {
    // Initialize a list of all the Pods in the cloud cluster
    podsInAllNamespace := &corev1.PodList{}
    isPending := false
    // Get all Pods from API Server and add to pods list
    err := r.List(ctx, podsInAllNamespace)
    ...
    for i := 0; i < len(podsInAllNamespace.Items); i++ {
      // Judge whether the Pod is in the namespace of the microservice
      if podsInAllNamespace.Items[i].Namespace == r.Spec.MicroserviceNamespace
        // Judge whether the status of the Pod is "Pending" or not
        if podsInAllNamespace.Items[i].Status.Phase == "Pending" {
          ...
          // Create an Application resource object in Kubernetes cluster
          err = bc.CreateResource(applicationName, application)
          // Set the isPending to "true"
          isPending = true
          break
          }
    }
    //If none of the pods are in the Pending state
    //There is no need to auto-scale the microservice to other clouds
    if isPending == false{
      ...
      // Delete an Application resource object in Kubernetes cluster
      err = bc.DeleteResource(applicationName, application)
    }
    //Set the query interval
    time.Sleep(r.Spec.UpdateInterval * time.Second)
  }
}
```

Fig. 8: Pseudocode for WatchPod function.

Once the resource creation job has been executed, the CR status will reflect the status of the job, i.e. Watching/Failed/Unknown. If the resources are created successfully, the CR status will be set to Watching. As shown in Fig.8, the Scheduler then asynchronously calls a function named WatchPod. Note that a variable named isPending is defined in this function. It is used to judge whether there is a Pod in the namespace of the microservice. If the value of isPending is still "false" after traversing the list of all Pods in the private cloud cluster, it either represents that there are still available computing resources in the private cloud cluster, so there is no need to scale out the microservice to the public cloud, or the workload on the private cloud has decreased after the auto-scaling process across the hybrid cloud, and there are free compute resources. When the microservice is scaled to the public cloud, the status of the CR changes from Watching to Finished, and when the microservice is scaled down from the public cloud to the private cloud, the status of the CR will change from Finished back to Watching.

V. EXPERIMENTAL EVALUATION

A. Testbed Setup

Our private cloud testbed was built on a cluster of four nodes. Each node has a Intel Core i7-12700 Processor and 32 GB RAM. On each node we installed Centos 7 with Linux kernel version 5.4.1. We deployed a Kubernetes platform with version 1.20.1 on these machines, one served as master and the others as workers. Our public cloud testbed used Amazon Elastic Kubernetes Service (EKS). The EKS platform is a three-node Kubernetes cluster of version 1.21.12. The cluster has two worker nodes and a master node. The specification of each node server is 4 vCPU and 16 GB RAM.

To evaluate the HCA Operator, we used Spring PetClinic Microservice benchmark [26] on our testbed. It is implemented as five modules. Among these modules, there are three stateful microservice modules which are Java websites. We have experimented only on these three stateful modules. In our experiments, the stateful microservice modules are three separate Kubernetes Service and either may consist of several Pod instances. The three stateful modules are the targets of our hybrid-cloud system shown in Section IV. As shown in Table I, we set up the label, CPU and memory of their Pods.

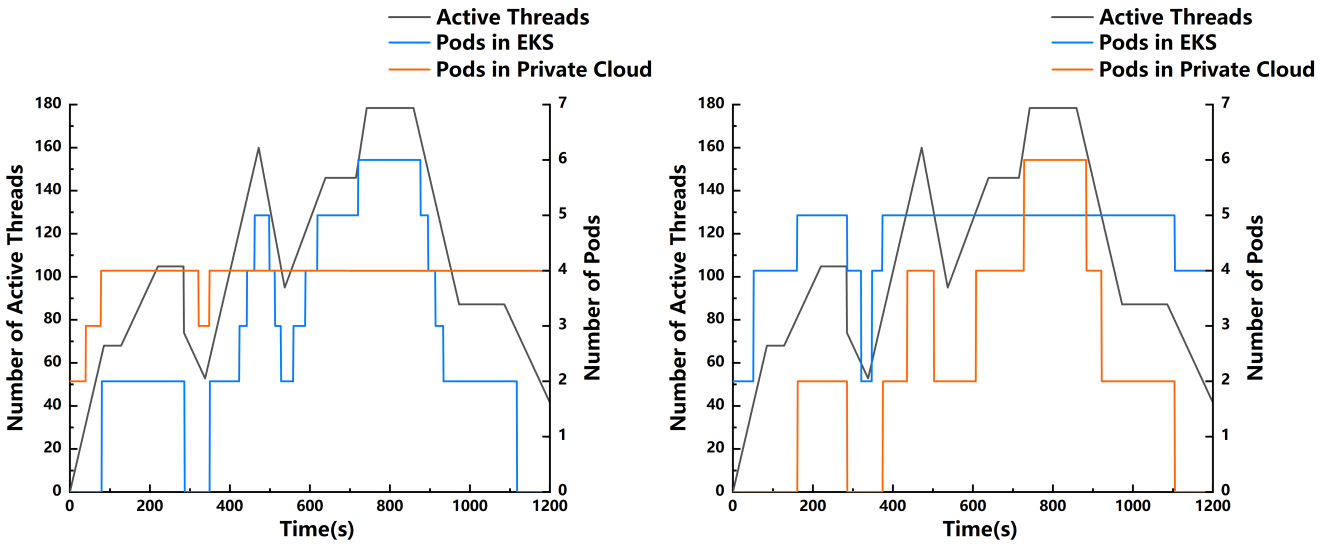
TABLE I: An example of the Spring PetClinic Pod and all parameter settings

Parameter	Value
matchLabels	app: custom
resources.requests.cpu	0.5
resources.requests.memory	512M
resources.limits.cpu	1
resources.limits.memory	1024M

In order to make the experimental results more accurate, we repeated each set of experiments three times. The final experimental result is the average of the three experimental results.

B. Performance Under Different Auto-scaling Scenarios

To validate the feasibility of the HCA Operator, we designed two auto-scaling scenarios. The first scenario deploys the Spring PetClinic microservice on a private cloud and we set Operator to auto-scale based on CPU utilization. In this case, we set the scaling CPU threshold to 70% and we use the Ultimate Thread Group component of Apache Jmeter [27] to generate a real workload on our private cloud. The other scenario deploys Spring PetClinic microservice on our public cloud and all other parts are the same as in scenario one. We adjusted the number of threads in the component to be appropriate for our experiments. As shown in Fig.9(a) and



(a) Deployed in private cloud.

(b) Deployed in Amazon EKS.

Fig. 9: The number of Pods in hybrid cloud vs changing load

Fig.9(b), our Operator not only auto-scales microservice across hybrid clouds, but also handle real workloads including flash crowds.

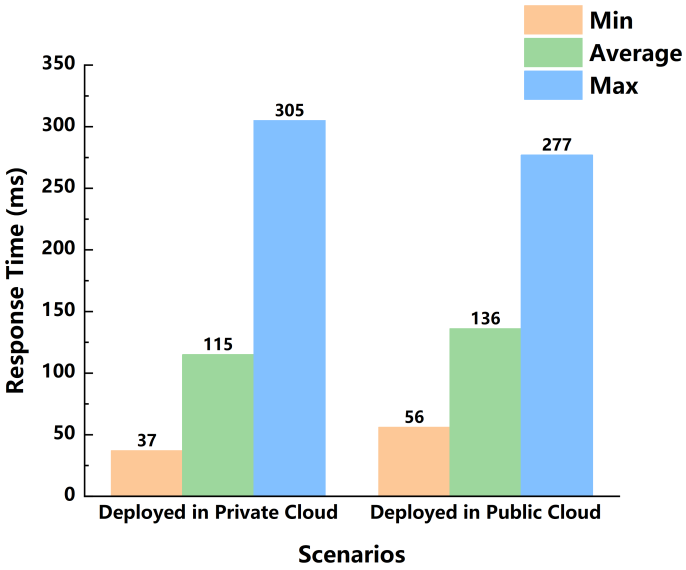


Fig. 10: Response time of user requests.

We focus on the response time of user requests since it is one of the most important performance metrics for evaluating web services. We also set the Service Level Agreement (SLA) to test our Operator and hybrid cloud system. We set the SLA value to 200ms, meaning that we expect user requests to be responded within 200ms. While minimizing SLA violations is our first goal, we also aim to avoid resource over-provisioning to save cost. Fig.10 shows the average, minimum and maximum response times for Spring PetClinic deployed in our private and public clouds. As shown in Fig.10, we can see that when the HCA Operator runs on CPU usage, the average response time under scenario I and II only differs by about 10ms and both of them meet the SLA requirement.

TABLE II: Comparison of the SLA violation rate, total delay and exception rate of deploying the Spring PetClinic in our private cloud and public cloud.

Scenarios	SLA violations(%)	Total delay(ms)	Exception rate(%)
Deployed in Private Cloud	1.24	32.41	0
Deployed in Public Cloud	0.98	49.53	0

We calculated the SLA violation rate, total latency of Spring PetClinic under both scenarios using data provided by Monitor in the hybrid cloud system. The results are shown in TableII. The total delay represents the time from the start of our Operator's admission control to the microservice Pod, and further to the end of the Load Balancer's route, and finally to the microservice Pod. The exception rate represents the percentage of the number of error requests that occur during the experiment. We conclude from the Table II that in both scenarios, although Spring PetClinic violates the SLA, most of the response times meet the SLA requirement. Meanwhile, the total delay is also maintained at a low value and all user requests don't have exceptions, which means that our Operator has the ability to respond quickly when auto-scaling microservice applications, which is very important when the workload is unstable.

VI. Conclusion

Cloud resource provisioning has been one of the hottest research topic both in academia and industry. Among them, autoscaling compute resource in hybrid cloud, where public and private are managed as a whole presents a big challenge.

In this paper, we design HCA Operator that leverages the Kubernetes API and manages the provisioning in hybrid clouds. It is a custom Kubernetes controller that uses the Kubernetes custom resources to automatically scale microservice applications in hybrid clouds. It contains two components: Admission Controller and Scheduler. The Admission Controller automatically injects a sidecar container called confSender into a microservice Pod, managing the global load balancing across the hybrid cloud. The Scheduler automatically scales microservice applications to destination clouds by managing three Kubernetes resources and monitoring cluster status. We implement the HCA Operator in a hybrid cloud environment and evaluated the capabilities of its autoscaling under real workloads. The experimental results show that the HCA Operator manages to properly auto-scale microservice applications in a hybrid cloud environment and manages a reasonable SLA requirements.

REFERENCES

[1] S. Newman, *Building microservices*. " O'Reilly Media, Inc.", 2021.

[2] A. Srinivasan, M. A. Quadir, and V. Vijayakumar, "Era of cloud computing: A new insight to hybrid cloud," *Procedia Computer Science*, vol. 50, pp. 42–51, 2015.

[3] M. Villamizar, O. Garces, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *Computing Colombian Conference*, 2015, pp. 583–590.

[4] C. C. Chang, S. R. Yang, E. H. Yeh, P. Lin, and J. Y. Jeng, "A kubernetes-based monitoring platform for dynamic cloud resource provisioning," in *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, 2018.

[5] T. R. Connor and J. Southgate, "Automated cloud brokerage based upon continuous real-time benchmarking," in *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 2015, pp. 372–375.

[6] G. Breiter and V. K. Naik, "A framework for controlling and managing hybrid cloud service integration," in *2013 IEEE international conference on Cloud engineering (ic2e)*. IEEE, 2013, pp. 217–224.

[7] "Alibaba cloud hybrid cloud apsara stack," [Online], https://www.aliyun.com/solution/hybridcloud.

[8] N. Xue, H. Haugerud, and A. Yazidi, "On automated cloud bursting and hybrid cloud setups using apache mesos," in *2017 3rd International Conference of Cloud Computing Technologies and Applications (CloudTech)*. IEEE, 2017, pp. 1–8.

[9] C. Helene, G. Le Louet, and J.-M. Menaud, "Virtual machine placement for hybrid cloud using constraint programming," in *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2017, pp. 326–333.

[10] G. Le Louët and J.-M. Menaud, "Optiplace: Designing cloud management with flexible power models through constraint programing," in *2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*. IEEE, 2013, pp. 211–218.

[11] F. Huang, H. Li, Z. Yuan, and X. Li, "An application deployment approach based on hybrid cloud," in *2017 ieee 3rd international conference on big data security on cloud (bigdatasecurity), ieee international conference on high performance and smart computing (hpsc), and ieee international conference on intelligent data and security (ids)*. IEEE, 2017, pp. 74–79.

[12] C. Kan, "Docloud: An elastic cloud platform for web applications based on docker," in *2016 18th international conference on advanced communication technology (ICACT)*. IEEE, 2016, pp. 478–483.

[13] Y. Li and Y. Xia, "Auto-scaling web applications in hybrid cloud based on docker," in *2016 5th International Conference on Computer Science and Network Technology (ICCSNT)*, 2016.

[14] "Custom resource," [Online], https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/#custom-resources.

[15] "Customresourcedefinitions," [Online], https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/#customresourcedefinitions.

[16] "Experimenting with admission webhooks," [Online], https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/.

[17] U. Altaf, G. Jayaputera, J. Li, D. Marques, D. Meggyesy, S. Sarwar, S. Sharma, W. Voorsluys, R. Sinnott, A. Novak *et al.*, "Auto-scaling a defence application across the cloud using docker and kubernetes," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE, 2018, pp. 327–334.

[18] "Kubernetes horizontal pod autoscaler," [Online], https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[19] "Application," [Online], https://kubevela.net/docs/getting-started/core-concept.

[20] S. Bhagavan, S. Balasubramanian, P. R. Annem, T. Ngo, and A. Soundararaj, "Achieving operational scalability using razee continuous deployment model and kubernetes operators," *arXiv preprint arXiv:2012.10526*, 2020.

[21] W. Reese, "Nginx: The high-performance web server and reverse proxy," *Linux Journal*, 2008.

[22] "Load balancing with nginx and consul template," [Online], https://learn.hashicorp.com/tutorials/consul/load-balancing-nginx?in=consul/load-balancing.

[23] B. Burns and D. Oppenheimer, "Design patterns for container-based distributed systems," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, CO: USENIX Association, Jun. 2016. [Online]. Available: https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/burns

[24] "Prometheus operator," [Online], https://github.com/prometheus-operator/prometheus-operator.

[25] Y. Wang, C. Lee, S. Ren, E. Kim, and S. Chung, "Enabling role-based orchestration for cloud applications," *Applied Sciences*, vol. 11, no. 14, p. 6656, 2021.

[26] "Spring petclinic sample application," [Online], https://github.com/spring-projects/spring-petclinic.

[27] "Apache jmeter - apache jmeter™," [Online], https://jmeter.apache.org/.