A Case Study on Five Maturity Levels of A Kubernetes Operator

Ruxiao Duan
Department of Computer Science
The University of Hong Kong
Hong Kong, China
drx0411_sz@126.com

Fan Zhang
IBM Data and AI
Littleton
MA 01460
fzhang@us.ibm.com

Samee U. Khan
Department of Electrical and Computer Engineering
Mississippi State University
MS 39762
skhan@ece.msstate.edu

2021 IEEE Cloud Summit (Cloud Summit) | 978-1-6654-2582-7/21/$31.00 ©2021 IEEE | DOI: 10.1109/IEEECloudSummit52029.2021.00008

***Abstract*—Deploying distributed applications using their Operators in a containerized platform on the state-of-art cloud orchestration tooling, such as Kubernetes, has truly become widely accepted. However, the quality of an Operator has a significant impact on a few core metrics of the application, such as its availability, consistency, and quality of service. This paper introduces the Kubernetes Operator maturity model and its five maturity levels, and then gives a demonstration on how a demo Kubernetes Operator is capable of reaching all the five levels respectively by using an example Operator named New Visitors Site Operator. Finally, an experiment illustrating the capability of the example Operator's auto-scaling functions to improve the application performance is presented. This example Operator will enable developers and researchers to design containerized applications with more enhanced features. The code is available at https://github.com/ringdrx/visitors-operator.**

***Index Terms*—Kubernetes, Operator, maturity, auto-scaling**

I. Introduction

In recent years, distributed computing and cloud computing are coming into fashion and are increasingly more dependent on the technologies of lightweight application containerization. Kubernetes is the SOTA popular open-source software for container scheduling and orchestration, and Kubernetes Operators are extensions of Kubernetes to deploy, scale and manage container-based applications using custom resources.

Kubernetes Operators have five levels of maturity as indicated in the Operator maturity model [1], and different operators, depending on their capabilities to manage the application lifecycle, are able to achieve a portion or all of the five levels to manage the full lifecycle of the containerized applications, e.g., configuring, installing, upgrading, health-checking, scaling, recovering, scheduling applications as well as backing up and restoring data.

Developing Kubernetes Operators that can easily control the installations and upgrades, complete the full application lifecycle, and implement advanced functions like auto-scaling can be extremely challenging, especially for large-scale application systems. Therefore, understanding the capabilities of Operators in each of the five maturity levels and the ways for implementation should be important for Operator developers. However, up to now, there is still no Operator managing a real-case application that demonstrates the realization of five maturity levels, so our contribution is the first of its kind as a case study to develop such a full-fledged example Operator.

This paper aims to provide developers with an example Operator called New Visitors Site Operator that can reach all the five maturity levels by implementing their core functions respectively, and hopefully, this demonstration might reduce the work complexity for other developers and researchers and give them a foundation to extend powerful Operators with more enhanced features.

II. Related Work

Developers of Kubernetes Operators have produced many Operators with advanced features, especially some powerful auto-scaling algorithms that can dramatically improve the performance of applications. All those works with auto-scaling functions have reached the highest level, namely level 5 of the Operator maturity model.

Experiments have demonstrated the operational behaviors of Kubernetes' auto-scaling mechanism Horizontal Pod Autoscaler (HPA). Some of the results offer us insight into how to make auto-scaling more effective. For example, choosing a suitable kind of custom metrics or a combination of them based on the type of application might significantly increase the effectiveness of HPA [2].

Reference [3] has developed an adaptive autoscaler named Libra, which is able to perform horizontal scaling after finding the optimal resource set for a pod. The process might be adjusted automatically after detecting changes from the load or the underlying virtualized environment [3].

Another study [4] also indicates that machine learning algorithms can have a positive effect on the construction of an Intelligent Operator, a kind of Operator that aims to optimize the deployed system configuration by taking into account the observed metrics and tested configurations. Classifiers of decision trees including Gini index, entropy, PART, and Incremental Pruning to Produce Error Reduction (RIPPER), all contribute to the generation of explainable decision models, which enhance the performance of Operators by providing a better way to maintain a healthy and efficient runtime configuration [4].

978-1-6654-2582-7/21/$31.00 ©2021 IEEE
DOI 10.1109/IEEECloudSummit52029.2021.00008

Authorized licensed use limited to: Nanjing University. Downloaded on February 24,2022 at 03:45:09 UTC from IEEE Xplore. Restrictions apply.

III. Demonstration

In the Operator maturity model [5], the five maturity levels for Kubernetes Operators are defined as following (from level 1 to level 5 respectively):

- 1. Basic Install
- 2. Seamless Upgrades
- 3. Full Lifecycle
- 4. Deep Insights
- 5. Auto Pilot

An Operator that possesses the core functions at a level is said to have reached that particular maturity level. Fig. 1 illustrates all these five levels and the core functions that each level includes. Operators developed using the Go programming language are capable of reaching all five levels [5], thus in our demonstration, a Go-based Operator is constructed using Operator SDK.

The sample application used in this paper, called Visitors Site Application, comes from the book [6]. This is a simple web application that has only one web page, which displays nothing more than an app title and a table containing records of the recent visitors to the homepage, as is shown in Fig. 2. The three components of the application are illustrated in Fig. 3.

In the book [6], an Operator called Visitors Site Operator is built for managing this application. Some differences exist between this Operator and the Operator that will be introduced in this paper. The original one is constructed using Operator SDK pre-v1.0.0 version, which has already been outdated by now. However, in our example, the newest version of Operator SDK (v1.10.0) is applied. The Visitors Site Operator in [6] can only realize Operator maturity level 1 and level 2, but our New Visitors Site Operator is able to implement functions of all five levels.

In the following subsections, we are going to see how the functions of the New Visitors Site Operator enable it to reach all the five Operator maturity levels respectively.

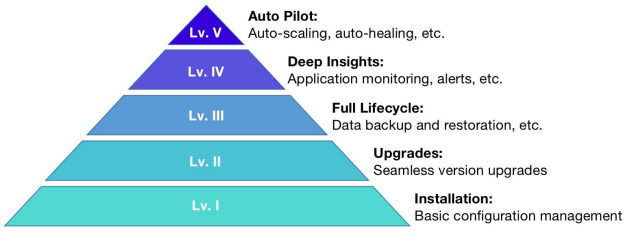

Fig. 1. The Operator maturity model [5].

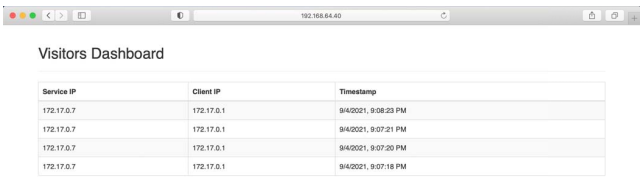

Fig. 2. Each time the homepage is visited, information about the visit is stored in the database, after which the table will update its content [6].

Fig. 3. The three application components, i.e., frontend, backend, and database, are developed using React, Django, and MySQL respectively [6].

A. Basic Install

This level is the most basic level in the Operator maturity model. Users should be able to install and configure the workload as they desire using a Custom Resource. A Custom Resource (CR) is a Kubernetes API's extension that stands for a customization of a specific installation. The Operator has a Kubernetes API resource called Custom Resource Definition (CRD) to define the content of the CR. After a CR is applied, the reconcile function in the Operator's controller is triggered to match the current cluster state to the desired state indicated in the spec section of the CR.

Installation of the New Visitors Site Operator is straightforward, thanks to the built-in functions of Operator SDK. Users can choose to run the Operator locally outside the Kubernetes cluster, or run it as a Deployment inside the cluster. After executing the command, resources like a VisitorsApp CRD and ConfigMap can all be successfully installed. Since the database of the application needs to be set up using an external MySQL Operator [7], we can install it together with a MysqlClustser CRD using Helm [8], a package manager for Kubernetes.

Then, a VisitorsApp CR with specific spec values can be created as app users wish. In our Operator, spec values including pod replicas of both backend and frontend, and the homepage's title can all be specified in the CR. The current state will be adjusted to be consistent with the specified one once the YAML file of CR is applied. Similarly, for the database, another MysqlCluster CR can be applied to create the database cluster. Fig. 4 shows how all the necessary

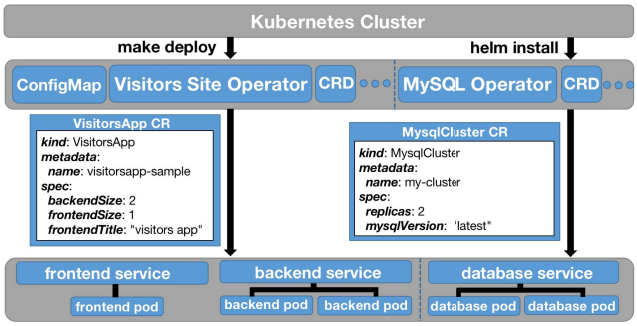

Fig. 4. One command "make deploy" (defined in Operator Makefile) to install resources such as the Deployment for the Visitors Site Operator, VisitorsApp CRD, ConfigMap, RoleBinding, etc. Another Helm command to install the external MySQL Operator. The Visitors Site Operator's controller takes action when a VisitorsApp CR is applied, creating two pods for backend and one for frontend as specified in the CR, together with the services to connect them. The database cluster is installed similarly.

Authorized licensed use limited to: Nanjing University. Downloaded on February 24,2022 at 03:45:09 UTC from IEEE Xplore. Restrictions apply.

resources for the application are installed and configured through the procedures mentioned above.

B. Seamless Upgrades

At this level, Operators should be able to upgrade the workload it manages without any data loss.

The original Visitors Site Operator in [6] is also capable of performing upgrades, but it can make changes only to the backend pod replicas and the app title. Without depending on any external database Operators, the MySQL pod number is constrained to one, and MySQL version upgrade is also not supported.

In New Visitors Site Operator, users can modify the content of CR and apply the changes. App title, service node ports, and pod replicas for both frontend and backend can all be altered seamlessly in this way. Besides, upgrading the database is also made possible in the new Operator. Since the Operator depends on an external MySQL Operator, which defines another CRD named MysqlCluster for users to create a database cluster, users can update the CR for MysqlCluster to perform a database upgrade, changing either MySQL version or database pod replicas. With those helpful functions in the MySQL Operator, upgrading the database no longer suffers from data loss, and there is also no downtime during the upgrade. Fig. 5 illustrates the process to seamlessly upgrade the MySQL version of the database cluster.

C. Full Lifecycle

Whether an Operator reaches this level can be determined by its capability to back up and restore. An Operator that completes a full lifecycle needs to be able to back up the data and, in case any data catastrophe takes place, be able to restore them from the backup.

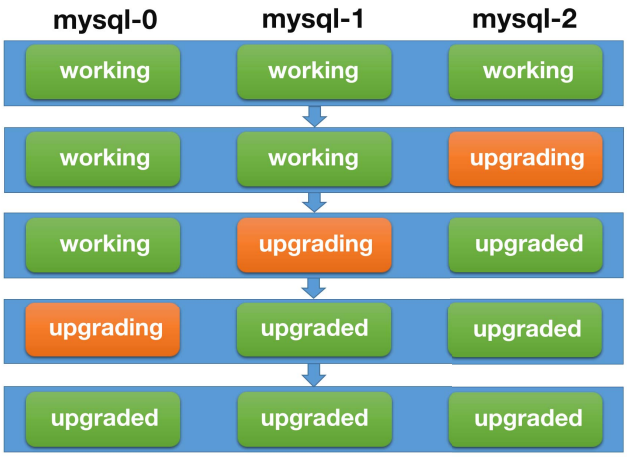

Fig. 5. This is the process to upgrade a MySQL cluster with three pods. An attribute of the cluster called "maxUnavailable" specifies the maximum number of pods that can be upgraded at the same time, and in this case, this value is set to one. Therefore, there is at most one pod upgrading and at least two pods working at any time. So the entire process does not have any downtime or data loss.

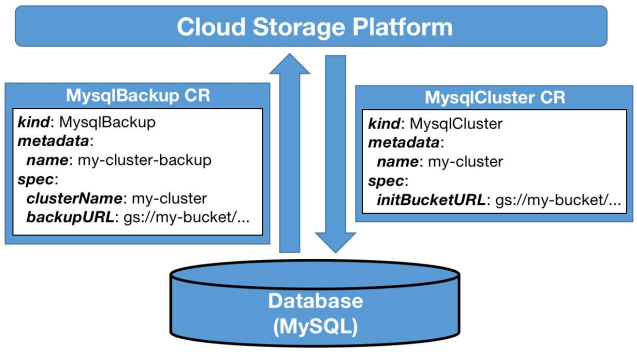

Fig. 6. Data backup and restoration only require the application of two CRs, in which the remote storage URL should be specified.

Our Operator reaches this level by taking advantage of the MySQL Operator. Functionalities of data backup and restoration are both within the capabilities of this external Operator, but a remote platform for data storage like AWS or Google Cloud Service is required.

Users can specify the backup credentials and remote backup URL in the CR, and then create a backup at remote by applying the backup YAML file (Fig. 6). Automatic backup is also possible if the schedule is configured, so backing up visitors' records every day or every hour is nothing difficult.

To restore data from a backup, the MysqlCluster CR has an attribute that can be specified as the remote path from which the database can retrieve previous data (Fig. 6). Even if the Operator runs on a different computer, the records of the original visitors can still be displayed on the homepage if data restoration from remote is successful.

D. Deep Insights

Operators at level 4 offer features like monitoring and alerting. Metrics containing the health conditions of all the application components can be exposed, and alerts are sent once any rule is violated.

In the New Visitors Site Operator, functions including monitoring and alerting can be realized with the help of Prometheus [9], an open-source toolkit for monitoring the state of the cluster and sending alerts. Besides, in order for Prometheus to understand the metrics sent by MySQL, a MySQL exporter [10] is needed to change the metrics to the proper format that is understandable by Prometheus. Fig. 7 describes the entire monitoring system of the application.

After performing port-forward for the Prometheus service, users can visit the Prometheus web page to check all the cluster components that are being monitored by Prometheus, as well as the details of all the rules and alerts.

App maintainers can check the health status of frontend, backend, and database, and are also able to establish Prometheus rules themselves by applying YAML files. Alerts will be displayed on Prometheus once any rule is broken. For example, if any pod is down due to some unexpected reason like resource shortage, maintainers are aware of that since alerts are forwarded to them via Prometheus.

Authorized licensed use limited to: Nanjing University. Downloaded on February 24,2022 at 03:45:09 UTC from IEEE Xplore. Restrictions apply.

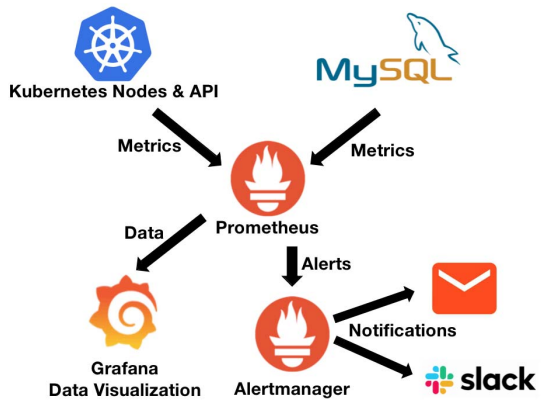

Fig. 7. Prometheus collects metrics from Kubernetes resources and MySQL database through ServiceMonitors. Data are sent to Grafana for visualization. An alert is created when any rule is broken and is sent to Alertmanager, which can further issue a notification to platforms like slack or developers' emails.

E. Auto Pilot

The final maturity level has many advanced features, among which auto-scaling is of great significance and is also essential in cloud computing. Operators with this capability should be able to scale the workload they manage depending on the metrics they receive.

There are two types of auto-scaling, known as horizontal and vertical auto-scaling. Horizontal scaling adds new nodes or pods to the cluster when the load exceeds a threshold, while vertical scaling allocates more resources like CPU and RAM to the existing nodes or pods. Additional pods, nodes, and other resources will be gradually removed once the load drops below the threshold.

In the New Visitors Site Operator, horizontal auto-scaling is implemented because of its unlimitedness and lower risk of downtime [11]. Frontend, backend, as well as database, are all targets to scale. The auto-scaling function of the Operator depends on the Horizontal Pod Autoscaler (HPA), a Kubernetes API resource that is utilized to auto-scale the number of pods of a scalable object like a Deployment or a StatefulSet based on specific information of the observed metrics, e.g., the average CPU utilization of pods (Fig. 8).

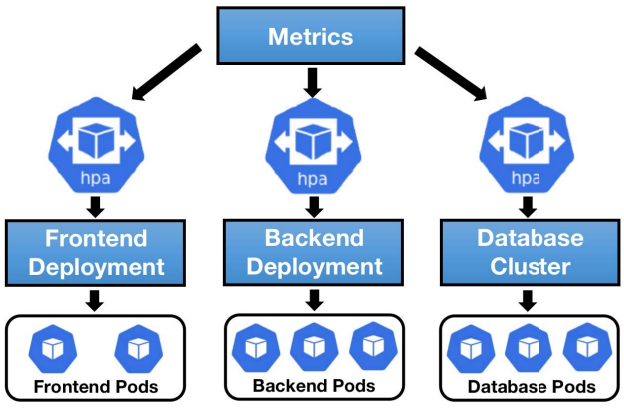

Fig. 8. When the load (average CPU utilization of pods) of the database exceeds the limit specified in HPA, some read-only slave pods will be added to the MySQL cluster, decreasing the pressure on each pod. After the average CPU utilization decreases, the additional pods will be gradually deleted. Auto-scaling for backend and frontend works in a similar way.

IV. Experiments

Two experiments are conducted to test the auto-scaling function of the Operator. We wish to verify that the HPA adopted in the Operator is actually able to improve the performance of Visitors Site Application.

A. Preparation and Setup

One of the most direct indicators of application performance is the response time, and in our case, the most important information the app users are interested in is the list of visiting records. Therefore, we measure the time it takes for the backend to retrieve data from the database and use it as a benchmark to evaluate the app performance.

Jmeter is used to simulate visitors continuously paying a visit to Visitors Site by sending requests to the backend service for historical records, and is also used to measure the response time. Applying this software, we are able to test the effect that the auto-scaling function has on app performance.

B. Results and Evaluation

In the first experiment, the load of backend (number of visitors sending requests at the same time) is set at a relatively high level, some time before the HPA for backend is applied. A reduction of response time is noticeable after the HPA is applied and generates more pods for the backend, as is illustrated in Fig. 9. The newly created pods also increase the overall stability of the app performance, which can be seen from the decrease of response time's fluctuation.

Fig. 10 demonstrates the result of the second experiment, which is designed to test the autoscaler's ability to detect the changes of load. In this experiment, the HPA is launched at the beginning, when the load is set at a low level. Since the average CPU utilization of pods does not reach the threshold, the HPA takes no action. However, after the load increases, the

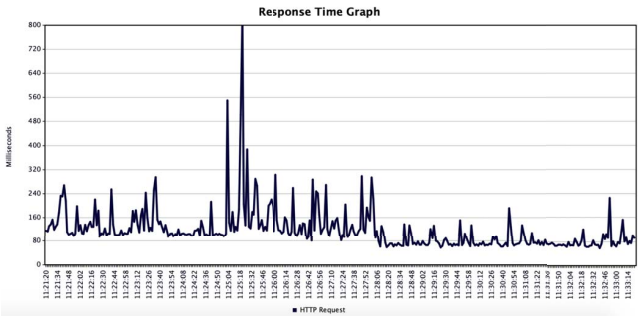

Fig. 9. Backend response time graph (HPA applied after the heavy load). The response time significantly decreases and also stabilizes after new pods are created by HPA.

Authorized licensed use limited to: Nanjing University. Downloaded on February 24,2022 at 03:45:09 UTC from IEEE Xplore. Restrictions apply.

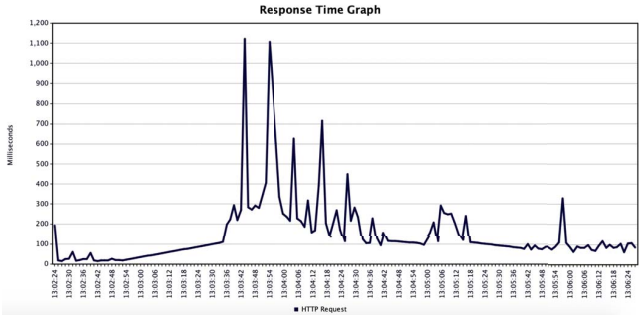

Fig. 10. Backend response time graph (heavy load exerted after HPA). The response time first increases dramatically because of the lack of resources to handle the new traffic, and then gradually drops to a normal level since HPA has created new pods for backend.

CPU utilization breaks the limit, so HPA knows that it should generate more pods to handle the overwhelming traffic. The response time gradually decreases after new pods are created one after another.

V. Discussion

A. Conclusions

This paper demonstrates how five maturity levels of the Operator maturity model can be realized respectively by using a New Visitors Site Operator, an Operator developed based on the Visitors Site Operator in the book [6]. Many improvements are made to provide this Operator with functions of all five levels. An external MySQL Operator, the MySQL exporter, and Prometheus are all exploited for easier implementation of essential functions in the last three levels, such as backup, restoration, monitoring, and auto-scaling. Finally, two simple experiments are presented to illustrate how the autoscaler in our Operator can improve the performance of the web application Visitors Site.

This demonstration should serve as a foundation for further discussion on topics including how to evaluate which maturity level an Operator has reached based on its functionalities, how to build more powerful Operators to realize more advanced features described in the Operator maturity model, and how to assess the auto-scaling capabilities of tools like Kubernetes HPA.

B. Challenges

One of the difficulties encountered when constructing this Operator lies in the connection between backend and database. Since the original Visitors Site Operator in the book assumes that there is only one MySQL pod, which is a read-write pod for the database, the backend links to the database only by a single MySQL service. However, in the new Operator, increasing the database pod replicas is made possible, so some new read-only pods might be created. Therefore, the read and write operations in the backend must be separated and linked to distinct database services, i.e., read operation should use a service that covers all database pods, and write operation should use a service that covers only the master pod (read-write pod).

Another problem is that the database cluster adopted by the MySQL Operator cannot be scaled by HPA at first. This is because MysqlCluster CRD does not have an attribute that is vital in selecting which child pods to scale. A MysqlCluster creates a MySQL StatefulSet, and that StatefulSet is responsible for scaling the database pods, so the MysqlCluster has no direct connection with all the pods owned by the StatefulSet. By updating the CRD to specify the label selector, HPA knows which pods belong to the database cluster, so that it understands how to scale the cluster. Note that the database StatefulSet can not be the target of HPA in this case because of the mechanism of the MySQL Operator.

C. Limitations and Improvements

The New Visitors Site Operator contains many new features that the original version does not have, but limitations still exist, based on which future improvements are possible.

First of all, the current version only allows the backup of data stored in MySQL database, i.e., the records of visitors. Therefore other app information like the frontend title, pod replicas, and database version is not stored in the backup, thus cannot be restored. Future versions of the Operator might include the backup of these attributes.

Secondly, the alerting system now only applies default rules offered by Prometheus, which are often too general, and no specific rules have been added. Thus Prometheus can only send alerts when one or more of those default rules are broken, e.g., when some pods are down. The Operator can be upgraded to provide rules that are more specific to the Visitors Site Application, e.g., sending an alert when the current number of visitors to the site exceeds a threshold.

Besides, the auto-scaling system of this Operator merely supports horizontal pod auto-scaling, so scaling the number of nodes and vertical auto-scaling are both beyond its current capability. Therefore, the Operator can be improved by adding these new features for auto-scaling.

Last but not least, other features of the final maturity level, such as auto-tuning and auto-healing can also be implemented in future versions. Advanced functions like shifting the workload to the least busy node, and healing unhealthy pods by taking proper actions like redeploying or restoring a backup after considering the metrics' information can be included in this Operator.

References

[1] R. Szumski, "Top Kubernetes Operators advancing across the operator capability model," *Redhat.com*. [Online]. Available: https://cloud.redhat.com/blog/top-kubernetes-operators-advancing-across-the-operator-capability-model. [Accessed: 04-Sep-2021].

[2] T. Nguyen, Y. Yeom, T. Kim, D. Park and S. Kim, "Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration", *Sensors*, vol. 20, no. 16, p. 4621, 2020. Available: 10.3390/s20164621.

[3] D. Balla, C. Simon, and M. Maliosz, "Adaptive scaling of Kubernetes pods," in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, 2020.

Authorized licensed use limited to: Nanjing University. Downloaded on February 24,2022 at 03:45:09 UTC from IEEE Xplore. Restrictions apply.

[4] S. Łaskawiec, M. Choraś, R. Kozik, and V. Varadarajan, "Intelligent operator: Machine learning based decision support and explainer for human operators and service providers in the fog, cloud and edge networks," *J. Inf. Secur. Appl.*, vol. 56, no. 102685, p. 102685, 2021.

[5] "Welcome to Operator framework", *Operatorframework.io*, 2021. [Online]. Available: https://operatorframework.io/operator-capabilities/. [Accessed: 04- Sep- 2021].

[6] J. Dobies and J. Wood, *KUBERNETES OPERATORS*, 1st ed. Sebastopol: O'REILLY MEDIA, INCORPORA, 2020.

[7] "GitHub - bitpoke/mysql-operator: Bulletproof MySQL on Kubernetes using Percona Server", *GitHub*, 2021. [Online]. Available: https://github.com/bitpoke/mysql-operator. [Accessed: 08- Sep- 2021].

[8] "Helm", *Helm.sh*, 2021. [Online]. Available: https://helm.sh/. [Accessed: 06- Sep- 2021].

[9] "Prometheus - Monitoring system & time series database", *Prometheus.io*, 2021. [Online]. Available: https://prometheus.io/docs/prometheus/latest/. [Accessed: 06- Sep- 2021].

[10] "helm-charts/charts/prometheus-mysql-exporter at main · prometheus-community/helm-charts", *GitHub*, 2021. [Online]. Available: https://github.com/prometheus-community/helm-charts/tree/main/charts/prometheus-mysql-exporter. [Accessed: 08- Sep- 2021].

[11] RedSwitches, "The difference between horizontal vs vertical scaling," *Redswitches.com*, 10-Dec-2019. [Online]. Available: https://www.redswitches.com/blog/difference-between-horizontal-vertical-scaling/. [Accessed: 06-Sep-2021].

Authorized licensed use limited to: Nanjing University. Downloaded on February 24,2022 at 03:45:09 UTC from IEEE Xplore. Restrictions apply.