Baking Disaster-Proof Kubernetes Applications with Efficient Recipes

Runyu Jin
runyu.jin@ibm.com
IBM Almaden Research Center
Almaden, California, USA

Paul Muench
pmuench@us.ibm.com
IBM Almaden Research Center
Almaden, California, USA

Travis Janssen
travis.janssen@ibm.com
IBM Almaden Research Center
Almaden, California, USA

Brian Hatfield
bhatfiel@us.ibm.com
IBM Almaden Research Center
Almaden, California, USA

Veera Deenadhayalan
veerad@us.ibm.com
IBM Almaden Research Center
Almaden, California, USA

ABSTRACT

Multicluster disaster recovery on cloud-native platforms such as Kubernetes usually replicates application data and Kubernetes resources to a safe recovery cluster. In the event of a disaster, Kubernetes resources are restored to the recovery cluster to recover the affected applications. We tested 10 popular Kubernetes applications using this naive approach, and 60% failed. Problems include data being restored in the wrong order, cluster-specific data being restored instead of generated by the cluster, etc. All these problems lead to our recipe design that enables disaster recovery of all Kubernetes applications. In this paper, we analyze the problems we encountered during the disaster recovery of Kubernetes applications and categorize applications based on their disaster recovery behaviors. We present a recipe that groups, orders, and filters Kubernetes resources to enable disaster recovery. Finally, we evaluate the reliability and efficiency of the recipe. Our evaluation shows that recipe achieves a 100% success rate of disaster recovery while adding mere seconds of overhead to the recovery time.

CCS CONCEPTS

• **Computer systems organization** → **Reliability**; **Cloud computing**; **Availability**.

KEYWORDS

Disaster Recovery, Cloud Computing, Kubernetes, Reliability, Containerized, Multi-cluster

ACM Reference Format:
Runyu Jin, Paul Muench, Travis Janssen, Brian Hatfield, and Veera Deenadhayalan. 2024. Baking Disaster-Proof Kubernetes Applications with Efficient Recipes. In *Companion of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24 Companion), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3629527.3651417

ICPE '24 Companion, May 7–11, 2024, London, United Kingdom

ACM ISBN 979-8-4007-0445-1/24/05.
https://doi.org/10.1145/3629527.3651417

1 INTRODUCTION

Kubernetes has become the container orchestration platform of choice across the IT industry [26]. Kubernetes is no longer focused only on stateless applications and as a result the users of Kubernetes are concerned with data resiliency [25]. This paper focuses on disaster recovery (DR). A naive approach to protecting stateful applications against disasters is to replicate all application persistent storage and all Kubernetes resources to a safe recovery cluster. This naive approach failed for 60% of 10 stateful applications evaluated. This paper proposes disaster recovery recipes, which is a method of implementing disaster recovery for Kubernetes applications for which the naive approach fails.

Recipes in our disaster recovery solution have 4 goals: (1) provide a disaster recovery solution to currently deployed applications (2) enable Site Reliability Engineers or developers to create disaster recovery recipes for applications (3) enable application developers to create disaster recovery hooks when other recovery techniques are not sufficient (4) make application disaster recovery reliable and efficient. Achieving all of these goals requires a user to deploy a disaster recovery framework along with Kubernetes as Kubernetes is not inherently disaster resistant. The disaster recovery framework discussed in this paper is Ramen [20], which is an IBM open-source project that provides a naive DR solution to Kubernetes applications. We implemented recipes in Ramen and evaluated the recipe design using Ramen. The resulting implementation is now available on the Ramen GitHub.

Ten Kubernetes data management system applications were studied. Four of the data management systems were able to recover from 100% of simulated disasters without a recipe. Two of the data management systems required recipes without hooks to achieve any successful recovery from a disaster. The remaining four applications require both recipes and hooks and that recovery technique is not the focus of this paper. With 1 to 3 days effort each of the 2 applications that needed recipes without hooks recovered from 100% of simulated disasters. On average the Kubernetes resource restore time for applications is 28% of the application's recovery time. Our DR solution has a small and fixed recovery time for any size of raw data. Thus our approach is both highly effective and highly efficient.

Our disaster recovery solution with recipes is the only known solution that can combine filtering, ordering, and hooks for Kubernetes resource protection and recovery. These protection and

recovery mechanisms enable two essential values. First, our solution can be applied to running applications. Second, our solution can be leveraged by users without changing application code which can make a solution rapidly available. Other solutions exist to protect Kubernetes applications from disasters. There are backup/restore solutions that require time to restore data before an application is restarted [21], which makes total recovery time longer. There are solutions based on cross data center Kubernetes clusters [19], but those solutions rely on synchronous replication. Synchronous replication is not suitable to address region wide disasters as the communication latency for maintaining cluster membership across regions is too high. There are other solutions like ours that can protect applications across regions with data in place recovery [30]. However, these other solutions do not provide the flexible Kubernetes resource protection and recovery mechanisms that are needed by data management systems.

2 BACKGROUND

Business continuity is the ability of a business to meet the demands of its clients by recovering from diverse types of problems as quickly as possible at a reasonable cost. The types of problems that affect business continuity can be broadly classified as follows: (1) failure of subsystems (such as, nodes, network, etc.) within a data center or availability zone (AZ), (2) unrecoverable corruption of data (either accidentally or maliciously), and (3) failure of the entire data center or multiple data centers in a region. These problems are solved respectively using: (a) high availability (HA) solutions, such as redundancy within the data center to avoid single points of failure, (b) backup and restore (B/R) of data using an external secondary store, and (c) disaster recovery (DR) solutions that replicate the application state to another data center that is in a different AZ or region [24]. In our paper, we focus on DR but some of the problems we outline in Section 3 also apply to B/R.

The goal of B/R and DR solutions is to recover critical application state after a disaster event. Bare-metal servers, virtual machines, and containers typically store their bootstrap configuration data in locally attached persistent volumes (e.g., in the `/etc` directory of the root file system) and may store their application volume data either in locally attached persistent volumes or in remote network attached persistent volumes. Whereas, in Kubernetes, application state is a combination of (a) Kubernetes API resources stored in an `etcd` backing store [17] and (b) volume data stored in persistent volumes. Examples of API resource types are deployments, pods, PersistentVolumeClaims (PVCs), PersistentVolumes (PVs), services, secrets, and ConfigMaps.

There are many DR solutions for Kubernetes. Kubernetes stretched cluster is a solution that takes a single Kubernetes cluster and stretches it by placing all the control plane components, including etcd, across AZs within a region. This solution leverages the HA features of Kubernetes to build a basic DR solution across multiple AZs [18]. Given that this solution uses a single etcd cluster across multiple AZs, it does not suffer from the multi-cluster DR problems we motivate in Section 3. However, this solution suffers from the effects of network latency across AZs. The longer the distance between the AZs, the higher the latency is between them and proportionally severe is the latency effect on etcd replication I/O across AZs. High network latencies can cause etcd to miss heartbeats, experience timeouts, result in leader elections that are disruptive to the cluster, and can also lead to API slowness [35]. This makes it unsuitable to stretch the cluster across long distances. Hence, this limitation of Kubernetes stretched cluster serves as a motivation for Kubernetes multi-cluster DR, which is the focus of this paper. Kubernetes multi-cluster DR [34] is a solution that overcomes the limitations of Kubernetes stretched cluster by using multiple independent Kubernetes clusters, each with its own etcd backing store.

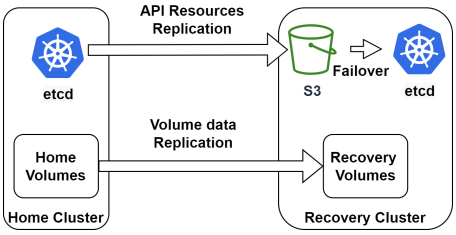

Figure 1: Kubernetes Multi-cluster DR

Kubernetes enables building DR features with its extensible design paradigm. While DR features are not part of core Kubernetes, many vendors offer custom extensions to Kubernetes. Ramen [20] is an example of an open-source DR solution. The Ramen DR solution can handle applications deployed using GitOps [7] and other traditional deployment methods. It focuses on protection and recovery of Kubernetes API resources only. To protect API resources, Ramen captures selected API resources, asynchronously stores them in an object store in the recovery DR cluster, and restores them to the etcd of the recovery DR cluster after a disaster event.

3 NAIVE KUBERNETES DR

Our primary motivation is to disaster-proof stateful Kubernetes applications (a) without modifying the applications themselves and (b) irrespective of the application deployment methodology [2].

3.1 Limitations Of The Naive Approach

GitOps [7] uses CI/CD pipelines to automate deployment of applications using declarative object-configuration stored in Git repositories. A naive approach to recover such a GitOps deployed application is to again use GitOps on the recovery cluster. However, this approach may not work for all types of applications. Consider an example of a GitOps deployed stateful application that has PVCs. Kubernetes will dynamically provision PV resources to fulfill the PVCs. These locally created PV resources in the home cluster do not come from the declarative Git source. If one were to use GitOps in the recovery cluster, Kubernetes will dynamically provision new PV resources to fulfill the PVCs that do not contain PV contents at the home cluster, resulting in data loss. We expect other application specific examples where not all API resources come from a declarative source, which makes GitOps-based DR fall short as a full DR solution. This implies that we need to capture the API resources stored in the home cluster and restore it to the recovery cluster.

A naive capture of the API resources on the home cluster periodically queries the API server and stores the results in the recovery cluster. After a disaster event, a naive recovery of API resources

on the recovery cluster restores previously captured API resources of the home cluster to the recovery cluster. This naive approach does not consider parent and child resources, sequence resource creation, nor exclude any resources.

We define a naive DR approach to use naive capture, naive recovery, or use naive GitOps for DR. While naive DR appears to be simple and intuitive, our case-study shows examples where naive DR falls short due to problems that come from (a) the application (b) the third-party DR service that is in use, (c) Kubernetes itself. Kubernetes aims to eliminate the need for orchestration of complex applications with its design comprising a set of independent, composable control processes that continuously drive the current state towards the provided desired state, but these principles do not apply to diverse applications that are out in the wild.

3.2 Case Study Using The Naive Approach

We selected 10 popular Kubernetes applications to study how the naive approach works. Five are database management systems ranked among DB-Engine's seven most popular either directly or as open-source relatives: Elasticsearch [8], EnterpriseDB [12], MariaDB [14], MongoDB [27], and Redis [36, 39]. Two are popular open-source machine learning frameworks: PyTorch [31] and TensorFlow [1, 40]. Jenkins [22] is one of the most popular continuous integration/continuous delivery (CI/CD) tools [6]. Apache Kafka [13] is the most popular open-source event streaming platform [23]. And Apache Spark [15] is an engine for large-scale data processing used by 80% of Fortune 500 companies [15].

The naive approach failed to recover 60% of the applications from simulated disaster. We categorize the reasons for failure into four modes. Table 1 below lists the applications tested and for which reasons they failed.

		Naive Approach Failure Reason			
Application	Success	Absence	Order	Stale	Mode
Elasticsearch		✓	✓		
EnterpriseDB					✓
Jenkins	✓				
Kafka	✓				
MariaDB				✓	
MongoDB	✓				
PyTorch				✓	
Redis					✓
Spark	✓				
TensorFlow				✓	

Table 1: Naive Approach DR Results

Following are examples of how applications failed to recover from simulated disaster, including some analysis of why the failures happened. Each example failure is classified according to the failure reason in Table 1. An application that fails to recover due to *absence* requires a Kubernetes API resource that is missing. For example, Elasticsearch could not query `ApmServer` resources because its custom resource definition (CRD) resource was not installed. The CRD resource type is cluster-scoped and the naive approach restores cluster-scoped resources by exception only. A CRD is only restored if a resource of the type it defines exists in the application's namespace [5].

The naive approach restores most resource types in alphabetical order by type name [4], e.g., `Deployments`, `Jobs`, `StatefulSets`. It recreates resources without delay between types, but some applications benefit from a different order or delay. The *order* column identifies applications that failed to recover due to their Kubernetes API resources being restored in an incorrect order. For example, Elasticsearch typically failed to reach healthy status when an `EnterpriseSearch` resource was restored immediately after its associated `Elasticsearch` one, but succeeded whenever they were separated by a five-second delay [9]. We discovered this accidentally by specifying an explicit restore order which introduced the delay.

Some applications failed to recover when *stale* information specific to the home cluster persisted in their Kubernetes API resources. For example, PyTorch, deployed by the OpenShift Data Science operator, failed to recover because an endpoint, containing the home cluster's name, was unreachable. The application recovered after replacing the home cluster's name with the recovery cluster's. Some applications require a specific *mode* to restart on another cluster. Redis, for example, failed to restart with the naive approach, but succeeded when its `RedisEnterpriseCluster` resource was modified setting its `spec.clusterRecovery` mode to `true` [37]. In summary, the naive approach to DR worked for some applications, but not all. The naive approach worked for applications that omitted cluster-specific information, did not have restore order requirements, had no dependencies, and generally tolerated being restored on another cluster.

4 ROBUST KUBERNETES DR USING RECIPES

Theoretically, Kubernetes application resources can be backed up as a single unordered group, then restored as a single unordered group, and eventually regain a functional state, provided the application data is available. Our case study indicates that this assumption does not hold true for many applications. Further, the issues that prevent backup or recovery in a single group may not be fixed without modifying the underlying application. To address these issues, we introduce the Recipe concept for robust disaster recovery.

4.1 Recipe

A recipe is a Kubernetes custom resource (CR) that defines the capture and recovery sequences of Kubernetes objects. Recipes enable and automate DR for any application. The recipe design specifically addresses the issues experienced with naive DR as discussed in Section 3. A key abstraction in a recipe is a workflow, which defines a sequence of actions to take during a capture or recovery sequence. A workflow is a sequentially processed list of groups and hooks. Groups define the resources to be included or excluded in a step of a workflow, and Hooks define actions that should be run in between groups. With these three abstractions, all issues encountered during the case study can be addressed.

The absence issue encountered with ElasticSearch involved a missing CRD (ApmServer resource) whose absence failed the recovery. Backing up an object CR requires the CRD, as does restoring it. A recipe can handle this situation by capturing an active object (ApmServer CR) along with the CRD it uses. This is preferred to

installing the full operator which includes the CRD on the recovery cluster, as the recovery cluster version may not match the operator version without prior planning.

Combinations of groups enable sequencing a recovery process. If a particular resource is dependent on another in a parent-child relationship, they can be split up into two groups to ensure the parent resource exists before the child resource. Groups address the absence and order problems encountered with ElasticSearch, where the ElasticSearchCluster CR needs to be available before the EnterpriseSearch CR (Recipe Example 1).

Hooks can address scenarios where an object must be modified after restoration [3]. Restoring an object involves copying the resource contents from the home cluster and reproducing them on the recovery cluster. Some applications, like OpenShift Data Science, embed cluster-specific information in their resources, like Notebooks. Kubernetes resources can become available, but the application is inaccessible through an endpoint, which uses a stale URL. By correcting this data, Hooks can address the stale issues encountered in the case study.

Hooks can also address scenarios where selectively ordering or filtering resource groups is insufficient to recover an application. Redis requires that a recovery mode is specified on the CR when a majority of nodes become unavailable [37]. During recovery, Redis begins without any nodes available, and setting the recovery mode is required to launch pods and continue operation. Adding spec.clusterRecovery=true to the RedisEnterpriseCluster CR to begin the recovery mode that is required by Redis. Hooks can be used to add the recovery mode field on application CRs, addressing mode-type scenarios found during the case study.

4.2 Recipe API and Examples

Now that the high level abstractions have been explained, the API can be introduced. The sample recipe is based on ElasticSearch, but adds hooks to demonstrate the feature. The recipe object itself is divided into the three abstractions: groups, hooks and workflows. The captureWorkflow is used for the backup/capture sequence, and the recoverWorkflow is used for the restore/recover sequence. A sequence is defined using a map of strings, where a user specifies a type (Group or Hook), then the name of that group or hook (for example: "group: everything"). Each step of the sequence must be completed before the next one begins.

Groups are defined with a unique name identifier, and may include and exclude resource types by name. Groups use namespace-scoped visibility by default, but may opt-into cluster-scoped resources with an additional field (includeClusterResources = true). Since capture and recovery sequences may not be symmetrical, a backupRef field is used to source recovery contents.

A full Recipe is shown below in Recipe Example 1. In the current recipe implementation, groups are processed independently. Therefore, it is possible to restore duplicate resources across different groups. Using excludedResourceTypes avoids this scenario.

Recipe Example 1

```
apiVersion: ramendr.openshift.io/v1alpha1
kind: Recipe
metadata:
  name: recipe-demo
  namespace: eck
spec:
  appType: eck
  volumes:
    name: volumes
    type: volume
    labelSelector: {} # select all PVCs
  groups:
  - name: everything
    type: resource
    includedResourceTypes:
    - "*"
  - name: cluster
    backupRef: everything
    type: resource
    includedResourceTypes:
    - elasticsearches.elasticsearch.k8s.elastic.co
  - name: enterprise-search
    backupRef: everything
    type: resource
    includedResourceTypes:
    - enterprisesearches.enterprisesearch\
      .k8s.elastic.co
  - name: misc
    backupRef: everything
    type: resource
    excludedResourceTypes:
    - enterprisesearches.enterprisesearch\
      .k8s.elastic.co
    - elasticsearches.elasticsearch.k8s.elastic.co
  hooks:
  - name: demo-hooks
    labelSelector:
      matchLabels:
        appname: eck
    type: exec
    ops:
    - name: date
      container: main
      timeout: 10m
    command: # runs as single command: "/bin/sh -c date"
      - "/bin/sh"
      - "-c"
      - "date"
  captureWorkflow:
    sequence:
    - hook: demo-hooks/date
    - group: everything
  recoverWorkflow:
    sequence:
    - group: cluster
    - group: enterprise-search
    - group: misc
```

4.3 Implementation

Recipes are a general concept which may be used as a library component [10]. The controller logic discussed in this paper was implemented within Ramen, an open-source Disaster Recovery solution

Category	Application
Type 1	Jenkins, Kafka, MongoDB, Spark
Type 2	Elasticsearch, MariaDB
Type 3	EnterpriseDB, PyTorch, Redis, Tensorflow

Table 2: Kubernetes Applications Categorization

[20]. Ramen handles volumes replication across clusters, while recipes handle application recovery. The separation of API and controller logic allows for customization of the software stack, if a user desires. Ramen was released as a part of OpenShift Data Foundations (formerly OpenShift Container Storage) v4.7 [32].

5 EVALUATION

We evaluated the performance of recipe-based disaster recovery following the design described in Section 4. We implemented the recipe based on Ramen [20]. Our key evaluation metrics are reliability and efficiency of application recovery.

5.1 Environment Setup

We setup two RedHat Openshift Container Platform (OCP) [16] 4.12 clusters, one cluster serves as the home cluster where applications are installed and initially deployed. Another cluster serves as the recovery cluster. When a simulated disaster happens, all the applications deployed on the home cluster are recovered on the recovery cluster. The application deployed on the home cluster is removed completely including all the Kubernetes resources and persistent volumes to mimic real-world production outages. Both clusters use the same external RedHat OpenShift Data Foundation (ODF) Ceph [33] cluster as the storage backend for all the application persistent data. ODF offers the Metro Disaster Recovery (MetroDR) solution which synchronously replicates persistent data between the home and recovery clusters. We deployed Ramen on both OCP clusters to protect Kubernetes application resources every 5 minutes. We simulate a disaster scenario by removing all the applications on the home cluster, and then initiate disaster recovery on the recovery cluster.

5.2 Kubernetes Applications Categorization

Applications are categorized into three types, which reflects the amount of custom handling required with a Recipe to enable DR protection for the application. Type 1 applications do not require Recipes: all the resources can be captured and recovered on its own. Type 2 applications require Recipe Groups: either resource filtering and/or ordering are required to restore the application. Type 3 applications require both Recipe Groups and Hooks: if an application requires a program to run to reach a consistent application state, this is done with a Hook. The categorization of the 10 applications is shown in Table 2 based on the naive DR approach we studied (Section 3).

We focus the evaluation on type 1 and type 2 applications in this paper and type 3 applications evaluation will be future work. We evaluated 6 modern applications that are either type 1 or type 2 applications which are a subset of the 10 applications introduced in Section 3. The applications include Elasticsearch (v2.8.0), Jenkins (v2.432), Apache Kafka (v2.5.0), MariaDB (v0.20.0), MongoDB (v7.0), and Apache Spark (v3.1.1). Among them, Elasticsearch and MariaDB are type 2 applications and use recipes. Elasticsearch uses 3 recipe groups for ordering while MariaDB uses 4 recipe groups for both ordering and filtering. All other applications are type 1 applications and do not use recipes.

Application	Ramen w/o Recipes	Ramen w Recipes
Elasticsearch	0%	100%
Jenkins	100%	100%
Kafka	100%	100%
MariaDB	0%	100%
MongoDB	100%	100%
Spark	100%	100%

Table 3: DR Success Rate Without & With A Recipe

During the evaluation, each application is tested alone for its disaster recovery reliability and efficiency. We tested disaster recovery 40 times one way for each application from home cluster to recovery cluster and then back from recovery cluster to home cluster back to back. Each disaster recovery iteration consists of Kubernetes resource restore time and application recovery time. Kubernetes resource restore time is the time for Ramen to create all the Kubernetes resources. Ramen's role in application recovery ends once the Kubernetes objects have been created on the recovery cluster. Application recovery time includes the Kubernetes resource restore time, but also includes the time for the application to become fully usable. We will show the success rate and the recovery efficiency for the 80 runs in the following sections.

5.3 Recipe Reliability And Efficiency

Table 3 shows the success rate for the 40 runs of disaster recovery of the 6 applications. Without recipe, Elasticsearch and MariaDB cannot be successfully recovered. All the type 1 applications were able to achieve 100% success rate of recovery. After applying the recipe, Elasticsearch and MariaDB recovered successfully 100% of the time. Also, the code modifications to Ramen to include the new recipe design do not reduce the reliability of type 1 applications, all the applications were recovered 100% of the time.

Figure 2 shows the the total recovery time v.s. the Kubernetes resource restore time for the six applications. All the applications were able to complete disaster recovery within 5 minutes. The longest one was Kafka, which takes around 265 seconds. The shortest one was MongoDB, which takes around 106 seconds. Kubernetes resource restore time takes a small portion of the total recovery time, with an average of 28%. For most applications, it takes around 20% of the total application recovery time to restore Kubernetes resources. MongoDB and Jenkins finish Kubernetes resources restore the fastest with 29 seconds. MariaDB takes the longest to restore Kubernetes resources with 92 seconds around 61% of the total recovery time. All the applications were able to finish recovering Kubernetes resources within one and half minutes. Note that besides the resource size, resource types also make a difference to the resource restore time. Certain resources like pods take longer to recover due to the underlying design of Velero where the restore process requires querying of the server object by object.

After Ramen restores all the resources, it takes some more time for the application to become ready by reconciling the application

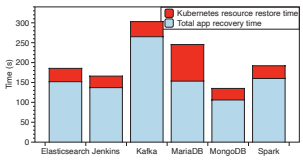

Figure 2: App DR Time v.s. Resource Restore Time

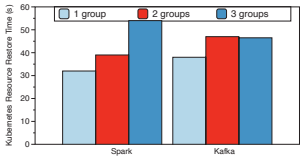

Figure 3: Adding Recipe Groups Overhead

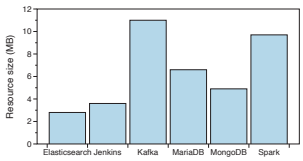

Figure 4: Application Resource Size In Megabytes

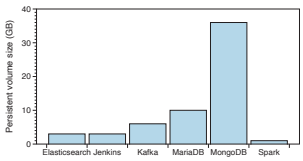

Figure 5: Application Data Size In Gigabytes

to match the desired application state in the Kubernetes resources. MariaDB takes the shortest time around 58 seconds to become ready. Kafka takes the longest of 227 seconds to become ready. Compared to other applications, Kafka has two sets of pods, zookeeper and kafka, to reconcile. MariaDB and Elasticsearch which are type 2 applications do not have a substantially longer recovery time compared to other type 1 applications.

To measure the overhead of using recipe groups, we forcefully added recipe groups to type 1 applications and show the Kubernetes resource restore time after adding different number of recipe groups. Figure 3 illustrates the results. Adding 1 group has the same Kubernetes resource restore time as not using recipe. When we add more recipe groups, the Kubernetes resource restore time increases. For Spark, from 1 group to 2 groups increases the restore time by 7 seconds. It further increases the restore time by 15 seconds from 2 groups to 3 groups. For Kafka, 2 groups and 3 groups add around 9 seconds of overhead. Most of the overhead come from communicating with API server or Kubelet when switching groups which takes around 5 seconds. We haven't tuned API server to be efficient and we believe this overhead can be eliminated after optimization. Given the reliability that recipe enhanced, recipe overhead is low in seconds and still restores resources with a reasonable amount of time.

The total recovery time is largely related with the amount of Kubernetes resources each application has. Figure 4 shows the size of the Kubernetes resources in Megabytes. In general, the larger the resource size, the longer it takes to recover the application. We can see Kafka has the largest amount of resources so it takes the longest to recover. For Elasticsearch and MariaDB which are type 2 applications, recipe adds some overhead to the total recovery time.

Figure 5 shows the total data size of all the Persistent Volumes (PV) for each application. Because we are using MetroDR which synchronously backups data volumes across the clusters, the data volumes don't need to be restored during disaster recovery. This brings the benefits that the disaster recovery time is not related to the data size of the PVs. Although MongoDB has the largest PV size, it doesn't take the longest to restore MongoDB.

6 RELATED WORKS

Disaster recovery for Kubernetes applications is new enough that there are no standards for how it should be deployed. Disaster recovery solutions can be divided into two classes, solutions for stateless applications and solutions for stateful applications. At this time the techniques used for each class of disaster recovery solution are distinct, but how techniques can be reused between the two classes is an open area of research. So this section will compare our solution with both classes of application disaster recovery.

The most common disaster recovery approach for Kubernetes stateful applications is to use a backup/restore solution with persistent volumes and Kubernetes resources being protected on a remote site. De et al [11], Pakrijauskas and Mažeika [29] and Rubio [38] all evaluate backup/restore solutions in the cloud. These studies do not focus on evaluating successful disaster recovery for a set of Kubernetes applications. This can explain why De et al [11], Pakrijauskas and Mažeika [29] and Rubio [38] do not mention the need for a disaster recovery solution based on more than simple replication. Torta [41] focuses on disaster recovery managed by data management systems. Due to the active-active nature of data management system disaster recovery the associated recovery time can be very small. The limitation of using the disaster recovery strategy in Torta [41] is that every application must have its own disaster recovery solution. This application by application approach to disaster recovery can be complex to manage and hence risky to operate. Tran et al [42] investigates the mechanisms for the protection and recovery of running containers based based on application checkpoints. Tran et al [42] does not investigate the protection and recovery of Kubernetes applications with persistent volumes and Kubernetes resources.

Stateless applications do not use persistent volumes and do not rely on updates to Kubernetes resources. Hence these applications can easily be recovered after a disaster. Moshfeghifar [28] studies stateless applications in the form of serverless computing. The key challenges addressed in Moshfeghifar [28] are deploying applications across multiple clusters and getting those clusters to act like a single cluster environment.

7 CONCLUSION AND FUTURE WORK

This paper presents a disaster recovery (DR) solution for Kubernetes. The novelties of this work lie in first, categorization of the problems that current modern Kubernetes applications have when doing DR; second, present a novel disaster recovery solution called recipes to enable DR for all the modern applications without modifications to the applications; third, evaluate the recipe solution to confirm that recipe can achieve 100% success rate of DR with low overhead. In our future work, we will further explore recipe design with hooks and how hooks can help disaster recovery of type 3 applications. Besides, we will explore the disaster recovery solution's applicability for large scale database deployments and databases under continuous load. We will also validate application data staleness.

REFERENCES

[1] Omdia Analyst. 2022. *The Evolution of ML Frameworks Report - 2022.* Technical Report. Omdia.
[2] The Kubernetes Authors. 2023. Kubernetes Object Management. https://kubernetes.io/docs/concepts/overview/working-with-objects/object-

management/
[3] Velero Authors. 2023. Velero 1.12 Restore Resource Modifiers. https://velero.io/docs/v1.12/restore-resource-modifiers/
[4] Velero Authors. 2023. Velero Docs - Restore Reference. https://velero.io/docs/v1.9/restore-reference/#restore-order
[5] Velero Authors. 2023. velero/pkg/backup/backup.go at 9b5678f32a4aa696de5d645d15bc0ff1f989f464 · vmware-tanzu/velero. https://github.com/vmware-tanzu/velero/blob/9b5678f32a4aa696de5d645d15bc0ff1f989f464/pkg/backup/backup.go#L410-L419
[6] Michael Azoff. 2023. *Omdia Universe: DevOps Release Management Solutions, 2023.* Technical Report. Omdia.
[7] Florian Beetz and Simon Harrer. 2022. GitOps: The Evolution of DevOps? *IEEE Software* 39, 4 (2022), 70–75. https://doi.org/10.1109/MS.2021.3119106
[8] Elasticsearch B.V. 2022. Elasticsearch Platform — Find real-time answers at scale | Elastic. https://www.elastic.co/
[9] Elastic B.V. 2023. Prerequisites | Enterprise Search documentation [8.11] | Elastic. https://www.elastic.co/guide/en/enterprise-search/current/prerequisites.html#prerequisites
[10] IBM Corp. 2023. Recipe API. https://github.com/RamenDR/recipe/blob/main/api/v1alpha1/recipe_types.go
[11] Suman De, R Prashant Singh, et al. 2022. Selective Analogy of Mechanisms and Tools in Kubernetes Lifecycle for Disaster Recovery. In *2022 IEEE 2nd International Conference on Mobile Networks and Wireless Communications (ICMNWC)*. IEEE, IEEE, 3 Park Avenue, 17th Floor New York, NY 10016-5997 USA, 1–6.
[12] enterprisedb. 2023. EnterpriseDB. "https://www.enterprisedb.com/"
[13] Apache Software Foundation. 2022. Apache Kafka. https://kafka.apache.org/
[14] MariaDB Foundation. 2022. https://mariadb.org/. MariaDBServer: Theopensourcerelationaldatabase
[15] The Apache Software Foundation. 2022. Unified engine for large-scale data analytics. https://spark.apache.org/
[16] The Linux Foundation. 2022. OpenShift Container Platform 4.12 Documentation. https://docs.openshift.com/container-platform/4.12/welcome/index.html
[17] The Linux Foundation. 2023. Kubernetes Components. https://web.archive.org/web/20231025011453/https://kubernetes.io/docs/concepts/overview/components/#etcd
[18] The Linux Foundation. 2023. Kubernetes: Running in multiple zones. https://web.archive.org/web/20231020051135/https://kubernetes.io/docs/setup/best-practices/multiple-zones/
[19] Red Hat. 2023. OpenShift Disaster Recovery using Stretch Cluster. https://red-hat-storage.github.io/ocs-training/training/ocs4/ocs4-metro-stretched.html
[20] Red Hat. 2023. Ramen DR opensource project. https://github.com/RamenDR/ramen/
[21] IBM. 2021. Overview of Kubernetes Backup Support. https://www.ibm.com/docs/en/spp/10.1.5?topic=containers-overview
[22] Jenkins. 2022. Jenkins. https://www.jenkins.io/
[23] Alex Johnston. 2022. *Connectivity is the watchword as Confluent continues to expand.* Technical Report. 451 Research.
[24] Th. Lumpp, J. Schneider, J. Holtz, M. Mueller, N. Lenz, A. Biazetti, and D. Petersen. 2008. From high availability and disaster recovery to business continuity solutions. *IBM Systems Journal* 47, 4 (2008), 605–619. https://doi.org/10.1147/SJ.2008.5386516
[25] Parth Sandip Mehta. 2023. *NoSQL Databases in Kubernetes.* Master's thesis. San Jose State University. https://doi.org/10.31979/etd.qrrp-3equ
[26] Christine Miyachi. 2021. The Rise of Kubernetes. In *2021 Cloud Continuum*. IEEE, 3 Park Avenue, 17th Floor New York, NY 10016-5997 USA, 1–5. https://doi.org/10.1109/CloudContinuum54760.2021.00002
[27] Inc. MongoDB. 2022. MongoDB: For the next generation of intelligent applications. https://www.mongodb.com/
[28] Amirhossein Moshfeghifar. 2022. *Active Disaster Recovery Strategy for Applications Deployed Across Multiple Kubernetes Clusters, Using Service Mesh and Serverless Workloads.* Master's thesis. Tampere University.
[29] Kęstutis Pakrijauskas and Dalius Mažeika. 2021. On recent advances on stateful orchestrated container reliability. In *2021 IEEE Open Conference of Electrical, Electronic and Information Sciences (eStream)*. IEEE, IEEE, 3 Park Avenue, 17th Floor New York, NY 10016-5997 USA, 1–6.
[30] Portworx. 2023. Disaster Recovery. https://docs.portworx.com/portworx-enterprise/operations/operate-kubernetes/disaster-recovery
[31] pytorch. 2023. PyTorch. "https://pytorch.org/"
[32] Inc. Red Hat. 2021. OpenShift Container Storage 4.7 release notes. https://access.redhat.com/documentation/en-us/red_hat_openshift_container_storage/4.7/html-single/4.7_release_notes/index
[33] Inc. Red Hat. 2022. Red Hat OpenShift Data Foundation. https://www.redhat.com/en/technologies/cloud-computing/openshift-data-foundation
[34] Inc. Red Hat. 2023. Introduction to OpenShift Data Foundation Disaster Recovery. https://web.archive.org/web/20231010175615/https://access.redhat.com/documentation/en-us/red_hat_openshift_data_foundation/4.13/html-single/configuring_openshift_data_foundation_disaster_recovery_for_openshift_workloads/index#introduction-to-odf-dr-solutions_common
[35] Inc Red Hat. 2024. Recommended etcd practices. https://web.archive.org/web/20231105012238/https://docs.openshift.com/container-platform/4.14/scalability_and_performance/recommended-performance-scale-practices/recommended-etcd-practices.html
[36] redis. 2023. https://redis.io/. Redis
[37] Redis. 2023. Recover a Redis Enterprise cluster on Kubernetes | Redis Documentation Center. https://docs.redis.com/latest/kubernetes/re-clusters/cluster-recovery/
[38] Sergio Fernández Rubio. 2022. *Disaster Recovery Analysis of different Cloud Managed Kubernetes Clusters.* Master's thesis. Edinburgh Napier University. https://www.researchgate.net/profile/Sergio-Fernandez-Rubio/publication/363632856_Disaster_Recovery_Analysis_of_different_Cloud_Managed_Kubernetes_Clusters/links/6325ee52873eca0c0094f0e1/Disaster-Recovery-Analysis-of-different-Cloud-Managed-Kubernetes-Clusters.pdf
[39] solid IT. 2023. DB-Engines Ranking. https://db-engines.com/en/ranking
[40] tensorflow. 2023. TensorFlow. "https://www.tensorflow.org/"
[41] Francesco Torta. 2023. *Business Continuity in Kubernetes Multi-Cluster Environments.* Ph. D. Dissertation. Politecnico di Torino.
[42] Minh-Ngoc Tran, Xuan Tuong Vu, and Younghan Kim. 2022. Proactive Stateful Fault-Tolerant System for Kubernetes Containerized Services. *IEEE Access* 10 (2022), 102181–102194.