Pods-as-Volumes: Effortlessly Integrating Storage Systems and Middleware into Kubernetes

Alberto Faria
INESC TEC & University of Minho

Ricardo Macedo
INESC TEC & University of Minho

João Paulo
INESC TEC & University of Minho

ABSTRACT

We present Pods-as-Volumes (PaV), a Kubernetes plugin that simplifies the implementation of storage volume provisioners by allowing all logic underlying the lifecycle and behavior of volumes to be specified as pod templates, which are then instantiated as needed to create, delete, and expose volumes to applications. PaV reduces the effort required to integrate storage systems into Kubernetes and enables the straightforward creation of storage middleware components, improving modularity and Kubernetes' ability to manage storage stacks.

CCS CONCEPTS

• **Software and its engineering** → *Software as a service orchestration system*; • **Information systems** → *Information storage systems*.

KEYWORDS

Kubernetes, storage, middleware

ACM Reference Format:
Alberto Faria, Ricardo Macedo, and João Paulo. 2021. Pods-as-Volumes: Effortlessly Integrating Storage Systems and Middleware into Kubernetes. In *Seventh International Workshop on Container Technologies and Container Clouds (WoC '21), December 6, 2021, Virtual Event, Canada.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3493649.3493653

1 INTRODUCTION

The Kubernetes container orchestration system [9] can automate and assist in managing many aspects of application deployment, such as scheduling containers to cluster nodes, handling failures, and scaling. In addition, it can expose storage resources to applications through a *volume* abstraction, enabling application *pods*—sets of one or more containers that run as a unit—to access those resources uniformly via a file system or block device interface regardless of the storage system that underlies them. This requires that a corresponding *provisioner*, which implements the lifecycle and behavior of volumes backed by that storage system, be available.

Kubernetes includes built-in provisioners for several storage systems such as Ceph [13] and Gluster [3], and others may be added by implementing the Container Storage Interface (CSI) [10, 12], which standardizes the interaction between container orchestration systems and providers of storage resources. However, fully and correctly implementing the CSI specification can take significant effort, typically requiring the construction of two gRPC [5] servers and the deployment of components across all cluster nodes.

In addition, Kubernetes' ability to orchestrate more elaborate storage stacks is limited. For instance, to ensure that data is encrypted at rest, encryption capabilities must either be built into all application containers or supported by every employed storage system and provisioner. This lack of modularity manifests further when there is a need to combine several storage functionalities (*e.g.*, applying both encryption and compression to a volume), leading to increased development effort and hampering the adaptability of the storage infrastructure to the needs of the applications.

We thus propose Pods-as-Volumes (PaV), a Kubernetes plugin that simplifies the implementation of volume provisioners. PaV enables users to specify through templates of pod definitions the logic for creating, deleting, and exposing volumes to applications. Pods are then created and run from those templates automatically to perform the corresponding actions. By introducing this approach, and compared to the alternative of manually implementing the CSI specification and deploying all necessary components, PaV significantly reduces the effort required to integrate storage systems into Kubernetes. We demonstrate this with a fully-functional integration of the Google Cloud Storage [4] service, amounting to only 62 lines of YAML and requiring no coding (§ 4.1).

Further, PaV enables the straightforward implementation of middleware components that may be placed in between volumes and application pods, and which are themselves exposed as volumes. We showcase this with a 70-line, transparent encryption layer that relies on the *cryptsetup* disk encryption tool [1] and can be applied to existing volumes regardless of whether their underlying storage systems and provisioners support encryption (§ 4.2). Several middleware components of this kind may be stacked to build complex storage architectures. PaV thus enables Kubernetes to orchestrate and manage the storage infrastructure that applications require with greater modularity and adaptability to their needs. PaV is available as an open-source project at:

https://github.com/albertofaria/pav

2 BACKGROUND

Kubernetes [9] is a container orchestration system for automating the deployment and management of applications, enabling their components to be defined and configured declaratively as *objects*, most commonly using the YAML data serialization language. Kubernetes reacts to the creation of these objects by instantiating and managing containers and other resources in order to match the desired configuration.

Pods. Objects can be of several types, or *kinds*. Each object is identified by a name, and objects of most kinds are partitioned into

WoC '21, December 6, 2021, Virtual Event, Canada

ACM ISBN 978-1-4503-9171-9/21/12…$15.00
https://doi.org/10.1145/3493649.3493653

namespaces. The most fundamental kind is the Pod. A *pod* is a set of one or more containers. When a Pod object is created, Kubernetes schedules those containers to run together on a single node in the cluster. In practice, container deployments are configured using higher-level objects that internally manage Pod objects. An example are `Deployment` objects, which combine the definition of a pod with a replication factor. When a `Deployment` is created, a corresponding number of underlying Pod objects are also created automatically. Further, whenever one of the pods fails (*e.g.*, due to a container crash), its object is deleted and a new one is created to restore the desired replication factor.

Volumes. In addition to containers, Kubernetes can manage persistent storage resources through the *volume* abstraction. A volume corresponds to a file system or block device, and can be accessed by containers in pods.

To allocate a volume, one creates a `PersistentVolumeClaim` (PVC) object specifying certain desired properties of the volume (*e.g.*, its minimum capacity) and referencing an existing `StorageClass` object. The latter in turn identifies and configures the *provisioner* that will be used to allocate the volume. Kubernetes has several built-in provisioners that rely on storage systems such as Ceph [13] and Gluster [3] to obtain storage resources. Once the volume is allocated, a `PersistentVolume` object representing it is created and bound to the PVC, from which point the volume may be used by pods. This process is known as *dynamic provisioning*.

A user may also *statically provision* a volume by creating the `PersistentVolume` object manually, in which case no automated resource allocation is performed for the volume.

Container Storage Interface. It may be desirable to add new volume provisioners to a Kubernetes cluster, for instance to leverage a storage system for which there is no built-in support. Provisioners can be created by implementing the Container Storage Interface (CSI) [10, 12], a gRPC [5] interface that standardizes the interaction between container orchestration systems and providers of storage resources. Such an implementation is termed a *CSI plugin* and typically consists of two gRPC servers: (1) the *controller plugin*, which runs as a single instance in the cluster and manages volume creation and deletion; and (2) the *node plugin*, deployed on every node and responsible for making volumes available to pods.

However, in addition to requiring the deployment of components across all nodes of a cluster, fully and correctly implementing the CSI specification can take significant effort. One particular difficulty arises with regard to error handling and cleanup when CSI plugin components fail during interaction with Kubernetes, and will be detailed further ahead in the context of PaV's implementation, which relies on CSI.

3 DESIGN AND IMPLEMENTATION

We present PaV, which simplifies the creation of volume provisioners by allowing all logic underlying volume allocation and behavior to be specified as templates of pod definitions. Pods are then instantiated from those templates and run automatically when needed. By introducing and building upon this concept, PaV is able to encapsulate the intricacies of the CSI specification. In this section, we describe how PaV is used, its architecture, and its implementation.

```
1  apiVersion: pav.albertofaria.github.io/v1alpha1
2  kind: PavProvisioner
3  metadata: (...)
4  spec:
5    provisioningModes: list containing Dynamic and/or Static
6    volumeValidation:
7      volumeModes: list containing Filesystem (default) and/or Block
8      accessModes: list containing ReadWriteOnce, ReadOnlyMany,
9                   and/or ReadWriteMany (default is all three)
10     minCapacity: numeric, possibly suffixed (default is no minimum)
11     maxCapacity: numeric, possibly suffixed (default is no maximum)
12     podTemplate: pod definition
13   volumeCreation:
14     volumeHandle: string
15     capacity: numeric, possibly suffixed
16     podTemplate: pod definition
17   volumeDeletion:
18     podTemplate: pod definition
19   volumeStaging:
20     podTemplate: pod definition
21   volumeUnstaging:
22     podTemplate: pod definition
```

Figure 1: Schema for `PavProvisioner` objects.

3.1 `PavProvisioner`

Volume provisioners are implemented using PaV by creating objects of the `PavProvisioner` kind, which becomes available when PaV is installed on a Kubernetes cluster. Each such object gives rise to a provisioner, and the aforementioned pod templates are specified directly in its definition.

Figure 1 depicts the schema for `PavProvisioner` objects. Briefly, one may specify templates of pods for validating the configuration of a volume to be provisioned (line 12), for creating and deleting a volume as part of dynamic provisioning (lines 16 and 18), and for *staging* a created volume—the action of making it available for use by a given client pod—and subsequently *unstaging* it (lines 20 and 22). When any these actions is requested, a pod is instantiated from the appropriate template and run automatically by PaV to perform all necessary work. For brevity, we omit discussion of other `PavProvisioner` fields. Two example `PavProvisioner` definitions are reproduced and explained in § 4.

3.2 Architecture

PaV is comprised of two main components: an *agent* and a *CSI plugin*. The agent is deployed once per cluster, and its first responsibility is monitoring `PavProvisioner` objects. For each created `PavProvisioner`, it deploys an instance of the CSI plugin, which implements the provisioner in question.

The agent is divided into two subcomponents: the *controller agent* and the *node agent*. The first runs as a single pod in the cluster, and the second is deployed on all nodes. The CSI plugin is similarly split into a *CSI controller plugin* and *CSI node plugin*, which are deployed for each `PavProvisioner` object in the same fashion as the agent. This is exemplified in Figure 2, which depicts a PaV deployment on a two-node cluster in which a single `PavProvisioner` exists. The controller agent happened to be scheduled to node A, and the `PavProvisioner`'s CSI controller plugin to node B.

Both the agent and CSI plugin are implemented in the Python language and packaged as a single container image. Asynchronous I/O is used in lieu of multithreading for improved resource

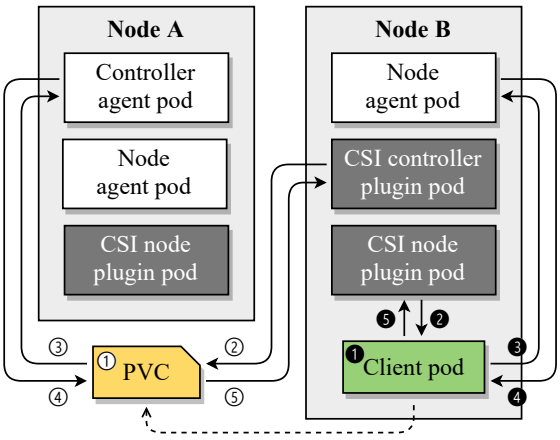

Figure 2: Agent and CSI plugin components of a PaV deployment in a 2-node cluster with a single `PavProvisioner` object. Also represented are a `PVC` and a `Pod` that uses it.

consumption, and the agent relies on the *kopf* framework [8] to monitor objects.

3.3 Workflow

The components enumerated above work in tandem to implement the functionality of each volume provisioner defined by `PavProvisioner` objects. Here, we illustrate the interactions between users, Kubernetes, agent, and CSI plugins by describing the steps involved in creating and staging volumes, under the scenario in Figure 2.

Volume creation. The dynamic provisioning of a volume by a `PavProvisioner` begins when a user creates a `PVC` object that ultimately references that provisioner (①). Kubernetes reacts to the creation of such an object by performing a Remote Procedure Call (RPC) on the appropriate CSI controller plugin, instructing it to create a volume.

Instead of directly taking action to create the volume, the CSI controller plugin then encodes Kubernetes' request under specific *annotations*—arbitrary key-value pair metadata that may be attached to any Kubernetes object—on the `PVC` that initially triggered provisioning (②). PaV's controller agent monitors `PVC` objects for such annotations, and detects the request to create a volume, taking on that burden (③).

After validating the requested volume configuration, the controller agent instantiates and runs the *volume creation pod* (not depicted in the figure) defined in the `PavProvisioner`. It then waits until this pod terminates, and encodes information about its success or failure under other annotations on the same `PVC` object (④). The CSI controller plugin, still in the context of the aforementioned RPC call, notices this and returns that information back to Kubernetes (⑤). Upon success, a `PersistentVolume` object representing the volume is then created and bound to the `PVC`.

Volume creation work is delegated from the CSI controller plugin to the controller agent to ensure that error handling and cleanup is performed under certain failure scenarios. Consider, for instance, that the CSI controller plugin fails immediately prior to returning an indication of successful volume creation to Kubernetes. In this case, persistent resources backing the volume have already been allocated, but Kubernetes is not aware of this fact and there is no guarantee that it will retry the RPC or even perform a cleanup RPC.

To guarantee that resources are not leaked, PaV thus stores information about the volume creation process in the annotations of `PVC` objects. These objects are then continuously monitored, and when they are being deleted, their annotations are checked to determine whether the volume creation pod has been run for the corresponding volume. If it has, PaV then instantiates the `PavProvisioner`'s *volume deletion pod* to free any resources that may have been allocated. While each `PavProvisioner`'s CSI controller plugin could perform this monitoring, it is less resource-intensive to have a single component doing so, and thus this responsibility is transferred to the controller agent. The task of performing volume creation is then also delegated to it for symmetry.

Volume staging. Figure 2 also depicts a client `Pod` that requests access to a volume of the `PavProvisioner` by referencing its `PVC` object, and that was scheduled to node B (❶). Before starting the containers in the pod, Kubernetes performs an RPC on the `PavProvisioner`'s CSI node plugin instance running on node B, instructing it to stage the volume.

Analogously to volume creation, the CSI node plugin encodes Kubernetes' request as annotations on the client `Pod` (❷). The node agent on the same node notices this (❸) and instantiates the *volume staging pod* (not depicted in the figure). A dedicated directory from the host is mounted at path `/pav` in all containers of this pod, which must make the volume available as a directory or block special file at `/pav/volume`. Information about the staging pod's success or failure is then recorded in the client `Pod` (❹), and relayed to Kubernetes by the CSI node plugin (❺). Upon success, Kubernetes retrieves the volume from the aforementioned host directory and exposes it to the client pod, which finally begins execution.

The task of running the volume staging pod is delegated to the node agent for the same reasons as for volume creation. The node agent is similarly responsible for running the *volume unstaging pod* under failure scenarios that cause Kubernetes not to request so, and accomplishes this by monitoring the deletion of `Pod` objects.

3.4 Discussion

As explained, PaV introduces and builds upon the concept of specifying the logic underlying the allocation and behavior of volumes as pod templates, which are then instantiated as needed to create, delete, stage, and unstage volumes. It should be clear from the description above that directly implementing the CSI specification requires the creation of various components, and can involve many intricacies related to error handling and cleanup due to the possibility of those components failing during interaction with Kubernetes. In contrast, creating a single `PavProvisioner` object is sufficient to implement a volume provisioner, and PaV provides the clear-cut guarantee that all runs of the volume creation (staging) pod are eventually succeeded by a run of the volume deletion (unstaging) pod, even when they fail.

Note that since PaV simply orchestrates the lifecycle of volumes, not interacting with storage components while they are in use by application pods, it does not incur overhead on volume performance. However, since the creation, deletion, staging, and unstaging of

volumes require additional pods to be run, these steps are subject to the delay between the creation of a Pod and the start of its execution. This is typically in the order of a few seconds but varies between clusters. Nonetheless, since Kubernetes applications are generally long-running, this additional latency is often insignificant.

4 USE CASES

Having described PaV's design and implementation, we now present two example use cases that showcase its applicability and ease of use: an integration of Google Cloud Storage into Kubernetes, and a transparent middleware component that adds encryption capabilities to existing volumes.

4.1 Google Cloud Storage

Google Cloud Storage (GCS) [4] is an object storage service. Objects are organized into *buckets*, and libraries for several languages are provided to access them. A *gcsfuse* tool [2] is also provided to expose buckets through the POSIX file system interface, presenting objects as files and enabling file-based applications to use GCS. However, Kubernetes has no support for exposing GCS buckets as volumes, and we thus implement this capability using PaV.

Figure 3 reproduces the *complete* definition[1] of a `PavProvisioner` implementing a fully-functional integration of GCS into Kubernetes. It supports both dynamic and static provisioning (line 5), with each volume corresponding to a bucket. Note that several string fields of the pod templates include expressions enclosed in {{ ... }} (*e.g.*, line 9). These are evaluated using the Jinja templating engine [7] and replaced with the evaluation result each time those pods are instantiated. Such expressions have access to information about the volume being managed and to the state of relevant objects such as the corresponding `PVC` and `StorageClass`, allowing pod definitions to be customized to each particular volume.

Using the provisioner. Volumes can be dynamically provisioned by creating `PVC` objects referencing a `StorageClass` that in turn references the `PavProvisioner`. The GCS service account key, required to manage and access buckets, must be stored in a `Secret`—an object intended to hold sensitive data in a key-value mapping—under key `key`. This `Secret` must then be identified through parameters `secretName` and `secretNamespace` specified in the `StorageClass`. Additionally, the GCS project in which to create buckets must be identified through parameter `projectId`. These parameters are available in Jinja expressions under variable `params` (*e.g.*, line 9). Note that different `StorageClass` objects can use different GCS accounts and projects.

Volume creation and deletion. The volume creation pod (lines 8–28) relies on a container image packaging the *gsutil* bucket management tool [6], and mounts the aforementioned secret so that the GCS key is exposed to the container as file `/secret/key` (lines 23–28). It invokes *gsutil* to create a bucket whose name corresponds to the *volume handle*—a string that identifies the volume. The bucket's geographical location (passed on to *gsutil* using flag `-l`) is configurable through a `location` parameter on the `StorageClass`, and defaults to "US" (line 21). Other bucket properties could easily be

```
1  apiVersion: pav.albertofaria.github.io/v1alpha1
2  kind: PavProvisioner
3  metadata: { name: gcs-provisioner }
4  spec:
5    provisioningModes: [ Dynamic, Static ]
6    volumeCreation:
7      capacity: 1Ei
8      podTemplate:
9        metadata: { namespace: "{{params.secretNamespace}}" }
10       spec: &gsutil-pod-spec
11         restartPolicy: Never
12         containers:
13           - &gsutil-container
14             name: gsutil
15             image: albertofaria/gsutil:5.2
16             command: [ gsutil, -o,
17               Credentials:gs_service_key_file=/secret/key,
18               -o, "GSUtil:default_project_id={{
19               params.projectId }}" ]
20             args: [ mb, -b, "on",
21               -l, "{{ params.location or 'US' }}",
22               "gs://{{ defaultVolumeHandle }}" ]
23             volumeMounts:
24               - { name: secret, mountPath: /secret }
25         volumes:
26           - &secret-volume
27             name: secret
28             secret: { secretName: "{{params.secretName}}" }
29   volumeDeletion:
30     podTemplate:
31       metadata: { namespace: "{{params.secretNamespace}}" }
32       spec:
33         <<: *gsutil-pod-spec
34         containers:
35           - <<: *gsutil-container
36             args: [ rm, -r, "gs://{{ volumeHandle }}" ]
37   volumeStaging:
38     podTemplate:
39       metadata: { namespace: "{{params.secretNamespace}}" }
40       spec:
41         restartPolicy: Never
42         containers:
43           - name: gcsfuse
44             image: albertofaria/gcsfuse:0.36.0
45             command: [ /bin/bash, -c ]
46             args:
47               - |
48                 mkdir /pav/volume &&
49                 gcsfuse -o=allow_other \
50                   --key-file=/secret/key --dir-mode=777 \
51                   --temp-dir=/temp --file-mode=666 \
52                   --stat-cache-ttl=0 --type-cache-ttl=0 \
53                   {{ volumeHandle|tobash }} /pav/volume &&
54                 touch /pav/ready &&
55                 sleep infinity
56             securityContext: { privileged: true }
57             volumeMounts:
58               - { name: secret, mountPath: /secret }
59               - { name: temp, mountPath: /temp }
60         volumes:
61           - *secret-volume
62           - { name: temp, emptyDir: {} }
```

Figure 3: Google Cloud Storage integration.

[1] YAML anchors are used to avoid repetition: `&label` labels a value and `*label` reuses that value; `<<: *label` allows overriding fields in mappings.

```
1  apiVersion: pav.albertofaria.github.io/v1alpha1
2  kind: PavProvisioner
3  metadata: { name: crypt-provisioner }
4  spec:
5    provisioningModes: [ Dynamic ]
6    volumeValidation: { volumeModes: [ Block ] }
7    volumeCreation:
8      podTemplate:
9        metadata: { namespace: "{{pvc.metadata.namespace}}" }
10       spec: &cryptsetup-pod-spec
11         restartPolicy: Never
12         containers:
13           - &cryptsetup-container
14             name: cryptsetup
15             image: albertofaria/cryptsetup:2.4.1
16             command: [ /bin/bash, -c ]
17             args:
18               - |
19                 set -o errexit -o pipefail
20                 cryptsetup -q luksFormat \
21                   /volume /secret/passphrase
22                 size="$( blockdev --getsize64 /volume )"
23                 offset="$( cryptsetup luksDump \
24                   /volume --dump-json-metadata |
25                   jq '.segments."0".offset' | tr -d '"' )"
26                 echo "$(( size - offset ))" > /pav/capacity
27             securityContext: { privileged: true }
28             volumeMounts:
29               - { name: secret, mountPath: /secret }
30             volumeDevices:
31               - { name: underlying, mountPath: /volume }
32         volumes:
33           - name: secret
34             secret:
35               secretName: "{{ pvc.metadata.annotations[
36                 'crypt/secretName'] }}"
37           - name: underlying
38             persistentVolumeClaim:
39               claimName: "{{ pvc.metadata.annotations[
40                 'crypt/underlyingClaimName'] }}"
41   volumeDeletion:
42     podTemplate:
43       metadata: { namespace: "{{pvc.metadata.namespace}}" }
44       spec:
45         <<: *cryptsetup-pod-spec
46         containers:
47           - <<: *cryptsetup-container
48             args: [ cryptsetup -q erase /volume ]
49   volumeStaging:
50     podTemplate:
51       metadata: { namespace: "{{pvc.metadata.namespace}}" }
52       spec:
53         <<: *cryptsetup-pod-spec
54         containers:
55           - <<: *cryptsetup-container
56             args:
57               - |
58                 dev={{ volumeHandle|tobash }} &&
59                 cryptsetup open /volume "$dev" \
60                   --key-file /secret/passphrase &&
61                 cp -p "/dev/mapper/$dev" /pav/volume
62   volumeUnstaging:
63     podTemplate:
64       metadata: { namespace: "{{pvc.metadata.namespace}}" }
65       spec:
66         <<: *cryptsetup-pod-spec
67         containers:
68           - <<: *cryptsetup-container
69             args: [ "cryptsetup close
70               {{ volumeHandle|tobash }} || (( $? == 4 ))" ]
```

Figure 4: Transparent encryption middleware.

made configurable by introducing more parameters. Conversely, the volume deletion pod (lines 30–36) removes the bucket underlying a dynamically provisioned volume when its PVC is deleted.

Volume staging. The volume staging pod (lines 38–62) uses an image packaging *gcsfuse*, and mounts both the aforementioned secret and a temporary directory to be used as scratch space (lines 57–62). It invokes *gcsfuse* with several parameters including the name of the bucket underlying the volume, which corresponds to the volume handle. *gcsfuse* then mounts the bucket at `/pav/volume` and launches a daemon process backing the file system. Since the daemon must continue running while the client pod uses the volume, the container then creates a file at `/pav/ready` to indicate to PaV that the volume is ready, and sleeps until it is terminated.

4.2 Transparent Encryption

Kubernetes' volume abstraction enables storage resources to be represented as objects, and allows application pods to use those resources uniformly regardless of the storage infrastructure that underlies them. However, Kubernetes' ability to build and manage that infrastructure itself is more limited. As a concrete example, no general mechanism is offered to enable volume encryption. Instead, to ensure that data stored in a volume is encrypted, this functionality must either be built into all pods that use the volume or implemented by all employed volume provisioners. This lack of modularity leads to increased development effort and hampers the adaptability of the storage infrastructure to the needs of the applications.

We thus use PaV to create a middleware component that relies on the *cryptsetup* disk encryption tool [1] to add encryption capabilities to any existing block volume transparently. The complete definition of the `PavProvisioner` implementing this is reproduced in Figure 4. Briefly, it enables the dynamic provisioning (line 5) of volumes that act as a transparent encryption layer for other existing volumes. Pods can then use the new volumes, and all written (read) data is automatically encrypted to (decrypted from) the underlying volumes. Since *cryptsetup* is limited to block devices, the provisioner only supports block volumes (line 6). Support for file system volumes could be added using stackable, encrypting file systems such as *eCryptfs* [11].

Using the provisioner. To add encryption capabilities to an existing block volume, the passphrase for encryption must first be stored in a `Secret` under key `passphrase`. Then, one creates a PVC with annotations `crypt/secretName` and `crypt/underlyingClaimName` set to the names of the `Secret` and of the PVC corresponding to the existing underlying volume, respectively, and referencing a `StorageClass` that in turn references the `PavProvisioner`. These annotations are retrieved in Jinja expressions from the mapping `pvc.metadata.annotations` (*e.g.*, lines 35–36). The `Secret` and both PVC objects must all exist in the same namespace.

Volume creation and deletion. The creation of the PVC triggers the volume creation pod (lines 8–40), which mounts both the secret and the underlying volume (lines 28–40) and begins by formatting the latter using *cryptsetup* (lines 20–21). Since this marginally decreases the volume's capacity due to the addition of a metadata header, the pod computes the new capacity and exposes it to PaV by writing

it to file `/pav/capacity` (lines 22–26). The contents of encrypted volumes are erased by the volume deletion pod (lines 42–48) when the PVC is deleted.

Volume staging and unstaging. The volume staging pod (lines 50–61) mounts the underlying volume and uses *cryptsetup* to create a block device that provides unencrypted access to it (lines 59–60). It then copies the resulting block special file to `/pav/volume` (line 61) and terminates, as all encryption and decryption is handled by the kernel without the need for a daemon process. It is thus unnecessary to create the `/pav/ready` file. Finally, the volume unstaging pod (lines 63–70) undoes this process.

Discussion. The technique showcased here can be applied to build reusable, transparent middleware components implementing many other storage functionalities, such as compression, caching, or tracing. It may also be generalized to accept more than one underlying volume, enabling the creation of replication or load-balancing layers. By stacking several such components, complex and modular storage architectures can be represented directly as interconnected objects and managed more effectively with Kubernetes.

5 CONCLUSION AND FUTURE WORK

We present PaV, a Kubernetes plugin that simplifies the construction of volume provisioners. Procedures for creating, deleting, staging, and unstaging volumes are specified as pod templates, which are then instantiated as needed. Compared to the alternative of manually implementing the CSI interface and deploying all necessary components, PaV significantly reduces the effort required to integrate storage systems into Kubernetes. Further, it enables the straightforward creation of storage middleware components, improving modularity and unlocking the ability to build advanced storage stacks and architectures using Kubernetes.

We plan to extend PaV's functionality to further broaden its applicability. Examples include the ability to create *volume snapshots*, enabling the creation of other volumes from those snapshots; support for *volume expansion*, which allows the capacity of existing volumes to be increased; and specifying *volume accessibility constraints*, restricting the set of nodes to which pods that use the volumes may be scheduled.

ACKNOWLEDGMENTS

This work was supported by project BigHPC (POCI-01-0247-FEDER-045924), funded by the European Regional Development Fund (ERDF) through the Operational Programme for Competitiveness and Internationalization (COMPETE 2020) and by National Funds through the Portuguese Foundation for Science and Technology (FCT), I.P. in the scope of the UT Austin Portugal Program; and supported by FCT through PhD Fellowship SFRH/BD/146059/2019.

REFERENCES

[1] [n.d.]. *cryptsetup.* https://gitlab.com/cryptsetup/cryptsetup
[2] [n.d.]. *gcsfuse.* https://github.com/GoogleCloudPlatform/gcsfuse
[3] [n.d.]. *Gluster.* https://www.gluster.org
[4] [n.d.]. *Google Cloud Storage.* https://cloud.google.com/storage
[5] [n.d.]. *gRPC.* https://grpc.io
[6] [n.d.]. *gsutil.* https://github.com/GoogleCloudPlatform/gsutil
[7] [n.d.]. *Jinja.* https://palletsprojects.com/p/jinja
[8] [n.d.]. *kopf.* https://github.com/nolar/kopf
[9] [n.d.]. *Kubernetes.* https://kubernetes.io
[10] 2021. *Container Storage Interface (CSI) specification version 1.5.0.* https://github.com/container-storage-interface/spec/blob/v1.5.0/spec.md
[11] Michael Halcrow. 2005. eCryptfs: An Enterprise-class Encrypted Filesystem for Linux. In *Proceedings of the 2005 Linux Symposium.*
[12] Gerry Seidman. 2020. Understanding Kubernetes Storage: Getting in Deep by Writing a CSI Driver. In *Vault '20.* USENIX Association.
[13] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation.* USENIX Association.