www.cockroachlabs.com

Managing State in Kubernetes

In this guide, we'll explore a couple different ways to manage state in Kubernetes, with different trade-offs, and dive into how StatefulSets work with CockroachDB.

Cockroach Labs

The State of State in Distributed Workloads

In the early days, it took great skill (and a lot of work) to successfully run a stateful application on Kubernetes. Pods are ephemeral. That's just one of the tradeoffs that makes them portable and flexible--they can live and die without adverse effects of an overarching function.. Attaching state becomes possible using Daemon sets, but even this proved difficult as you need to carefully coordinate state. It's managing state (constantly updating and handling data changes) that's the hard part.

State and storage in Kubernetes has evolved a lot since those early days. First, Kubernetes changed to allow for storage volumes mounted into the pod, and then for volumes managed directly within Kubernetes. Now, there are a number of ways to manage state and complex, data-intensive workloads in Kubernetes.

In this guide, we'll explore a couple different ways to manage state in Kubernetes, with different trade-offs, and dive into how StatefulSets work with CockroachDB.

Solving for State: applications and databases in Kubernetes

But the magic of orchestration—being able to quickly swap and update containers on the fly—is also a big obstacle to stateful applications and databases. The whole promise of containerization, being able to move and manage quickly and freely, breaks down when an application database is chained to local storage. Why? Because Kubernetes wins through replication. This is how DevOps can build, deploy, scale, and fall back with less effort and more confidence. But replication doesn't work for stateful apps, for a few reasons:

- Database replicas aren't interchangeable like other containers, they have unique states
- Databases require tight coordination with other nodes running the same application to ensure version and other and require careful coordination

Getting Around the problem of state

NoSQL databases like MongoDB and Cassandra are already optimized for clustering, but Kubernetes adopters have three options:

Outside the host, outside the container: Run the DB outside of Kubernetes

`MAXIMUM CHOICE, MAXIMUM EFFORT`

It's simple enough to connect application and database through declaration. The benefit of running it outside on a traditional VM is that you maximize choice. The downside is that you're creating infrastructure redundancy and additional Ops work. Rather than automation, you'll have to manually manage at least five tools already available in K8s.

- Monitoring
- Load Balancing
- Configuration
- Service Discovery
- Logging

Outside the container, inside the host: Rely on Cloud Services

`LESS CHOICE, LESS EFFORT`

It's also possible to leverage cloud services to run your DB outside of K8s. Choosing the DbaaS route eliminates the need for Ops to manage spinning up, scaling, and managing DB, and they're not responsible for a redundant infrastructure stack. It's an external service, but doesn't add the redundancy of a full DB stack.

The downside is you're stuck with the DBaaS as offered by your cloud services provider, which makes even less sense for those running things in house or on prem. And since you don't have direct access to the infrastructure running the DB, fine-tuning performance and managing compliance can be an issue.

3 Inside the container: Run the DB in Kubernetes

CLOUD-NATIVE SIMPLICITY, DEVOPS AGILITY

Since its inception, the Kubernetes community has worked tireless to solve some of the existential challenges presented by achieving maximum fluidity in a world still mostly run on persistent storage. How do you maintain the seamless flexibility of distributed pods in cases where state demands an application and database stay connected even as pods are added, subtracted, or restarted?

Kubernetes provides two native paths to get there. StatefulSets or DaemonSets. StatefulSets assign a unique ID that keeps application and database containers connected through automation. DaemonSets create a 1:1 relationship where a pod runs on all the nodes of the cluster. When a node is added or removed from a cluster, the pod is also automatically added or removed.

We'll look at both now in a little more detail.

Up Close: StatefulSets

Kubernetes excels in orchestration. But how do you accommodate stateful applications' need for persistency connectivity in a world where infrastructure is designed to be disposable and replication rules?

The idea is fairly simple. By assigning each pod a unique network ID, the application and database can maintain connection no matter which node they're assigned to. It also means that as an application is scaled up and down, connections are maintained and persistency achieved.

Secondly, StatefulSets also solves for database requirements around ordered deployment, scaling, and deletion. This is absolutely necessary for the "ordered and graceful" deployment, scaling ,and deletion needs of application databases on persistent storage. These IDs are assigned via an easy to build Headless Service, that binds a DNS record directly to the node, not a clusterIP.

What about performance and resource competition?

Because Kubernetes and the databases are both running on the same machine, you'll take a slight performance hit. Our testing showed ~5%. And by declaring specific resource requirements (in this case CPU/RAM) per container, you can help the scheduler prioritize pod placement and avoid further performance constraints. Consider this a best practice when preparing for StatefulSet support.

Up Close: DaemonSets

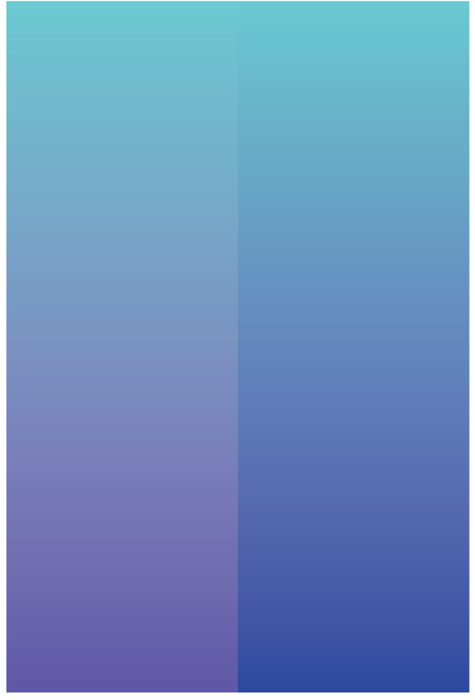
Where StatefulSets balance the need for persistence with the ability to reschedule across a cluster, DaemonSets let you designate a specific set of dedicated nodes to run your database. You can also dedicate these nodes to the database, reducing some of the performance worries of StatefulSets. This makes this an ideal approach for applications like logging and monitoring.

StatefulSets (as of the publication of this guide and K8s version 1.18) support for local storage is still in beta. By contrast, DaemonSets are much more reliable consumers of local disk storage, since you're not worried about lost disks due to rescheduling. But relying on local disks also reduces the ability to automate failover and recovery.

What about performance and resource competition?

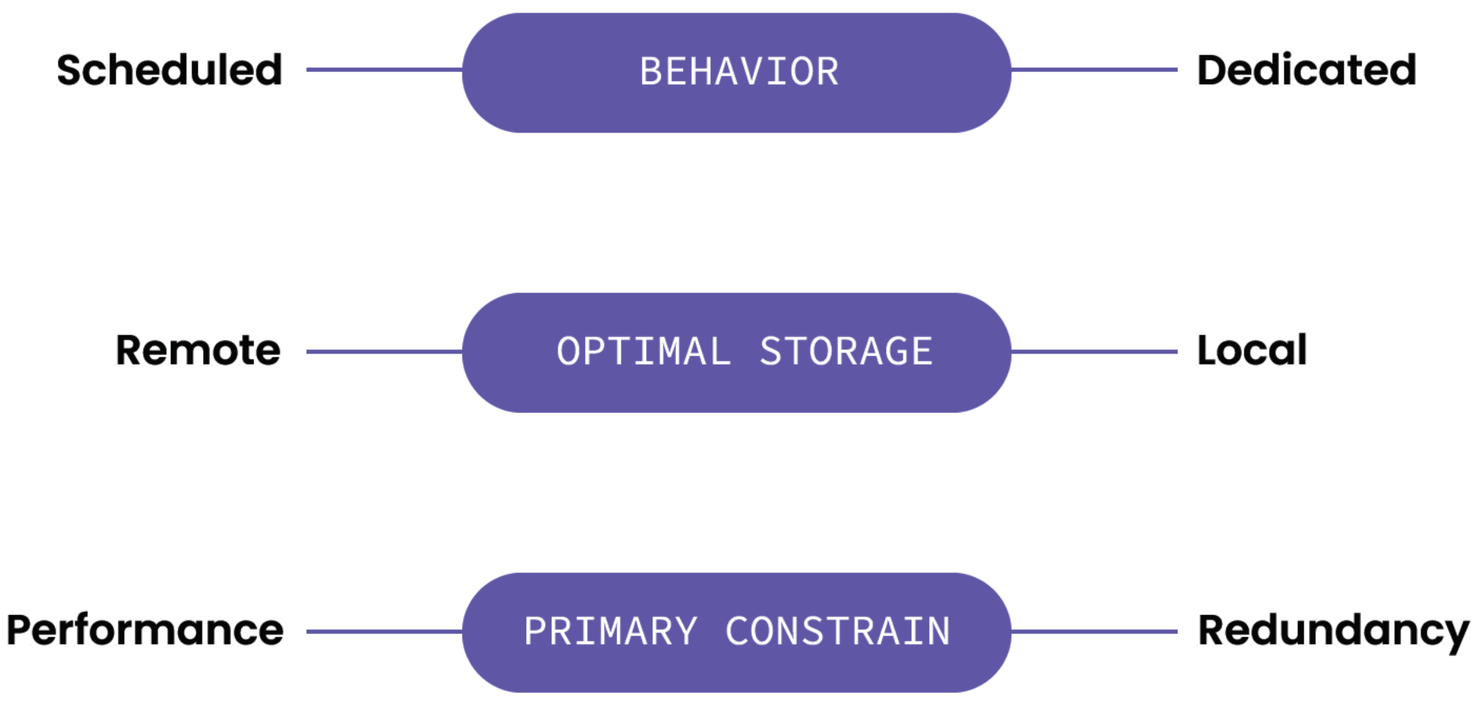
By restricting nodes to database support, DaemonSets eliminate the potential performance issues of StatefulSets caused by resource contention and competition.

Deciding between StatefulSets and DaemonSets for running CockroachDB inside Kubernetes depends mostly on infrastructure needs and priorities. It also depends on how comfortable you are running Kubernetes, although documentation is available to assist for both approaches. DaemonSets carry additional complexity because of the absence of automatic scheduling.



StatefulSets

DaemonSets



Our recommendation is StatefulSets, and we'll look at an ideal deployment in a bit. But if DaemonSets is the right answer, please see our resources page to learn more about getting started.

Assessing Application Changes and Constraints

CockroachDB largely supports the PostgreSQL dialect of SQL, and drivers.

- If your migrating an existing app to CockroachDB, read our SQL feature support page to see what changes you'll have to make to your DML
- For those moving from PostgreSQL, read our guidance on simplifying your migration

To test drive your application, read more about quickly creating a simple, single-node cluster. If you just want to run through some basic simulations, try a CockroachDB load generator.

Hardware Configuration

Test and tune your hardware setup before moving to production to ensure enough CPU, RAM, network, and storage capacity.

At minimum, each node should have 2GB RAM and one entire core. These minimums will rise with workload complexity and concurrency. NOTE: Avoid burstable/shared core VMs that limit the load on a single core.

- Use SSD, not HDDs. You can use a simple configuration declaration via YAML file to modify the Kubernetes StorageClass API object.
- Use larger/more powerful nodes. Adding more CPU is typically more beneficial than adding more RAM.

For best resilience:

- Use many smaller nodes instead of fewer larger ones. This speeds recovery.
- Use zone configs to increase the replication factor from the default 3 to 5. **This is especially recommended if you are using local disks rather than a cloud providers' network-attached disks.** You can do this for the entire cluster or for specific databases or tables.

For more details, see our **Hardware recommendations**.

Additional Software

CockroachDB requires nodes to be somewhat synchronized, so we recommend running a service like NTP on each node. If you're using a hosted Kubernetes solution, you can skip this.

Optimizations

The following optimizations ensure CockroachDB and Kubernetes to work together with greater efficiency.

Disk Type

To create the SSD storage object, create a new StorageClass request.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
name: fast
provisioner: kubernetes.io/gce-pd
parameters:
type: pd-ssd
allowVolumeExpansion: true
```

For environments other than GCE, see **appropriate Storage Class** for your platform for help updating the provisioner and parameters.

Now define the fast StorageClass in your volumeClaimTemplates. Spec object.

```
accessModes:
- ReadWriteOnce
storageClassName: fast
resources:
requests:
storage: 30Gi
```

Disk Size

For some cloud providers (notably including all GCP disks and the AWS io1 disk type), the number of IOPS available to a disk is directly correlated to the size of the disk.
In such cases, increasing disk size can improve CockroachDB performance. Doing this is easy – before you create your CockroachDB cluster, modify the volumeClaim-Template in the CockroachDB yaml file to request more space.

Resource Requests and Limits

We talked earlier about configuring resource limits for database pods. This ensures they always have enough resources when/if they get rescheduled. Do this by adding a resource request object to your Cockroachdb container.

```
accessModes:
- ReadWriteOnce
storageClassName: fast
resources:
requests:
...
```

For more detailed help, see our **cluster performance optimization guide**.

Starting and Running Kubernetes

This guide assumes you already have Kubernetes. If you need help with general configuration, please see documentation on Picking the right solution.

Configure StatefulSet

To simplify the task, CockroachDB provides a yaml file that starts with a 3-node cluster, with each node running on a separate K8s node.

You can download this from GitHub.

For the sake of expediency, we provide a yaml file that starts a 3-node CockroachDB cluster, which each node running on a separate Kubernetes node, **which you can download from GitHub**.

Selecting Number of Replicas

Inside the yaml file, replicas represents the number of CockroachDB nodes you want to run. The default is set to three (the minimum for fault-tolerance), but it can easily be increased later.

Initializing StatefulSet

Now it's time to create the Kubernetes nodes. From your local workstation, point kubectl to the provided yaml file. If you've modified it, use a local copy:

```
$ kubectl create -f https://raw.githubusercontent.com/
cockroachdb/cockroach/master/cloud/kubernetes/cockroachdbstatefulset-
secure.yaml
```

In production, SSL encryption controls which clients can connect and secures communication between nodes. It's easy to do but requires manual certificate approval via certificate signing request.

1 Get CSR for POD 1

For some cloud providers (notably including all GCP disks and the AWS io1 disk type), the number of IOPS available to a disk is directly correlated to the size of the disk.
In such cases, increasing disk size can improve CockroachDB performance. Doing this is easy – before you create your CockroachDB cluster, modify the volumeClaim-Template in the CockroachDB yaml file to request more space.

```
$ kubectl get csr


Name                                                   Condition          Age
default.node.cockroachdb-0                             Pending            1m
system:serviceaccount:default:default

node-csr-0Xmb4UTVAWMEnUeGbW4KX1oL4XV_LADpkwjrPt       Approved, Issued   4m
QjlZ4 kubelet

node-csr-NiN8oDsLhxn0uwLTWa0RWpMUgJYnwcFxB984mw       Approved, Issued   4m
jjYsY kubelet

node-csr-aU78SxyU69pDK57aj6txnevr7X-8M3XgX9mTK0       Approved, Issued   5m
Hso6o kubelet
```

If you don’t see a Pending CSR, wait a minute and try again.

2 Examine Pod 1 CSR

```
$ kubectl describe csr default.node.cockroachdb-0


Name: default.node.cockroachdb-0
Labels: <none>
Annotations: <none>
CreationTimestamp: Thu, 09 Nov 2017 13:39:37 -0500
Requesting User: system:serviceaccount:default:default
Status: Pending
Subject:
Common Name: node
Serial Number:
Organization: Cockroach
Subject Alternative Names:
DNS Names: localhost
Cockroachdb-0.cockroachdb.default.svc.cluster.local
Cockroachdb-public
IP Addresses: 127.0.0.1
10.48.1.6
Events: <none>
```

3 Approve Pod 1 CSR

If everything looks correct, the CSR can be approved.

```
$ kubectl get csr


$ kubectl certificate approve default.node.cockroachdb-0
$ kubectl describe csr default.node.cockroachdb-0
certificatesigningrequest "default.node.cockroachdb-0" approved
```

4 Approve requests for other pods

Repeat steps 1-3 for additional pods.

Initialize the Cluster

After verifying pods and disks, you can init the cluster.

1 Verify pods and disks

But first, verify pods look as you expect:

```
$ kubectl get pods

NAME              READY   STATUS    RESTARTS   AGE
cockroachdb-0     1/1     Running   0          2m
```

Then do the same for disks:

```
$ kubectl get pods

NAME                                     CAPACITY  ACCESSMODES  RECLAIMPOLICY  STATUS  CLAIM REASON                   AGE
pvc-52f51ecf-8bd5-11e6-a4f4-42010a800002  1Gi       RWO          Delete         Bound   default/datadir-cockroachdb-0  26s
pvc-52fd3a39-8bd5-11e6-a4f4-42010a800002  1Gi       RWO          Delete         Bound   default/datadir-cockroachdb-1  27s
pvc-5315efda-8bd5-11e6-a4f4-42010a800002  1Gi       RWO          Delete
```

2 Generate Init Pod

Once you've verified everything, you can create a special pod to initialize via one-time init command. This pod will be deleted as soon as the cluster starts.

1. Run the following command to generate the Init pod:

```
$ kubectl create -f https://raw.githubusercontent.com/
cockroachdb/cockroach/master/cloud/kubernetes/cluster-init-secure.yaml
```

2. Approve the CSR for the one-time initialization pod:

```
$ kubectl certificate approve default.client.root

certificatesigningrequest "default.node.cockroachdb-0" approved
```

3 Verify Pod is Running

Confirm that cluster initialization has completed successfully:

```
$ kubectl get job cluster-init-secure

NAME                       DESIRED     SUCCESSFUL     AGE
cluster-init-secure        1           1              19m
```

Next up: Create Your Schema ⟶

Create Your Schema

The simplest way to put your schema on your database is to connect to your CockroachDB cluster's SQL shell where you can access CockroachDB SQL. To do this, use our client-secure.yaml file to launch a pod and keep it running indefinitely.
From your local machine, run:

```
$ kubectl create -f https://raw.githubusercontent.com/
cockroachdb/cockroach/master/cloud/kubernetes/clientsecure.yaml

pod "cockroachdb-client-secure" created
```

Since the pod uses the root client certificate created earlier to initialize the cluster, no CSR approval is required.

Get a shell into the pod and start the CockroachDB built-in SQL client:

```
$ kubectl exec -it cockroachdb-client-secure -- ./cockroach sql
--certs-dir=/cockroach-certs --host=cockroachdb-public
```

From here, you can execute the DDL statements required to configure your schema.

```
# Welcome to the cockroach SQL interface.
# All statements must be terminated by a semicolon.
# To exit: CTRL + D.
#
# Server version: CockroachDB CCL v1.1.2 (linux amd64, built
2017/11/02 19:32:03, go1.8.3) (same version as client)
# Cluster ID: 3292fe08-939f-4638-b8dd-848074611dba
#
# Enter \? for a brief introduction.
#
root@cockroachdb-public:26257/>
```

Connect Your Application

With everything in place, it's time to connect your client application to your CockroachDB cluster. Since SSL encryption will require your application to have its own certificate, we'll use an init container to get the certificate and store it in a client-certs storage volume. This will enable secure signing going forward.

Add the following to your application spec:

```
volumes:
- name: client-certs
emptyDir: {}
initContainers:
- name: init-certs
image: cockroachdb/cockroach-k8s-request-cert:0.3
imagePullPolicy: IfNotPresent
command:
- "/bin/ash"
- "-ecx"

- "/request-cert -namespace=${POD_NAMESPACE} -certsdir=/
cockroach-certs -type=client -user=root -symlink-ca-from=/
var/run/secrets/kubernetes.io/serviceaccount/ca.crt"
env:
- name: POD_NAMESPACE
valueFrom:
fieldRef:
fieldPath: metadata.namespace
volumeMounts:
- name: client-certs
mountPath: /cockroach-certs
volumes:
- name: client-certs
emptyDir: {}
initContainers:
- name: init-certs
image: cockroachdb/cockroach-k8s-request-cert:0.3
imagePullPolicy: IfNotPresent
command:
- "/bin/ash"
- "-ecx"
```

To use a CockroachDB user other than root, change-user=root to --user=<your-user>. This requires you to already have created the user in CockroachDB and given them any necessary privileges.

And then to that spec's containers:

```
volumeMounts:
- name: client-certs
mountPath: /cockroach-certs
```

To see an example of an entire application yaml file, see our example on GitHub. With that set, you can use the following connection string:

```
postgres://root@cockroachdb-public:26257/kv?sslmode=verifyfull&
sslcert=/cockroach-certs/client.root.crt&sslkey=/
cockroach-certs/client.root.key&sslrootcert=/cockroach-certs/ca.crt
```

Set Up Monitoring

Continuous monitoring is the key to good infrastructure. While we recommend Prometheus, a member of the Cloud Native Computing Foundation, you can bring your own K8s monitoring tool to the task. While it takes a little work to get the tool integrated, if you plan to keep your cluster around the effort pays off.

For more help see the **CoreOS Prometheus operator repo**.

Admin UI

For a quick view into your cluster, you can use CockroachDB's built-in Admin UI by port-forwarding from your local machine to one of the pods:

```
$ kubectl port-forward cockroachdb-0 8080

Forwarding from 127.0.0.1:8080 -> 8080
```

The command must be run on the same machine as the web browser in which you want to view the Admin UI.

If you have been running these commands from a cloud instance or other non-local shell, you must first configure kubectl locally and run the above portforward command on your local machine.

You can now view the CockroachDB Admin UI from https:// localhost:8080. See our admin UI documentation for more help on using it.

What's next?

Now with the application running connected to your CockroachDB, it's time to start simplifying your stack. One by one, you can remove the duplicative tools and services you're now able to leverage natively inside Kubernetes.

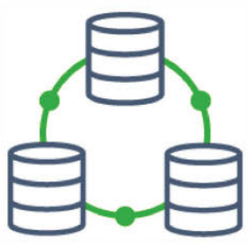
Congratulations, you're now one step closer to being fully cloud native.

-- Team Cockroach Labs

CockroachDB

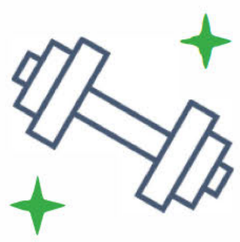
K8s + CockroachDB = Effortless App Deployment

Run your application on the cloud-native database uniquely suited to Kubernetes.



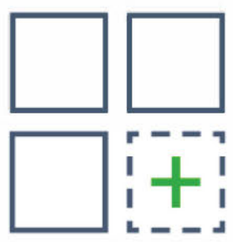
Scale elastically with distributed SQL

Say goodbye to sharding and time-consuming manual scaling.



Survive anything with bullet-proof resilience

Rest easy knowing your application data is always on and always available.



Build fast with PostgreSQL compatibility

CockroachDB works with your current applications and fits how you work today.

Get started for free today

cockroachlabs.com/k8s

Trusted by innovators

COMCAST SPACEX BOSE LUSH rubrik wework