ChatGPT

Feasibility of a Minimal GitOps-Driven Kubernetes Cluster on DigitalOcean

Overview

This feasibility study examines a **minimal GitOps-first Kubernetes stack** that can start on a single $20/month DigitalOcean droplet and grow into a production-grade cluster. The proposed design uses **ArgoCD for GitOps**, a lightweight Kubernetes distribution (Rancher's **K3s**), and external **S3-compatible storage** for backups and persistent data. The goal is to minimize upfront cost and complexity while ensuring a clear path to **horizontal/vertical scaling**, high availability (HA), and robust operations. We evaluate the technical feasibility, integration challenges, and recommend a stack with example configurations (K3s installation, ArgoCD sync config, Velero backup to S3, CSI driver setup) to meet these requirements.

GitOps-First Approach with ArgoCD

GitOps is the cornerstone of this design, using Git as the single source of truth for cluster state and applications. **Argo CD** (a CNCF project) continuously monitors the Git repository and **syncs the cluster state to match the declared manifests** [1] [2]. This ensures that all applications and infrastructure components are version-controlled, auditable, and can be restored or rolled back if needed.

ArgoCD works by declaring Applications (custom resources) that point to a git repo/path and target cluster. It flags any drift between the live cluster state and the Git repo as "OutOfSync" and can automatically pull the changes to reconcile state [2]. In our setup, ArgoCD itself will run inside the cluster and manage all other components (including its own upgrades or config) via GitOps. This **bootstrap problem** (installing ArgoCD before it can do GitOps) is solved by the initial script, which will install ArgoCD on the fresh cluster and configure it to point at the repository. In practice, many ArgoCD users employ an "**App of Apps**" bootstrap pattern – creating a parent Application that in turn deploys all other apps and configs in the cluster [3]. For example, a parent ArgoCD Application might sync a `clusters/dev` folder in the repo that contains K8s manifests or Helm charts for components like ingress, cert-manager, CSI driver, etc. This allows one Git commit to declaratively define the entire cluster add-ons.

GitHub API Token Usage: The GitHub token is used to grant ArgoCD read access to the config repo (if private). The bootstrap script can create a Kubernetes Secret in ArgoCD's namespace with the token (or better, a deploy key) and configure ArgoCD's repository credentials to use it. Once set, ArgoCD will pull the manifests from GitHub using this token. All cluster changes are made via Git commits, aligning with GitOps best practices (auditability and easy rollback).

Example – ArgoCD Application Manifest: An example ArgoCD App definition that could be applied post-bootstrap might look like:

```
# ArgoCD Application pointing to the GitOps repo
apiVersion: argoproj.io/v1alpha1
```

```
kind: Application
metadata:
  name: cluster-bootstrap
  namespace: argocd
spec:
  project: default
  source:
    repoURL: https://github.com/YourUser/your-gitops-repo.git
    targetRevision: main
    path: clusters/dev
  destination:
    server: https://kubernetes.default.svc
    namespace: default
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
```

In this manifest, ArgoCD will auto-sync the contents of the `clusters/dev` folder in the repo into the cluster (pruning any removed resources and correcting drift automatically). The initial script can apply such a manifest (or one for the parent "app of apps") to kick off the GitOps synchronization.

Lightweight Kubernetes on a Single Droplet (K3s)

To keep the footprint small and performance overhead low, the system uses **K3s**, a lightweight Kubernetes distribution by Rancher. K3s is a CNCF-certified Kubernetes that omits non-essential extras and is packaged as a single <100MB binary [4] [5]. It is designed for minimal resource environments (edge, IoT, CI) yet **"fully certified by the CNCF, highly available and production-ready"** [5]. This makes it ideal for a $20/month droplet (which, for example, provides 2 vCPUs and 4GB RAM) to run both the control plane and initial workloads.

Architecture of a single-server K3s cluster: one server node (left) runs the Kubernetes control plane (API server, controllers, scheduler) backed by an embedded SQLite datastore, and multiple agent (worker) nodes can optionally connect to it for running workloads. In a one-node setup, the server also runs workload pods. This minimal cluster can later be expanded to multiple servers for high availability, or additional agents for more capacity.

K3s Installation: K3s dramatically simplifies Kubernetes installation. The bootstrap script will provision a new Ubuntu droplet and install K3s in one step. For example, using the official K3s install script:

```
# Install latest K3s (single-node server)
curl -sfL https://get.k3s.io | sh -
```

This command installs K3s with default parameters (it runs a single-node Kubernetes master with **embedded SQLite** datastore by default). On startup, K3s will also deploy a default service load-balancer (for bare-metal), a basic ingress (Traefik Proxy), and a **local path storage** provisioner for persistence. These defaults mean our one-node cluster is immediately functional: we have a working ingress controller to expose services, and a way to fulfill PersistentVolumeClaims using local disk.

Feasibility on $20 Droplet: A 4 GB RAM droplet can comfortably run the K3s control plane (which is lightweight, using ~512MB or less in idle state) and a few essential services like ArgoCD. For instance, ArgoCD's components (API server, repo server, controller, UI) might use a few hundred MB of RAM collectively. That leaves resources for user workloads. K3s's efficiency (half the memory of standard Kubernetes) helps maximize what we get out of a small VM [6]. In terms of CPU, 2 vCPUs are sufficient for moderate control-plane operations and light app workloads; as load grows, we can scale up. In testing, K3s clusters with 4GB RAM can run dozens of low-traffic pods comfortably, though heavy builds or large database workloads would require more resources or additional nodes.

Transition to Full K8s: K3s **is** real Kubernetes, so no app changes are needed to "graduate" to a full cluster. When scaling up, we have two options: *(a)* **Add more K3s nodes** (which can be done seamlessly using the K3s token and join commands), or *(b)* **Migrate to a managed Kubernetes** service (like DigitalOcean Kubernetes) and re-deploy the GitOps repo there. Because everything is defined in Git and ArgoCD is managing it, migrating to another cluster is mostly a matter of installing ArgoCD there and pointing it to the same repo. This portability is a big advantage of the GitOps approach. In summary, using K3s now does not paint us into a corner later – it's a production-grade distro that can either scale out (by enabling HA mode with etcd) or be swapped out for another Kubernetes with minimal disruption.

Persistent Storage and Dynamic Volume Provisioning

Even a minimal cluster needs to handle **persistent data** for stateful workloads. The design must support dynamic provisioning of volumes (so that when an application claims storage, it's provided without manual intervention).

K3s Local Path Provisioner: Out-of-the-box, K3s includes Rancher's **Local Path Provisioner**, which allows creating PersistentVolumeClaims (PVCs) backed by the node's local disk [7]. The default `StorageClass` named "local-path" writes data under `/var/lib/rancher/k3s/storage/` on the droplet. This satisfies basic persistence needs with zero additional setup – for example, a simple PVC request will allocate a directory on the node's filesystem to that pod [8] [9]. However, **local-path storage is tied to the single node** – if the droplet is destroyed or the pod moves to another node (in a multi-node future), the data doesn't follow. It's fine for initial testing or ephemeral use, but not ideal for durable production data.

DigitalOcean Block Storage (CSI Driver): For long-term scalability, we integrate DigitalOcean's Block Storage via their CSI driver. DigitalOcean volumes are network-attached disks that can be dynamically created, attached to the droplet, and moved between droplets. The **DO CSI plugin** is an open-source Kubernetes driver that lets the cluster manage DO volumes for PVCs [10] [11]. It's the same mechanism used in DigitalOcean's managed Kubernetes service. Installing it is straightforward: we apply the CSI driver manifests (which include necessary CustomResourceDefinitions and driver deployments). For example:

```
# Install DigitalOcean CSI driver (use latest version matching K8s version):
kubectl apply -f https://raw.githubusercontent.com/digitalocean/csi-
digitalocean/v4.13.0/deploy/kubernetes/releases/csi-digitalocean-v4.13.0/
{crds.yaml,driver.yaml,snapshot-controller.yaml}
```

This registers a `StorageClass` called **do-block-storage** by default [12]. Thereafter, any PVC using `storageClassName: do-block-storage` will trigger Kubernetes to call the DO CSI driver, which in

turn creates a new DO Volume via the DO API (using the provided token) and attaches it to the node where the pod is scheduled. This enables **dynamic volume provisioning** backed by durable storage (DO volumes are replicated within the region and survive droplet reboots or replacement). For example, if an app requests a 5Gi PVC, the CSI will provision a 5Gi volume in the same region and attach it to the droplet [13] [14].

Integration notes: The bootstrap script can apply the CSI manifests directly, or the GitOps repo can contain them for ArgoCD to apply on first sync. Either way, the **DigitalOcean API token** is needed by the CSI controller to create volumes; this is typically provided via a Secret. The DO CSI project provides a YAML for a Secret where you put the API token, which the driver reads. We will ensure the script creates this Secret (or provides it via helm values if using a Helm chart for the CSI driver). Once set up, this system is fairly seamless – e.g. installing a MySQL Helm chart that requests storage will automatically get a DO volume provisioned and attached to the pod.

Alternatives: In scenarios where external block storage costs must be avoided initially, one could rely on the local-path provisioner (no additional cost) or use **Longhorn** (an open-source distributed storage by Rancher) which is compatible with K3s [15] [16]. Longhorn runs inside the cluster and replicates volumes across multiple nodes, providing resilience without cloud provider support. However, Longhorn would consume extra resources on our single small droplet and complicate the setup. Given that DO block storage is relatively inexpensive (and we only pay for what we use in GB) and offloads the heavy lifting to DO's infrastructure, the CSI approach is a good balance for production readiness. We prefer enabling DO's managed storage early, as it will ease the **transition to multi-node** (volumes can detach from one droplet and attach to another, so stateful pods can be rescheduled to new nodes over time).

Object Storage and Backups (Snapshots to S3)

For persistent data management, we require not only volumes but also **snapshots and backups** in case of disaster or migration. The design calls for an **external S3-compatible object store** to hold backups, because it's off-cluster (so backups survive cluster loss) and cost-efficient (only pay for storage used). This can be **DigitalOcean Spaces**, **Backblaze B2**, **Wasabi**, or even self-hosted **MinIO**. The system will use **Velero** for cluster backups:

- **Velero** is an open-source tool purpose-built for Kubernetes backups. It backs up **Kubernetes object state** (all cluster resources like Deployments, Services, etc.) into a tarball and stores it on S3, and can also backup volumes either by snapshotting them or by streaming their content to the backup (using Restic) [17]. In a restore, Velero can recreate all K8s objects from the backup file and restore persistent volume data from snapshots or backup files. This is crucial for disaster recovery – one could bootstrap a new cluster and use Velero to restore everything to a last known state.

- **Back up to S3**: We will configure Velero with an S3 bucket as the backup target. Velero has a built-in AWS S3 plugin which works for any S3-compatible service (Spaces, Wasabi, B2, etc.) [18] [19]. We simply provide the S3 endpoint and credentials. For example, using DigitalOcean Spaces (which has an S3-compatible API), we might run:

```
velero install \
  --provider velero.io/aws \
  --plugins velero/velero-plugin-for-aws:v1.5.0,digitalocean/velero-
plugin:v1.2.0 \
```

```
  --bucket <YOUR_BUCKET_NAME> \
  --backup-location-config s3Url=https://
<region>.digitaloceanspaces.com,region=<region> \
  --secret-file=./credentials-velero
```

In this command, we use the AWS provider (`velero.io/aws`) so that Velero knows how to talk to an S3 API, and specify two plugins: the AWS plugin (for general S3 functionality) and the DigitalOcean plugin (for volume snapshots) [20]. We point it to our Spaces bucket and region. The `credentials-velero` file contains the Spaces access key and secret in AWS format. We also pass `--use-volume-snapshots=false` initially, meaning that to start, we'll rely on file-level backup for PVs (Velero will use Restic to copy volume contents to the bucket) unless we enable snapshots explicitly.

- **Enabling Volume Snapshots:** A powerful feature is the ability to take cloud snapshots of volumes. The **DigitalOcean Velero plugin** allows Velero to create on-demand snapshots of DO Block Storage volumes and include the snapshot references in the backup [21] [22]. Snapshots are nearly instantaneous and avoid transferring all data over the network (DO does the snapshot internally). To enable this, after installing Velero we would:
- Create a VolumeSnapshotLocation in Velero pointing to DigitalOcean's API (Velero CLI: `velero snapshot-location create default --provider digitalocean.com/velero`) [23].
- Patch the Velero deployment to provide the DO API token to the snapshot plugin (the DO plugin needs the token to call the DO snapshot API) [24] [25].

Once set up, `velero backup create` commands can snapshot volumes (with `--snapshot-volumes` flag), and those snapshots will be listed in the DO control panel [26] [27]. During a restore, Velero can either recreate a new volume from the snapshot or restore file-by-file from the backup, depending on configuration.

- **External vs. Internal Object Store:** Using an external cloud storage (Spaces, B2, Wasabi) is recommended for reliability. For instance, DigitalOcean Spaces in the same region offers low-latency and high durability storage for backups. Wasabi or Backblaze B2 might reduce costs further (they are known to work with Velero via the S3 API [18]). The main cost is based on GB stored (and possibly egress on restore). These services are relatively low-cost (B2 is ~$5/TB/month, Wasabi ~$6/TB but no egress fees, DO Spaces $5/month for 250GB). Given our cluster is small, backup sizes will be modest (maybe a few GB for cluster state and database files), making this very affordable.

As an alternative, **MinIO** could be deployed on the cluster to serve as an S3 endpoint. This would avoid external service costs, but it introduces additional considerations: MinIO itself would need a persistent volume to store the backups. Likely we'd give MinIO a large DO volume. If the entire droplet or cluster goes down, we'd still need to access that volume's data to recover backups (meaning manual intervention to attach it to a new server). This is doable but not as seamless as having backups already off-site. Therefore, while MinIO-on-cluster is feasible (especially for dev/test environments), for true disaster recovery robustness we favor using an external S3 service for backups.

Example – Velero on DO Spaces: After running the above `velero install`, backups can be created with commands like `velero backup create full-backup --include-namespaces '*' --snapshot-volumes=true`. This would upload all Kubernetes objects to Spaces and snapshot any PVC volumes via the DO API. The snapshot lives in your DO account (with associated cost per GB per month), and the metadata goes to Spaces. The bootstrap script can also schedule backups (Velero supports scheduled backups via Cron), or this can be configured via ArgoCD (since Velero's schedule objects are just Kubernetes resources).

Restore-from-Snapshot in Bootstrap: The requirements mention the script should support restore-from-snapshot. In practice, a restore might involve: creating a new droplet/cluster, installing K3s, installing Velero, and then running `velero restore` to pull everything from the backups. We can automate much of this. For example, the script could accept a backup name or timestamp, and after cluster creation, it would download the Velero CLI, configure credentials, and perform a restore automatically. If using DO volume snapshots, we'd also need to ensure those snapshots are present and the DO token has access. Velero will handle creating new volumes from snapshots if the plugin is configured. This means a **one-command disaster recovery** is realistic: as long as the backup files and snapshots exist on S3/DO, the script can recreate the cluster and use Velero to put it back to the backed-up state.

Bootstrapping Process and Automation

One key goal is a **self-contained bootstrap script** that, given only a GitHub token and a DO API token, can **provision the entire system from scratch**. Here's a breakdown of how the bootstrap automation will work and the feasibility of each step:

1. **Provisioning the Droplet:** Using the DigitalOcean API (or CLI `doctl`), the script creates a new droplet (for example, Ubuntu 22.04 x64, in a chosen region). The DO API allows injecting **cloud-init user data** on creation – we leverage this to automate installation. We include our shell commands in the user-data so that as soon as the droplet boots, it runs them. This means no manual SSH is needed. The DO API token is used for this call and to tag the droplet or configure networking (e.g., if we want to attach a floating IP or set up a firewall). The droplet size will be the $20 plan (to start), and we can allow the user to override region or size via script parameters if needed. Creating a droplet via API is quick (usually 60-90 seconds to boot). The script can poll the DO API to know when the droplet is active.

2. **Installing K3s:** In the cloud-init script (or via an SSH command from the script once the droplet is up), we run the K3s installer. For example:

```
#cloud-config
runcmd:
  - curl -sfL https://get.k3s.io | INSTALL_K3S_EXEC="--write-kubeconfig-mode
644" sh -
```

This ensures K3s is up and running. We set `--write-kubeconfig-mode 644` so that the generated kubeconfig (`/etc/rancher/k3s/k3s.yaml`) is readable by non-root, which can be useful if subsequent commands run as a different user or via automation fetching the kubeconfig. After installation, we effectively have a one-node Kubernetes cluster ready to accept `kubectl` commands. The script will export the kubeconfig (or copy it) so that it can use `kubectl` to configure the cluster in subsequent steps.

1. **Installing ArgoCD:** With Kubernetes running, the next step is to deploy ArgoCD. This can be done by applying the official ArgoCD manifest YAML (which sets up the `argocd` namespace and all ArgoCD components). For example:

```
kubectl create namespace argocd
kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml
```

This will spin up ArgoCD (which itself might take a minute or two to get all pods running). ArgoCD's API/UI service by default is ClusterIP – since we have Traefik ingress (from K3s) we could also deploy an Ingress resource to expose the ArgoCD UI (and CLI server) externally, or map it to a DO LoadBalancer. For a minimal setup, we might skip exposing the UI publicly (the user can SSH port-forward if needed), but it's convenient to have access. We can use the DO token to allocate a Load Balancer via the Kubernetes cloud controller, but since this is a single node, a simpler approach is to use the node's IP and an Ingress. The GitOps repo could contain an ArgoCD ingress manifest (which ArgoCD would apply to expose itself). In any case, ArgoCD is now running in the cluster.

1. **Configuring GitOps Sync:** Now we need to tell ArgoCD about the Git repository and what to sync. This can be done in two ways:
2. Programmatically, using ArgoCD's CLI or API. For instance, the script could use `argocd login` (port-forward to ArgoCD API) and `argocd app create` commands to define the root application pointing to the repo. This requires the ArgoCD admin password (which we can get from a Kubernetes secret created during install). This approach is a bit involved but doable.
3. Declaratively, by applying an `Application` manifest (as shown earlier) to the cluster. Since ArgoCD is installed, applying an `Application` CR will cause ArgoCD to pick it up. We might apply a "bootstrap" Application that points to the repo. This manifest can include the GitHub token if necessary (ArgoCD supports putting credentials in a Secret and referencing them in the Application spec or via a secret resource).

Using the declarative approach aligns with GitOps – we could even embed the initial bootstrap Application manifest in the user-data so that it's created immediately after ArgoCD comes up. Alternatively, we include it in the Git repo and have the script create it with `kubectl`. Both ways are feasible. We prefer the **App-of-apps pattern** for scalability: e.g., the script creates a parent ArgoCD Application (named "cluster-bootstrap") that points to something like `repo:myrepo, path: /clusters/single-node`. In that repo path, the user can have a Kustomize or Helm chart that defines all other components (ingress, cert-manager, etc.). ArgoCD will then synchronize those automatically. This means after bootstrap, **all further configuration is driven from Git.** The script's job is essentially done once ArgoCD is pulling from the repo.

1. **Deploying Storage and Backup Components:** Many of the add-ons (CSI driver, Velero, maybe cert-manager, etc.) can actually be deployed via ArgoCD if their manifests/Helm charts are in the Git repo. However, there's a chicken-and-egg concern for storage: if ArgoCD itself needed a PVC (it doesn't by default) or if any early component needed storage before CSI is installed, there's a dependency. In our case, ArgoCD doesn't require a PV for core functionality (it stores its state in Kubernetes API itself), so we can safely deploy the CSI driver via GitOps. Velero also can be deployed via GitOps (or via script). We might choose to install the **DO CSI driver via the Git repo** (so ArgoCD applies it). This ensures it's part of the managed configuration. The script would then *not* create it, avoiding duplicate effort. The only caveat: until CSI is up, any PVCs for other apps will be pending. But we can control ordering with sync waves or just accept that ArgoCD will install CSI first (assuming the manifest is in an early apply wave or no one requests a PVC until it's ready). This integration challenge is manageable. Alternatively, the script can install the CSI driver *before* handing off to ArgoCD, to guarantee storage is ready. Either method is technically feasible – doing it via ArgoCD is cleaner long-term.

Similarly, **Velero** can be installed via ArgoCD (using its Helm chart or manifest). However, for Velero to function, we need credentials for S3 and possibly the DO token for snapshots. We would supply those via Kubernetes Secret. We can have the script create the `cloud-credentials` secret (with S3 keys) and the `digitalocean-token` secret in the Velero namespace, then let ArgoCD install Velero deployment which will mount those secrets. This decouples sensitive info from Git (you generally don't want to commit cloud credentials to the repo). So the bootstrap script's responsibility is to pass in secrets (GitHub token for repo access, DO token for CSI/Velero, S3 keys for Velero). Everything else (deployments, configmaps, etc.) can live in Git. This approach addresses **secret management** in a GitOps way: use tools like Vault or Sealed Secrets for better secret management in the future, but initially, a straightforward approach is fine.

1. **Post-Bootstrap Output:** Once the script finishes, we will have:
2. A kubeconfig file for the new cluster (so the user can access it with kubectl).
3. ArgoCD up and syncing from Git (so the cluster will progressively configure itself to the desired state).
4. Possibly an initial backup schedule configured (Velero can be set to auto-backup every X hours).
5. The script can output the ArgoCD UI URL and initial credentials (ArgoCD admin password), so the user can log in to the dashboard if desired.
6. If any restore was requested (in a DR scenario), the script would report the success of restoration steps (e.g., "Cluster restored from backup <name> and ready").

Feasibility & Idempotence: Each of these bootstrap steps uses proven tools and APIs. DigitalOcean's API is simple and reliable for VM provisioning. K3s's install script is very reliable and fast (and doesn't require heavy dependencies). Applying ArgoCD manifest and other YAML is straightforward with kubectl. The process can be made idempotent or at least repeatable: if something fails mid-way, the script can be re-run (perhaps cleaning the partial droplet) or it can check for existence of resources before creating new ones. Because everything ultimately converges via ArgoCD, even partial successes will converge once ArgoCD is running (for example, if the script died after installing ArgoCD but before creating the app, one could manually create the ArgoCD Application or rerun that part). This resilience is another benefit of GitOps – the desired state is in Git and can be re-applied as needed.

Scaling to Production (HA and Performance Considerations)

While the initial deployment is single-node and minimal, the design anticipates growth to a **production-grade, highly available cluster**. Scaling involves two dimensions: **vertical scaling** (bigger machines) and **horizontal scaling** (more machines), as well as improving redundancy of the control plane and critical services.

Vertical Scaling: A $20/month droplet provides modest resources. If an application's demands grow, the quickest improvement is to move to a larger VM (say, $40/month for 4vCPU/8GB). DigitalOcean allows resizing droplets (CPU/RAM) relatively easily by powering off and changing the plan. Since all state is either on the node (etcd/SQLite, local volumes) or in attached volumes, an offline resize is feasible (though it causes downtime). This is a coarse-grained but effective way to buy breathing room. For example, if memory becomes a bottleneck due to many pods or large in-memory processing, upgrading the droplet doubles the capacity with minimal change to the cluster itself. Vertical scaling has limits though, and it doesn't provide HA.

Horizontal Scaling (Workers): K3s makes it simple to add **worker nodes**. We can create additional DO droplets (e.g., $15/month 3GB nodes or another $20 node) and have them join the cluster as agents. This requires the **K3S_URL** (the address of the server) and **K3S_TOKEN** (join token). The token is generated on the server at install (stored in `/var/lib/rancher/k3s/server/node-token`). The

bootstrap script can output this token or even automate adding workers by creating more droplets with a similar cloud-init that runs `K3S_URL=https://<server>:6443 K3S_TOKEN=<token> sh -` [28]. Each new node will register with the server and appear in `kubectl get nodes`. Workloads can now be distributed. ArgoCD doesn't care about node count; it just sees more capacity to fulfill Kubernetes scheduling. With multiple nodes, we gain redundancy for worker pods (if one node goes down, others can run the pods, assuming persistent volumes are available to attach elsewhere, which the CSI driver enables). We can also spread out system components: perhaps taint the master so only core services run there, and use workers for apps.

High Availability Control Plane: The single biggest SPOF in the initial setup is the Kubernetes master (api-server and datastore). **K3s's default SQLite datastore cannot be clustered** – it's for single-server only [29]. For HA, we need to move to either an embedded etcd cluster or an external datastore. K3s now supports an **embedded etcd HA mode**: we can bring up 3 server nodes and they will form an etcd quorum (this is built-in and much easier than earlier external-DB setups). Amazingly, one can even **migrate a single-node K3s from SQLite to embedded etcd** simply by restarting it with `--cluster-init` flag [30]. This will detect no etcd DB present, convert the SQLite data to etcd, and start the etcd listener. Then additional servers can join using the token and `--server` flag [31] [32]. So, when the time comes for HA, the upgrade path is: - Create two new droplets for control plane. - On the original node, restart K3s with `--cluster-init` (promoting it to etcd). - Launch K3s on the new nodes with `K3S_TOKEN=<same token> --server https://<oldnode>:6443` [33]. They will join the etcd cluster and share control duties. - Now you have 3 masters (which K3s will also run as etcd members) [34]. The cluster can tolerate one master failing without downtime to the API.

This HA process is feasible and has been used in practice – it provides a clear path to eliminate the single point of failure. Alternatively, one could start with an **external DB** (like a managed MySQL or etcd) from the beginning, but that adds complexity/cost that isn't justified for the minimal start. The embedded etcd approach is fine as we scale up, and we'll have the DO token and automation to create new nodes when needed.

Load Balancing and Networking: In a multi-node cluster, especially with multiple masters, clients should have a stable endpoint for the API server. DO droplets don't have an integrated control plane load balancer (outside of DOKS). We could use a floating IP or round-robin DNS to point to one of the masters. However, in our GitOps scenario, direct external access to the API is not frequently needed (ArgoCD runs in-cluster, and developers can kubectl via the primary node's IP or set up a load balancer manually if desired). For services (applications) running in the cluster, if we add more nodes, we should consider using **MetalLB** or the **DigitalOcean Cloud Controller Manager (CCM)**. The DO CCM, if installed, can automatically provision DigitalOcean Load Balancer instances for Kubernetes Services of type LoadBalancer [35]. It uses the DO token and droplet information to attach the LB to your droplets. This is a nice way to expose services in a production scenario (e.g., you deploy a LoadBalancer Service for a web app, the CCM creates a DO LB that points to all node IPs on the service's node ports). In a single-node setup, CCM isn't very necessary (you could just use NodePort or the node's IP), but as we scale to multiple nodes or require HA on ingress, CCM + DO LBs are beneficial. The CCM also handles things like ensuring the Kubernetes Node objects have the correct external IPs, etc. We recommend, for production, installing the **digitalocean-cloud-controller-manager** deployment in the cluster (available via a YAML manifest or helm chart) so that the cluster is "DO-aware." This way, transitioning to more nodes or using DO resources is smoother. This component can also be managed via ArgoCD (just another manifest in Git).

For now, our minimal cluster might simply rely on the single node's IP and the Traefik ingress to accept traffic (Traefik listens on ports 80/443 on the node by default in K3s). The script can output that IP, and one can point a DNS record to it. As we scale, we could pivot to using a DO Load Balancer in front of

multiple nodes (Traefik can run on each node and the LB will distribute traffic to them). All these options are open and supported by the stack with minor config changes.

Performance Considerations: Running everything on one droplet means the control plane and workloads share resources. K3s's efficient use of SQLite is actually slightly less performant for write-heavy workloads than etcd, but for a small cluster it's negligible and it avoids the overhead of running an etcd server. As we add more objects and nodes, moving to etcd is wise for performance (etcd is built for distributed consensus and higher throughput). The ArgoCD controller will periodically compare Git state vs live state; on a tiny cluster this is very fast. If the Git repository contains thousands of manifests, ArgoCD's memory usage will increase accordingly. But for a typical setup with maybe tens of apps and a moderate number of resources, ArgoCD's resource usage (a few hundred MB RAM, low CPU) is fine on our droplet. In production, if ArgoCD needs it, we can scale its pods vertically or run its Redis component externally for more performance, but that's likely not needed until a much larger scale.

Each additional component (CSI driver, Velero, etc.) has overhead: - The CSI controller pods consume some memory/CPU but quite minimal (tens of MB) when idle. The CSI node plugin runs on each node as a daemonset; on one node that's one extra process plugin. It will use some CPU when attaching/detaching volumes, but that's infrequent and quick. - Velero server runs as a deployment, using some memory (around 50-100MB) and CPU mainly when processing a backup or restore. Backups can be scheduled during off-peak hours to mitigate impact. Also Velero can be scaled down (one could even install it on-demand to do a restore then remove it to save resources, though typically it stays to schedule recurring backups). - If we add monitoring (Prometheus) or logging (EFK stack) later for full production needs, those would be the most resource-intensive services. They might not fit well on a single 4GB node alongside everything else. We'd likely delay adding those until we have larger or multiple nodes. It's a trade-off: a truly production-grade cluster usually has monitoring/alerting. We assume for initial phases or non-critical workloads, those can be optional. When needed, one could deploy a lightweight monitoring (like Prometheus Agent or a cloud monitoring service) to avoid burdening the cluster.

Integration Challenges as the system grows: There are a few areas to watch: - **Database Migrations:** Switching K3s from SQLite to etcd is straightforward as documented (and tested) [30], but it does require a brief downtime (restarting the server). This should be planned in a maintenance window. - **Maintaining GitOps hygiene:** As we add more apps and perhaps multiple environments, the Git repo structure should evolve (e.g., separate folder for prod vs dev clusters, using ArgoCD **ApplicationSet** to deploy apps across clusters, etc.). The initial single cluster repo might need reorganization when scaling to many clusters or a more complex setup. But ArgoCD supports this growth with tools like ApplicationSets, so it's feasible. - **Secrets management:** Storing the GitHub and DO tokens as plaintext in the cluster (in Secrets) is acceptable initially but could be a security weakness if not handled properly (ensure RBAC prevents unauthorized reading of Secrets, consider enabling K3s secrets encryption at rest with a key). For production, integrating **Sealed Secrets** or HashiCorp Vault via an operator would be recommended so that GitOps can manage secrets safely (e.g., store only encrypted secrets in Git). This wasn't in initial requirements, but for long-term production readiness, it's worth planning. ArgoCD can work with SealedSecrets (the sealed-secret controller would decrypt and create real Secrets on the cluster). - **Traffic and Load**: If the cluster starts hosting customer-facing applications with significant traffic, one droplet's network and CPU might become a bottleneck. DO droplets have a shared 1Gbps network interface; one node can handle quite some traffic but all ingress/egress going through one machine could be a limit. By adding nodes and perhaps external load balancers, we can distribute load. This is a normal scaling concern and not a fundamental issue with the design. - **Cost Considerations:** Our stack is composed of open-source components (K3s, ArgoCD, etc.) – so no licensing costs. The primary costs are cloud resources: droplets ($20 each), block storage volumes (~$0.10/GB/month), object storage ($5 for 250GB on Spaces, or pennies on B2, etc.), and potentially load balancer ($10/

month on DO if used). Starting with one droplet, one might spend ~$25/month (droplet + small volume + some backup GBs). As we scale to say 3 nodes and some larger volumes, it could be on the order of $60-$100/month – still very reasonable for a production cluster compared to managed solutions. Eventually, if the operations burden grows, one might consider migrating to DigitalOcean's managed K8s. That service charges only for the worker nodes (control plane free up to certain limits), so cost-wise it could be similar. The decision would come down to control vs convenience. With our stack, we maintain full control of versions and configuration (e.g., we can use the latest K3s or specific customizations), whereas managed service would offload upgrades and etcd management to DO. Both approaches can be managed with GitOps (ArgoCD doesn't care if it's K3s or DOKS). Thus, our design keeps the door open for that transition if desired – indeed, the **GitOps methodology greatly eases moving between clusters** because the entire desired state is in Git and can be applied to a new cluster in an automated way.

Conclusion and Recommended Stack

Feasibility: It is technically feasible to achieve a GitOps-driven Kubernetes platform with only a GitHub token and a DigitalOcean token. All required steps – VM provisioning, K3s install, ArgoCD setup, storage provisioning, and backups – can be automated with API calls and scripts. The system can start minimal (one node, low cost) and incrementally evolve to a resilient, production-grade cluster. Key Kubernetes features (like CSI dynamic storage, ingress, load balancing, etc.) are supported on a self-managed DO droplet cluster with the addition of the appropriate controllers. Performance on a single droplet is sufficient for small-to-medium workloads, and the option to scale out addresses larger demands.

Recommended Tech Stack:

- **Kubernetes Distribution: K3s** (Lightweight Kubernetes by Rancher). K3s provides a full Kubernetes API in a single binary, optimized for minimal resource usage. It's production-ready and CNCF conformant [5], allowing a smooth transition to HA when needed. Starting with the default SQLite datastore is fine for a single node [29]; migrate to embedded etcd for multi-master HA later [30]. (Alternative: Canonical's MicroK8s could also be used, but K3s has the edge in simplicity for multi-node joins and includes out-of-the-box components we need like Traefik and local storage.)

- **GitOps Orchestration: Argo CD** for continuous delivery [1]. ArgoCD will ensure the cluster always matches the Git repository state, making operations predictable and auditable. Use the **App-of-Apps pattern** to manage cluster add-ons via one parent Application [3]. For bootstrapping new clusters (down the road, e.g., staging/prod clusters), consider ArgoCD Autopilot or similar tools to reduce manual steps, but the initial script approach is sufficient.

- **Ingress Controller: Traefik 2.x** (bundled with K3s by default) or switch to **NGINX Ingress Controller** via ArgoCD if preferred. Traefik in K3s is lightweight and works out-of-the-box for exposing services. It also supports Let's Encrypt integration for TLS. This covers north-south traffic. As you scale, ensure an approach for distributing ingress across nodes (Traefik can use a Service of type LoadBalancer which the DO CCM could bind to a DO LB).

- **Service Mesh (optional):** Not needed initially, but if the production use-case requires advanced traffic shaping or mTLS between services, a lightweight mesh like Linkerd could be added later via GitOps. We mention this only to note nothing in the design precludes adding such components as needed.

- **Persistent Storage: DigitalOcean CSI Driver** for Block Storage. This provides dynamic PersistentVolumeClaims via DO Volumes [10]. It's a critical piece for running databases or any stateful service reliably, and it integrates seamlessly with K3s (just a standard CSI driver deployment). All it needs is the DO API token and it will create/attach volumes on demand. For multi-node, it enables pods to move nodes (volumes detach/attach accordingly). Keep the default **local-path** storage class for ephemeral or dev use if desired, but use the **do-block-storage** class for important data. (Alternative: **Longhorn** if wanting an in-cluster storage solution without cloud reliance, but use with caution on limited resources).

- **Backup and Recovery: Velero** with an S3-compatible bucket. Velero will back up Kubernetes resource state to object storage and handle PV snapshots [17]. We recommend using **DigitalOcean Spaces** (same cloud, low latency) or **Backblaze B2 / Wasabi** for potentially lower cost per GB [18]. Configure Velero with the **DO snapshot plugin** to enable native volume snapshots in addition to file backups [22]. This gives fast restoration of large volumes if needed. Schedule regular backups (e.g., nightly) and before any major changes (ArgoCD can trigger Velero backups pre-deployment through webhooks or manual steps, if integrated). Test restoration process on a staging cluster to ensure the procedure is solid. (Alternative/Addition: use DigitalOcean's automated backups for Droplets as a second layer of safety, though those are VM-level snapshots not application-aware).

- **Object Storage Service (for apps):** If your workloads themselves need S3 storage (e.g., storing user uploads), you can either use an external service (Spaces, etc.) directly or deploy **MinIO** inside the cluster. For a cost-conscious approach, MinIO on top of a DO volume could serve as a local S3 service for dev/test. For production, an external service might be more reliable. This component is workload-specific, so include it only if needed.

- **Continuous Integration (CI):** While not part of the cluster, it's worth noting that with GitOps, your CI pipeline will basically push manifests or helm charts to the Git repo and ArgoCD will deploy them. Ensure proper testing and possibly use **ArgoCD ApplicationSets** for multi-environment promotion (dev -> staging -> prod via Git branches or folders). This ensures changes are reflected in the minimal cluster and any scaled-out clusters consistently.

- **Security and Hardening:** Apply best practices as you move to prod:

- Enable **HTTPS** on ingress (Traefik with Let's Encrypt or cert-manager issuing certs).
- Use **Rancher's Sealed Secrets** or Vault to avoid storing any plaintext secrets in Git; for example, the GitHub and DO tokens used by ArgoCD/CSI/Velero can be encrypted with Sealed Secrets and decrypted in-cluster [3]. ArgoCD can sync those sealed secrets safely.
- Enforce RBAC and network policies (K3s comes with Canal/Flannel and supports NetworkPolicy; you might add Calico for advanced policy if needed).
- Enable **Kubernetes resource quotas** and Pod Disruption Budgets as appropriate for stability once multiple teams or apps deploy to the cluster.
- Regularly update components (ArgoCD, K3s, etc.) – ArgoCD itself can be auto-updated via GitOps if you treat its install manifest as part of the repo. K3s updates can be handled by a script or tooling like Rancher system-upgrade controller.

Potential Challenges:
- *Integration issues:* Most components (K3s, ArgoCD, CSI, Velero) are well-tested independently, but ensure proper ordering (e.g., install CSI before any apps need a PVC). Also, watch out for version compatibility (match CSI driver version to K8s version, ArgoCD version to any CRDs in repo, etc.). These are manageable by pinning versions in your manifests and updating in a controlled manner. - *Resource*

contention: On a single node, if a backup job (Velero) is running while ArgoCD is syncing a large app and the app itself is under load, you could experience high IO or CPU usage. Mitigate by scheduling heavy jobs in off-hours and considering using QoS (request/limit) on pods to prioritize critical components (e.g., ensure K3s default pods have system-cluster-critical priority class, etc., which K3s does for its own components). - *DO API limits:* When scaling up volumes or load balancers via CSI/CCM, note that DO has API rate limits (usually 5000 requests/hour) – our use-case is far below that, but if you had hundreds of PVC creations at once, you'd want to stagger them. Also DO volume attach/detach has some time cost (a few seconds each), so mass rescheduling of pods might be slow if each needs a volume moved. In practice, not a big concern for moderate workloads.

Overall, this stack **balances minimal cost with long-term scalability** by starting lean (one VM, open-source tools) and incrementally adding only what is necessary as demand grows. Each component can scale horizontally: ArgoCD can manage multiple clusters or be HA (in larger installs you might run ArgoCD with multiple replicas and even a separate Redis for its session state), K3s can add nodes or join a high-availability setup, and data stores (volumes, backups) are on durable infrastructure from the get-go. There is no hard ceiling – you could grow this deployment to serve serious production workloads with many nodes. The important factor is to implement **DevOps best practices from the beginning (infrastructure as code, GitOps, backups)** so that scaling up is a matter of replicating patterns rather than reinventing architecture. Given the above analysis, the proposed approach is **feasible and recommended** for a cost-conscious team looking to start small and evolve their Kubernetes infrastructure over time.

Example Configuration Recap: To tie everything together, here are abridged examples of key configuration pieces, as one might have in the GitOps repository or script:

- *K3s Install Script (User Data):* Installs K3s and ArgoCD, and bootstraps ArgoCD sync

```
#cloud-config
runcmd:
  - curl -sfL https://get.k3s.io | sh -
  - kubectl wait --for=condition=Ready node/$(hostname) --timeout=120s
  - kubectl create ns argocd
  - kubectl apply -n argocd -f https://raw.githubusercontent.com/
argoproj/argo-cd/stable/manifests/install.yaml
  - kubectl apply -f /root/bootstrap-app.yaml   # bootstrap-app.yaml
contains the ArgoCD Application pointing to the repo
```

(This assumes `/root/bootstrap-app.yaml` *was attached via user-data or fetched from somewhere. In practice, the script could wget it from the repo if the repo is public, or the script template could include it.)*

- *ArgoCD Bootstrap Application (app-of-apps):* Points ArgoCD to the cluster config folder

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: cluster-bootstrap
  namespace: argocd
spec:
```

```
  destination:
    server: https://kubernetes.default.svc
    namespace: argocd
  source:
    repoURL: git@github.com:<org>/<repo>.git
    targetRevision: main
    path: clusters/single-node
  syncPolicy:
    automated:
      selfHeal: true
      prune: true
```

Note: The above uses an SSH git URL – we would have configured an SSH key or use https with token. This Application would then cause ArgoCD to sync everything under `clusters/single-node/` directory, which might include: `csi-driver.yaml`, `velero.yaml`, `ingress.yaml`, etc., as well as app manifests.

- *DigitalOcean CSI Driver manifest reference:* included in GitOps as `csi-driver.yaml` (or installed via Helm). For example, ensure it creates `StorageClass` default to do-block-storage. After applying, one can create a PVC like:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: test-pvc
spec:
  accessModes: [ReadWriteOnce]
  resources: { requests: { storage: 1Gi } }
  storageClassName: do-block-storage
```

and see it bound to a DO Volume automatically [10].

- *Velero values (for Helm or CLI):* For Helm chart, values would include the S3 endpoint, bucket, creds, and enabling the DO plugin. For CLI, we showed an example command. Key part is setting environment:

```
# Secret for Velero (for credentials)
apiVersion: v1
kind: Secret
metadata:
  name: cloud-credentials
  namespace: velero
stringData:
  cloud: |
    [default]
    aws_access_key_id = <SPACES_OR_S3_KEY_ID>
    aws_secret_access_key = <SPACES_OR_S3_SECRET_KEY>
```

```
  digitalocean: |
    digitalocean_token = <DO_API_TOKEN>
```

And a `BackupStorageLocation` resource:

```
apiVersion: velero.io/v1
kind: BackupStorageLocation
metadata:
  name: default
  namespace: velero
spec:
  provider: aws
  objectStorage:
    bucket: <your-bucket-name>
    prefix: velero
  config:
    region: <region>
    s3Url: https://<region>.digitaloceanspaces.com
```

This tells Velero to use AWS plugin (provider `aws`) to talk to the given endpoint with the credentials. Similar for B2 or Wasabi (just change s3Url and region accordingly). The DO snapshot plugin is enabled by creating a `VolumeSnapshotLocation` as mentioned. All these can be codified and applied via ArgoCD.

In conclusion, the stack consisting of **K3s + ArgoCD + DO CSI + Velero (+ external S3)** is not only feasible but aligns well with modern infrastructure-as-code and GitOps practices. It keeps initial complexity low (single node, few moving parts) but each component lays groundwork for scaling: ArgoCD for managing more apps/clusters, K3s for scaling nodes or migrating to HA, CSI for adding storage on the fly, and Velero for recovering from anything unforeseen. By starting minimal and augmenting as requirements grow, you achieve a cost-efficient yet robust Kubernetes platform.

---

[1] [2] [3] Implementing GitOps using Argo CD | DigitalOcean
https://www.digitalocean.com/community/developer-center/implementing-gitops-using-argo-cd

[4] Lightweight Kubernetes Distribution - K3s - Rancher
https://www.rancher.com/products/lightweight-kubernetes-distribution

[5] What is K3s and How is it Different from K8s? | Traefik Labs
https://traefik.io/glossary/k3s-explained/

[6] k3s-io/k3s: Lightweight Kubernetes - GitHub
https://github.com/k3s-io/k3s

[7] [8] [9] [15] [16] Volumes and Storage | K3s
https://docs.k3s.io/storage

[10] [11] [12] [13] [14] GitHub - digitalocean/csi-digitalocean: A Container Storage Interface (CSI) Driver for DigitalOcean Block Storage
https://github.com/digitalocean/csi-digitalocean

17 21 22 How To Back Up and Restore a Kubernetes Cluster on DigitalOcean Using Velero | DigitalOcean
https://www.digitalocean.com/community/tutorials/how-to-back-up-and-restore-a-kubernetes-cluster-on-digitalocean-using-velero

18 19 Velero Docs - Providers
https://velero.io/docs/v1.13/supported-providers/

20 23 24 25 26 27 GitHub - digitalocean/velero-plugin: DigitalOcean plugin for https://velero.io
https://github.com/digitalocean/velero-plugin

28 What is K3s? Architecture, Setup, and Security - Aqua
https://www.aquasec.com/cloud-native-academy/kubernetes-101/k3s/

29 Cluster Datastore | K3s
https://docs.k3s.io/datastore

30 31 32 33 34 High Availability Embedded etcd | K3s
https://docs.k3s.io/datastore/ha-embedded

35 kubernetes - How to add a loadbalancer to a cluster on digitalocean
https://stackoverflow.com/questions/62371926/how-to-add-a-loadbalancer-to-a-cluster-on-digitalocean