INSTALLATION GUIDE

A guide for installing or migrating to CircleCI Server v4.0.0 on Google Cloud Project

docs@circleci.com

Version 4.0.0, 08/18/2022: FINAL

Phase 1 - Prerequisites

CircleCI server v4.x is installed as a helm chart. The installation process is broken down into four phases. There is a validation step at the end of each phase, allowing you to confirm success before moving to the next phase. Depending on your requirements, phases 3 and 4 may include multiple steps. This installation guide assumes you have already read the CircleCI Server v4.x Overview.

In the following sections, replace any sections indicated by `< >` with your details.

1. Install required software

Download and install the following software before continuing:

Tool	Version	Used for	Notes
Terraform	0.15.4 or greater	Infrastructure Management	
kubectl	1.19 or greater	Kubernetes CLI	
Helm	3.9.2 or greater	Kubernetes Package Management	
Helm Diff	3.5.0 or greater	Helping with `values.yaml` changes and updates	Optional, but may help with troubleshooting between releases
Velero CLI	Latest	Backup and restore capability	See Velero's supported providers documentation for further information.

GCP required software

- `gcloud` and `gsutil`. You can install and set-up these tools up by installing Google Cloud SDK. For further information refer to the Google Cloud SDK docs.

S3 compatible storage required software

- Install and configure MinIO CLI for your storage provider.

2. Create a Kubernetes cluster

CircleCI server installs into an existing Kubernetes cluster. The application uses a large number of resources. Depending on your usage, your Kubernetes cluster should meet the following requirements:

Number of daily active CircleCI users	Minimum Nodes	Total CPU	Total RAM	NIC speed
< 500	3	12 cores	32 GB	1 Gbps
500+	3	48 cores	240 GB	10 Gbps

Supported Kubernetes versions:

CircleCI Version	Kubernetes Version
4.0.0	1.22 - 1.23

Creating a Kubernetes cluster is your responsibility. Please note:

- You must have appropriate permissions to list, create, edit, and delete pods in your cluster. Run this command to verify your permissions:

```
kubectl auth can-i <list|create|edit|delete> pods
```

- There are no requirements regarding VPC setup or disk size for your cluster. It is recommended that you set up a new VPC rather than use an existing one.

GKE

You can learn more about creating a GKE cluster in the GKE docs.

Do not use Autopilot cluster. CircleCI requires functionality that is not supported by GKE Autopilot.

1. Install and configure the GCP CLI for your GCP account. This includes creating a Google Project, which will be required to create a cluster within your project.

When you create your project, make sure you also enable API access. If you do not enable API access, the command we will run next (to create your cluster) will fail.

Setting the default `project id`, compute `zone` and `region` will make running subsequent commands easier:

```
gcloud config set project <PROJECT_ID>
gcloud config set compute/zone <ZONE>
gcloud config set compute/region <REGION>
```

2. Create your cluster

CircleCI recommends using Workload Identity to allow workloads/pods in your GKE clusters to impersonate Identity and Access Management (IAM) service accounts to access Google Cloud services. Use the following command to provision a simple cluster:

```
gcloud container clusters create circleci-server \
  --num-nodes 3 \
  --machine-type n1-standard-4 \
  --workload-pool=<PROJECT_ID>.svc.id.goog
```

Your kube-context should get updated with the new cluster credentials automatically.

If you need to update your kube-context manually, you can by running the following:

```
gcloud container clusters get-credentials circleci-server
```

3. Install the GKE authentication plugin for `kubectl`:

```
gcloud components install gke-gcloud-auth-plugin
```

4. Verify your cluster:

```
kubectl cluster-info
```

5. Create a service account:

```
gcloud iam service-accounts create <SERVICE_ACCOUNT_ID> --description="<DESCRIPTION>" \
  --display-name="<DISPLAY_NAME>"
```

6. Retrieve credentials for the service account:

```
gcloud iam service-accounts keys create <KEY_FILE> \
  --iam-account <SERVICE_ACCOUNT_ID>@<PROJECT_ID>.iam.gserviceaccount.com
```

Enable Workload Identity in GKE (optional)

Follow these steps if you already have a GKE cluster and need to enable Workload Identity on the cluster and the node pools.

1. Enable Workload Identity on existing cluster:

```
  gcloud container clusters update "<CLUSTER_NAME>" \
    --workload-pool="<PROJECT_ID>.svc.id.goog"
```

2. Get node pools of existing GKE cluster:

```
gcloud container node-pools list --cluster "<CLUSTER_NAME>"
```

3. Update existing node pools:

```
gcloud container node-pools update "<NODEPOOL_NAME>" \
  --cluster="<CLUSTER_NAME>" \
  --workload-metadata="GKE_METADATA"
```

You must repeat Step 3 for all the existing node pools. Follow these links for steps to enable Workload Identity for your Kubernetes service accounts:

- Nomad Autoscaler
- VM
- Object-Storage

3. Create a new GitHub OAuth app

If GitHub Enterprise and CircleCI server are not on the same domain, then images and icons from GHE will fail to load in the CircleCI web app.

Registering and setting up a new GitHub OAuth app for CircleCI server allows for authorization control to your server installation using GitHub OAuth and for updates to GitHub projects/repos using build status information. The following steps apply for both GitHub.com and GitHub Enterprise.

1. In your browser, navigate to **your GitHub instance** > **User Settings** > **Developer Settings** > **OAuth Apps** and click the **New OAuth App** button.

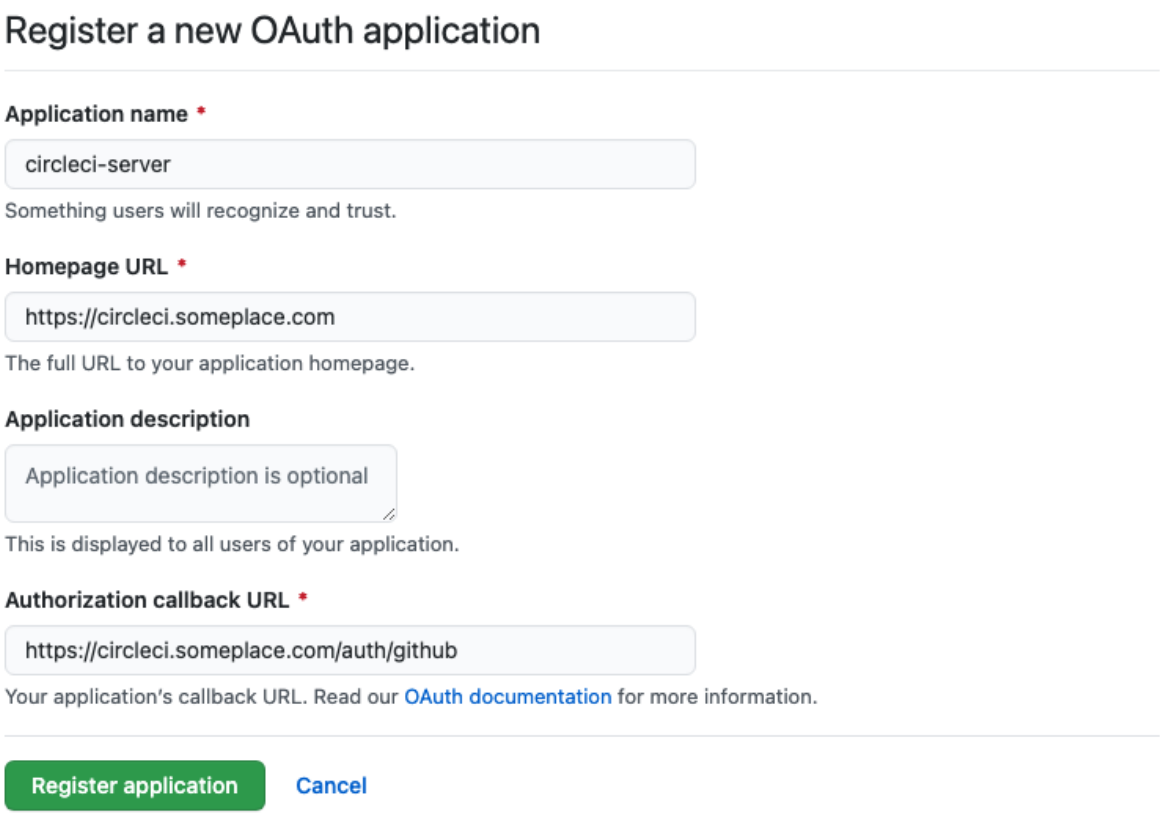


Figure 1. New GitHub OAuth App

2. Complete the following fields, based on your planned installation:
    - **Homepage URL**: The URL of your planned CircleCI installation.
    - **Authorization callback URL**: The authorization callback URL is the URL of your planned CircleCI installation followed by `/auth/github`
3. Once completed, you will be shown the **Client ID**. Select **Generate a new Client Secret** to generate a Client Secret for your new OAuth App. You will need these values when you configure CircleCI server.

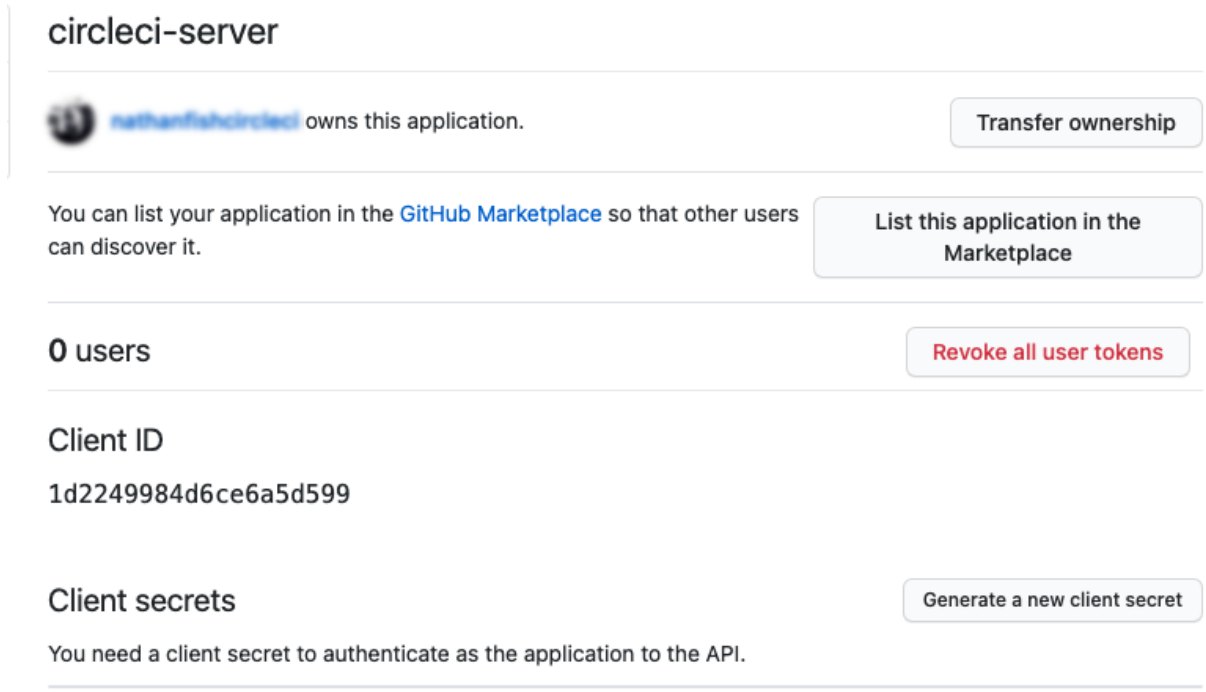


Figure 2. Client ID and Secret

GitHub Enterprise

If using GitHub Enterprise, you also need a personal access token and the domain name of your GitHub Enterprise instance.

Create the `defaultToken` by navigating to **User Settings > Developer Settings > Personal access tokens**. The default token requires no scopes. You will need this value when you configure CircleCI server.

4. Frontend TLS certificates

By default, CircleCI server creates self-signed certificates to get you started. In production, you should supply a certificate from a trusted certificate authority. The Let's Encrypt certificate authority, for example, can issue a free certificate using their certbot tool. The sections below cover using Google Cloud DNS and AWS Route 53.

It is important that your certificate contains both your domain and the `app.*` subdomain as subjects. For example, if you host your installation at `server.example.com`, your certificate must cover `app.server.example.com` and `server.example.com`.

Once you have created your certificates using one of the methods described below, you can use the following commands to retrieve the certificates later when you need them during this installation:

```
ls -l /etc/letsencrypt/live/<CIRCLECI_SERVER_DOMAIN>
```

```
cat /etc/letsencrypt/live/<CIRCLECI_SERVER_DOMAIN>/fullchain.pem
```

```
cat /etc/letsencrypt/live/<CIRCLECI_SERVER_DOMAIN>/privkey.pem
```

Google Cloud DNS

1. If you host your DNS on Google Cloud, you need the **certbot-dns-google** plugin installed. You can install the plugin with the following command:

   ```
   python3 -m pip install certbot-dns-google
   ```

2. The service account used to run `certbot` will need to have access to Cloud DNS in order to provision the necessary records used by Let's Encrypt for domain validation.

   a. Create a custom role for certbot:

   ```
   gcloud iam roles create certbot --project=<PROJECT_ID> \
       --title="<TITLE>" --description="<DESCRIPTION>" \
       --permissions
   ="dns.changes.create,dns.changes.get,dns.changes.list,dns.managedZones.get,dns.managedZones.l
   ist,dns.resourceRecordSets.create,dns.resourceRecordSets.delete,dns.resourceRecordSets.list,d
   ns.resourceRecordSets.update" \
       --stage=ALPHA
   ```

   b. Bind the new role to the service account which we created earlier:

   ```
   gcloud projects add-iam-policy-binding <PROJECT_ID> \
       --member="serviceAccount:<SERVICE_ACCOUNT_ID>@<PROJECT_ID>.iam.gserviceaccount.com" \
       --role="<ROLE_NAME>"
   ```

3. Finally, the following commands will provision a certification for your installation:

   ```
   certbot certonly --dns-google --dns-google-credentials <KEY_FILE> -d "<CIRCLECI_SERVER_DOMAIN>"
   -d "app.<CIRCLECI_SERVER_DOMAIN>"
   ```

AWS Certificate Manager

Instead of provisioning your own TLS certificates, if you are setting up CircleCI Server in an AWS environment, you can have AWS provision TLS certificates using Certificate Manager.

```
aws acm request-certificate \
  --domain-name <CIRCLECI_SERVER_DOMAIN> \
  --subject-alternative-names app.<CIRCLECI_SERVER_DOMAIN> \
  --validation-method DNS \
  --idempotency-token circle
```

After running this command, navigate to the Certificate Manager AWS console and follow the wizard to provision the required DNS validation records with Route53. Take note of the ARN of the certificate once it is issued.

Upstream TLS termination

You may have a requirement to terminate TLS for CircleCI server outside the application. This is an alternate method to using ACM or supplying the certificate chain during Helm deployment. An example would be a proxy running in front of the CircleCI installation providing TLS termination for your domain name. In this case the CircleCI application acts as the backend for your load balancer or proxy.

CircleCI server listens on a number of ports which need to be configured depending how your are routing the traffic. See the list of port numbers below:

- Frontend / API Gateway [TCP 80, 443]
- VM service [TCP 3000]
- Nomad server [TCP 4647]
- Output processor [gRPC 8585]

Depending on your requirements you may choose to terminate TLS for only the frontend/api-gateway or provide TLS for services listening on all the ports.

The Output Processor service communicates using gRPC and requires the proxy or loadbalancer to support HTTP/2.

5. Encryption/signing keys

The keysets generated in this section are used to encrypt and sign artifacts generated by CircleCI. You will need these values to configure server.

Store these values securely. If they are lost, job history and artifacts will not be recoverable.

a. Artifact signing key

To generate an artifact signing key, run the following command:

```
docker run circleci/server-keysets:latest generate signing -a stdout
```

b. Encryption signing key

To generate an encryption signing key, run the following command:

```
docker run circleci/server-keysets:latest generate encryption -a stdout
```

6. Object storage and permissions

Server v4.x hosts build artifacts, test results, and other state object storage. The following storage options are supported:

- AWS S3
- MinIO
- Google Cloud Storage

While any S3 compatible object storage may work, we test and support AWS S3 and MinIO. For object storage providers that do not support the S3 API, such as Azure blob storage, we recommend using MinIO Gateway.

Follow the instructions below to create a bucket and access method for S3 or GCS.

If you are installing behind a proxy, object storage should be behind this proxy also. Otherwise, proxy details will need to be supplied at the job level within every project `.circleci/config.yml` to allow artifacts, test results, cache save and restore, and workspaces to work. For more information see the Installing Server Behind a Proxy guide.

Google Cloud Storage

a. Create a GCP bucket

If your server installation runs in a GKE cluster, ensure that your current IAM user is cluster admin for this cluster, as RBAC (role-based access control) objects need to be created. More information can be found in the GKE documentation.

```
gsutil mb gs://circleci-server-bucket
```

b. Set up authentication

The recommended method for workload/pod authentication is to use Workload Identity. However, you may also use static credentials (json key file).

1. Create a Service Account.

```
gcloud iam service-accounts create circleci-storage --description="Service account for CircleCI
object storage" --display-name="circleci-storage"
```

2. Bind the `objectAdmin` role to the service account.

```
gcloud projects add-iam-policy-binding <PROJECT_ID> \
    --member="serviceAccount:circleci-storage@<PROJECT_ID>.iam.gserviceaccount.com" \
    --role="roles/storage.objectAdmin" \
    --condition='expression=resource.name.startsWith("projects/_/buckets/circleci-server-bucket"),title=restrict_bucket'
```

3. Either enable Workload Identity or use static credentials.

   Option 1: Workload Identity

   When using Workload Identity you need to configure your account such that the workloads/pods can access the storage bucket from the cluster using the Kubernetes service account `"<K8S_NAMESPACE>/object-storage"`.

```
gcloud iam service-accounts add-iam-policy-binding circleci-storage@<PROJECT_ID>.iam.gserviceaccount.com \
    --role roles/iam.workloadIdentityUser \
    --member "serviceAccount:<PROJECT_ID>.svc.id.goog[<K8S_NAMESPACE>/object-storage]"
```

```
gcloud projects add-iam-policy-binding <PROJECT_ID> \
    --member serviceAccount:circleci-storage@<PROJECT_ID>.iam.gserviceaccount.com \
    --role roles/iam.serviceAccountTokenCreator \
    --condition=None
```

Option 2: Static credentials

If you are not using Workload Identity, create a json file containing static credentials.

```
gcloud iam service-accounts keys create <KEY_FILE> \
    --iam-account circleci-storage@<PROJECT_ID>.iam.gserviceaccount.com
```

Phase 2 - Core services

Before you begin with the CircleCI server v4.x core services installation phase, ensure all prerequisites are met.

In the following sections, replace any sections indicated by `< >` with your details.

1. Create a namespace

Create a namespace to install the application into.

```
kubectl create ns <namespace>
```

Once you have created your namespace, we recommend setting your `kubectl` context too, with the following command: `kubectl config set-context --current --namespace <namespace>`

2. Pull images

Credentials to pull the images from CircleCI's image registry will be provided to you as part of the onboarding process. A `docker-registry` Kubernetes Secret will be used to pull images from Azure Container Registry (ACR). You have two options, depending on whether your application has access to the public internet.

Option 1: Your application has access to the public internet.

This example creates a Kubernetes Secret to enable deployments to pull images from CircleCI's image registry. The `docker-registry` Kubernetes Secret takes the following form:

```
kubectl create secret docker-registry regcred \
  --docker-server=cciserver.azurecr.io \
  --docker-username=<your-username> \
  --docker-password="<provided-token>" \
  --docker-email=<your-contact-email>
```

Option 2: Your application does NOT have access to the public internet.

The credentials provided to you allow you to pull and store copies of our images locally. Pull and store the images in whichever Docker repository you have available. The `docker-registry` Kubernetes Secret takes the following form:

```
kubectl create secret docker-registry regcred \
  --docker-server=<your-docker-image-repo> \
  --docker-username=<your-username> \
  --docker-password=<your-access-token> \
  --docker-email=<your-email>
```

3. Create helm values

Before installing CircleCI, it is recommended to create a new `values.yaml` file unique to your installation. The Installation Reference section contains some example `values.yaml` files that are a good place to start. The following describes the minimum required values to include in `values.yaml`. Additional customizations are available, see the provided `values.yaml` for all available options.

For sensitive data there are two options:

- add into the `values.yaml` file
- add them as Kubernetes Secrets directly

This flexibility allows you to manage Kubernetes Secrets using whichever process you prefer.

Whichever option you choose, this sensitive information is stored as a Kubernetes Secret within CircleCI.

a. API token

The application requires a Kubernetes Secret containing an API token. This API token is used to facilitate internal API communication to api-service. Use a random string and store it securely, CircleCI will not be able to recover this value if lost. There are two options depending on whether you want to create the Kubernetes Secret, or if you want CircleCI to create it for you.

Option 1: Create the Kubernetes Secret yourself.

```
kubectl create secret generic api-token \
  --from-literal=api-token=<your-super-secret-random-value>
```

Option 2: CircleCI creates the Kubernetes Secret for you.

Add the value to `values.yaml`. CircleCI will create the Kubernetes Secret automatically.

```
apiToken: "<your-super-secret-random-value>"
```

b. Session cookie

The application requires a session cookie key Kubernetes Secret, which CircleCI uses to sign session cookies. The Secret must be exactly 16 characters long. Use a random string and store it securely, CircleCI will not be able to recover this value if lost. There are two options depending on whether you want to create the Kubernetes Secret, or if you want CircleCI to create it for you.

Option 1: Create the Kubernetes Secret yourself.

```
kubectl create secret generic session-cookie \
--from-literal=session-cookie-key=<your-secret-key-16-chars>
```

Option 2: CircleCI creates the Kubernetes Secret for you.

Add the value to `values.yaml`. CircleCI will create the Secret automatically.

```
sessionCookieKey: "<your-secret-key-16-chars>"
```

c. Encryption

The application requires a Kubernetes Secret containing signing and encrpytion keysets. These keysets are used to encrypt and sign artifacts generated by CircleCI. These keys were created during the prerequisites phase. CircleCI will not be able to recover the values if lost. Depending on how you prefer to manage Kubernetes Secrets, there are two options.

Option 1: Create the Kubernetes Secret yourself.

```
kubectl create secret generic signing-keys \
  --from-literal=signing-key=<your-generated-signing-key> \
  --from-literal=encryption-key=<your-generated-encryption-key>
```

Option 2: CircleCI creates the Kubernetes Secret.

Add the value to `values.yaml`. CircleCI will create the Secret automatically.

```
keyset:
  signing: '<your-generated-signing-key>'
  encryption: '<your-generated-encryption-key>'
```

d. Postgres credentials

The application requires a Kubernetes Secret containing Postgres credentials. This is true when using either the internal (default) or an externally hosted instance of Postgres. CircleCI will not be able to recover the values if lost. Based on how you prefer to manage Kubernetes Secrets there are two options.

Option 1: Create the Secret yourself.

```
kubectl create secret generic postgresql \
  --from-literal=postgres-password=<postgres-password>
```

You must then provide the following to the `values.yaml` file:

```
postgresql:
  auth:
    existingSecret: postgresql
```

Option 2: CircleCI creates the Kubernetes Secret.

Add the credentials to `values.yaml`, and CircleCI will create the Secret automatically.

```
postgresql:
  auth:
    postgresPassword: "<postgres-password>"
```

e. MongoDB credentials

The application requires a Kubernetes Secret containing MongoDB credentials. This is true when using either the internal (default) or an externally hosted instance of MongoDB. CircleCI will not be able to recover the values if lost. Based on how you prefer to manage Kubernetes Secrets there are two options.

Option 1: Create the Kubernetes Secret yourself.

```
kubectl create secret generic mongodb-credentials \
  --from-literal=mongodb-root-password=<root-password> \
  --from-literal=mongodb-password=<user-password>
```

You must then provide the following to the `values.yaml` file:

```
mongodb:
  auth:
    existingSecret: mongodb-credentials
```

Option 2: CircleCI creates the Kubernetes Secret.

Add the credentials to `values.yaml`, and CircleCI will create the Secret automatically.

```yaml
mongodb:
  auth:
    rootPassword: "<root-password>"
    password: "<user-password>"
```

f. RabbitMQ configurations and auth Secrets

The RabbitMQ installation requires two random alphanumeric strings. CircleCI will not be able to recover the values if lost. Based on how you prefer to manage Kubernetes Secrets there are two options.

Option 1: Create the Secret yourself.

```
kubectl create secret generic rabbitmq-key \
--from-literal=rabbitmq-password=<secret-alphanumeric-password> \
--from-literal=rabbitmq-erlang-cookie=<secret-alphanumeric-key>
```

You must then provide the following to the `values.yaml` file:

```yaml
rabbitmq:
  auth:
    existingPasswordSecret: rabbitmq-key
    existingErlangSecret: rabbitmq-key
```

Option 2: CircleCI creates the Kubernetes Secret.

Add the value to `values.yaml`, and CircleCI will create the Kubernetes Secret automatically.

```yaml
rabbitmq:
  auth:
    password: "<secret-alphanumeric-password>"
    erlangCookie: "<secret-alphanumeric-key>"
```

g. Pusher Kubernetes Secret

The application requires a Kubernetes Secret for Pusher. CircleCI will not be able to recover the values if lost. Based on how you prefer to manage Kubernetes Secrets there are 2 options:

Option 1: Create the Kubernetes Secret yourself.

```
kubectl create secret generic pusher \
--from-literal=secret=<pusher-secret>
```

Option 2: CircleCI creates the Kubernetes Secret.

Add the value to `values.yaml`, and CircleCI will create the Kubernetes Secret automatically.

```
pusher:
  secret: "<pusher-secret>"
```

h. Global

All values in this section are children of `global` in your `values.yaml`.

CircleCI domain name (required)

Enter the domain name you specified when creating your Frontend TLS key and certificate.

```
global:
  ...
  domainName: "<full-domain-name-of-your-install>"
```

License

A license will be provided by CircleCI, add it to `values.yaml`:

```
global:
  ...
  license: '<license>'
```

Registry

The registry to pull images from will have been provided to you, or you may have added the images to your own hosted registry. Add the registry to `values.yaml`:

```
global:
  ...
  container:
    registry: <registry-domain eg: cciserver.azurecr.io >
    org: <your-org-if-applicable>
```

i. TLS

For TLS, you have 4 options:

Do nothing

Do nothing. Let's Encrypt will automatically request and manage certificates for you. This is a good option for trials but not recommended for production use.

Supply a private key and certificate

You can supply a private key and certificate, which you may have created during the prerequisites steps. The key and certificates will need to be base64 encoded. You can retrieve and encode the values with the following commands:

```
cat /etc/letsencrypt/live/<CIRCLECI_SERVER_DOMAIN>/privkey.pem | base64
cat /etc/letsencrypt/live/<CIRCLECI_SERVER_DOMAIN>/fullchain.pem | base64
```

And add them to `values.yaml`:

```
tls:
  certificate: '<full-chain>'
  privateKey: '<private-key>'
```

Use ACM

Have AWS Certificate Manager (ACM) automatically request and manage certificates for you. Follow the ACM documentation for instructions on how to generate ACM certificates.

Enable `aws_acm` and add the `service.beta.kubernetes.io/aws-load-balancer-ssl-cert` annotation to point at the ACM ARN:

```
nginx:
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-ssl-cert: <acm-arn>
  aws_acm:
    enabled: false
```

If you have already deployed CircleCI server, enabling ACM is a destructive change to the loadbalancer. The service will have to be regenerated to allow the use of your ACM certificates and so the associated loadbalancer will also be regenerated. You will need to update your DNS records to the new loadbalancer once you have redeployed CircleCI server.

Disable TLS within CircleCI

You can choose to disable TLS termination within CircleCI. The system will still need to be accessed over HTTPS, so TLS termination will be required somewhere upstream of CircleCI. Implement this by following the first option (do nothing) and forward the following ports to your CircleCI load balancer:

- Frontend / API Gateway [TCP 80, 443]
- VM service [TCP 3000]
- Nomad server [TCP 4647]
- Output processor [gRPC 8585]

j. GitHub integration

To configure GitHub with CircleCI, there are two options for providing credentials to the deployment. Steps for both GitHub and GitHub Enterprise (GHE) are given in the next two sections.

GitHub

These instructions are for the GitHub.com, **not** GitHub Enterprise. Use the client ID and secret you created with your Github OAuth application in the prerequisites phase.

Option 1: Create the Kubernetes Secret yourself.

```
kubectl create secret generic github-secret \
  --from-literal=clientId=<client-id> \
  --from-literal=clientSecret=<client-secret>
```

Option 2: CircleCI creates the Kubernetes Secret.

Add the client ID and secret to the `values.yaml` file. CircleCI will create the Kubernetes Secret automatically.

```
github:
  clientId: "<client-id>"
  clientSecret: "<client-secret>"
```

GitHub Enterprise

The instructions for GitHub Enterprise are similar, with a few extra steps to enable Enterprise and create the required default token.

In the case of GitHub Enterprise add the `defaultToken` created in the prerequisite phase to the `GitHub` section. The hostname should not include the protocol, ex: `github.exampleorg.com`.

Option 1: Create the Kubernetes Secret yourself.

```
kubectl create secret generic github-secret \
  --from-literal=clientId=<client-id> \
  --from-literal=clientSecret=<client-secret> \
  --from-literal=defaultToken=<default-token>
```

You must then provide the following to the `values.yaml` file:

```
github:
  enterprise: true
  hostname: "<github-enterprise-hostname>"
```

Option 2: CircleCI creates the Kubernetes Secret.

Add `clientID`, `clientSecret` and `defaultToken` to the `values.yaml` file. You must also set `enterprise` to `true`, and provide the `hostname` for your enterprise GitHub. CircleCI will create the Kubernetes Secret automatically.

```
github:
  ...
  clientId: "<client-id>"
  clientSecret: "<client-secret>"
  enterprise: true
  hostname: "<github-enterprise-hostname>"
  defaultToken: "<token>"
```

k. Object storage

Regardless of your storage provider, the bucket name you created during the prerequisites phase will need to be included.

```
object_storage:
  bucketName: "<bucket-name>"
```

Google Cloud Storage

Under `object_storage` add the following.

```
gcs:
    enabled: true
```

Under `object_storage.gcs` you may add `service_account`, `workloadIdentity`, or neither. The keys/role were created during the prerequisites steps.

Option 1: Use a service account.

Add a JSON format key of the Service Account to use for bucket access. Add the following to the `object_storage.gcs` section:

```
service_account: "<service-account>"
```

Option 2: Use Workload Identity.

Add the Service Account Email of the workload identity. Add the following to the `object_storage.gcs` section:

```
workloadIdentity: "<workload-identity-service-account-email>"
```

Option 3: Create the Kubernetes Secret yourself

Instead of storing the service account in your `values.yaml` file, you may create the Kubernetes Secret yourself.

```
kubectl create secret generic object-storage-secret \
  --from-literal=gcs_sa.json=<service-account>
```

I. Installing behind a proxy

Depending on your security requirements, you might want to install CircleCI server behind a proxy. Installing behind a proxy gives you the power to monitor and control access between your installation and the broader Internet. For further information including limitations of installation behind a proxy, see the Installing Server Behind a Proxy guide.

The following fields need to be configured in your `values.yaml`:

- Toggle `proxy.enabled` to `"1"`
- Enter details for `proxy.http.host` and `proxy.https.host`, along with their associated ports. These values can be the same but they both need to be configured.
- For authentication you will need to configure `proxy.http.auth.enabled` and `proxy.https.auth.enabled` as `"1"`. You will also need to configure the respective username and password for both HTTP and HTTPS.
- configure the `no_proxy` hosts and subnets. This should include localhost, your GitHub Enterprise host (optional), the hostname of your CircleCI installation (see Known Limitations for an explanation), and the CIDRs of both vm-service and Nomad.

```
proxy:
  enabled: "1"
  http:
    host: "<proxy.example.internal>"
    port: "3128"
    auth:
      enabled: "1"
      username: "<proxy-user>"
      password: "<proxy-password>"
  https:
    host: "<proxy.example.internal>"
    port: "3128"
    auth:
      enabled: "1"
      username: "<proxy-user>"
      password: "<proxy-password>"
  no_proxy:
    - localhost
    - 127.0.0.1
    - "<github.example.internal>"
    - "<circleci.example.internal>"
    - "<nomad-subnet-cidr>"
    - "<vm-service-cidr>"
    - "<vpc-or-subnet-cidr>"   # VPC or subnets to exclude from the proxy (optional)
```

4. Deploy

Once you have completed the fields detailed above, you can deploy CircleCI's core services:

```
USERNAME=<provided-username>
PASSWORD=<token>
namespace=<your-namespace>
helm registry login cciserver.azurecr.io/circleci-server -u $USERNAME -p $PASSWORD
helm install circleci-server oci://cciserver.azurecr.io/circleci-server -n $namespace --version
4.0.0 -f <path-to-values.yaml>
```

5. Create DNS entry

Create a DNS entry for your NGINX load balancer, for example, `circleci.your.domain.com` and `app.circleci.your.domain.com`. The DNS entry should align with the DNS names used when creating your TLS certificate and GitHub OAuth app during the prerequisites steps. All traffic will be routed through this DNS record.

You need the IP address, or, if using AWS, the DNS name of the NGINX load balancer. You can find this information with the following command:

```
kubectl get service circleci-proxy
```

For more information on adding a new DNS record, see the following documentation:

- Managing Records (GCP)
- Creating records by using the Amazon Route 53 Console (AWS)

6. Validation

You should now be able to navigate to your CircleCI server installation and log in to the application successfully.

Now we will move on to build services. It may take a while for all your services to be up. You can periodically check by running the following command (you are looking for the `frontend"` pod to show a status of `running` and **ready** should show `1/1`):

```
kubectl get pods -n <YOUR_CIRCLECI_NAMESPACE>
```

VM service and Nomad server pods are expected to fail at this stage. You will set up your execution environments in the next phase of the installation.

Phase 3 - Execution environments

Before you begin with the CircleCI server v4.x execution environment installation phase, ensure you have run through Phase 1 – Prerequisites and Phase 2 - Core Services Installation.

In the following sections, replace any sections indicated by `< >` with your details.

1. Nomad clients

Nomad is a workload orchestration tool that CircleCI uses to schedule (through Nomad server) and run (through Nomad clients) CircleCI jobs.

Nomad clients are installed outside of the Kubernetes cluster, while their control plane (Nomad Server) is installed within the cluster. Communication between your Nomad Clients and the Nomad control plane is secured with mTLS. The mTLS certificate, private key, and certificate authority will be output after you complete installation of the Nomad Clients.

a. Create your cluster with Terraform

CircleCI curates Terraform modules to help install Nomad clients in your chosen cloud provider. You can browse the modules in our public repository, including example Terraform config files for both AWS and GCP.

GCP cluster

You need the following information:

- The Domain name of the CircleCI application
- The GCP Project you want to run Nomad clients in
- The GCP Zone you want to run Nomad clients in
- The GCP Region you want to run Nomad clients in
- The GCP Network you want to run Nomad clients in
- The GCP Subnetwork you want to run Nomad clients in

A full example, as well as a full list of variables, can be found in the example GCP Terraform configuration.

Once you have filled in the appropriate information, you can deploy your Nomad clients by running the following commands:

```
terraform init
```

```
terraform plan
```

```
terraform apply
```

After Terraform is done spinning up the Nomad client(s), it outputs the certificates and key needed for configuring the Nomad control plane in CircleCI server. Copy them somewhere safe.

b. Nomad Autoscaler configuration

Nomad can automatically scale up or down your Nomad clients, provided your clients are managed by a cloud provider's autoscaling resource. With Nomad Autoscaler, you need to provide permission for the utility to manage your autoscaling resource and specify where it is located. CircleCI's Nomad terraform module can provision the permissions resources, or it can be done manually.

GCP autoscaler service account

Create a service account for Nomad Autoscaler. You may take **one** of the following approaches:

Option 1: CircleCI creates the Kubernetes Secret.

The CircleCI Nomad module can create a service account and output a file with the JSON key. For this option, set the variable `nomad_auto_scaler = true`. You may reference the examples in the link for more details. The created service account key will be available in a file named `nomad-as-key.json`.

Option 2: Use Workload Identity.

The CircleCI nomad module can create a service account using Workload Identity and send out the email. Set the variables `nomad_auto_scaler = true` and `enable_workload_identity = true`.

Option 3: Create the Kubernetes Secret yourself.

When creating the Kubernetes Secret manually, an additional field is required, as outlined below.

```
# Base64 encoded additional configuration field
ADDITIONAL_CONFIG=dGFyZ2V0ICJnY2UtbWlnIiB7CiAgZHJpdmVyID0gImdjZS1taWciCiAgY29uZmlnID0gewogICAgY3JlZG
VudGlhbHMgPSAiL2V0Yy9ub21hZC1hdXRvc2NhbGVyL2NyZWRzL2djcF9zYS5qc29uIgogIH0KfQo=
kubectl create secret generic nomad-autoscaler-secret \
  --from-literal=gcp_sa.json=<service-account> \
  --from-literal=secret.hcl=$ADDITIONAL_CONFIG
```

When creating a Nomad GCP service account manually, the service account will need the role `compute.admin`. It will also need the role `iam.workloadIdentityUser` if using Workload Identity. This step is only required if you choose not to create the service account using Terraform.

2. Nomad Servers

Now that you have successfully deployed your Nomad clients and have the permission resources, you can configure the Nomad Servers.

a. Nomad gossip encryption key

Nomad requires a key to encrypt communications. This key must be exactly 32 bytes long. CircleCI will not be able to recover the values if lost. Depending on how you prefer to manage Kubernetes Secrets, there are two options:

Option 1: Create the Kubernetes Secret yourself.

```
kubectl create secret generic nomad-gossip-encryption-key \
--from-literal=gossip-key=<secret-key-32-chars>
```

Once the Kubernetes Secret exists, no change to `values.yaml` is required. The Kubernetes Secret will be referenced by default.

Option 2: CircleCI creates the Kubernetes Secret.

Add the value to `values.yaml`. CircleCI will create the Kubernetes Secret automatically.

```
nomad:
  server:
    gossip:
      encryption:
        key: "<secret-key-32-chars>"
```

b. Nomad mTLS

The `CACertificate`, `certificate` and `privateKey` can be found in the output of the terraform module. They must be base64 encoded.

```
nomad:
  server:
    ...
    rpc:
      mTLS:
        enabled: true
        certificate: "<base64-encoded-certificate>"
        privateKey: "<base64-encoded-private-key>"
        CACertificate: "<base64-encoded-ca-certificate>"
```

c. Nomad Autoscaler

If you have enabled Nomad Autoscaler, also include the following section under `nomad`:

GCP

You created these values in the Nomad Autoscaler Configuration section.

```
nomad:
  ...
  auto_scaler:
    enabled: true
    scaling:
      max: <max-node-limit>
      min: <min-node-limit>

    gcp:
      enabled: true
      project_id: "<project-id>"
      mig_name: "<instance-group-name>"

      region: "<region>"
      # or
      zone: "<zone>"

      workloadIdentity: "<service-account-email>"
      # or
      service_account: "<service-account-json>"
```

d. Helm upgrade

Apply the changes made to your `values.yaml` file:

```
namespace=<your-namespace>
helm upgrade circleci-server oci://cciserver.azurecr.io/circleci-server -n $namespace --version
4.0.0 -f <path-to-values.yaml>
```

3. Nomad Clients Validation

CircleCI has created a project called realitycheck which allows you to test your server installation. We are going to follow the project so we can verify that the system is working as expected. As you continue through the next phase, sections of realitycheck will move from red (fail) to green (pass).

Before running realitycheck, check if the Nomad servers can communicate with the Nomad clients by executing the below command.

```
kubectl -n <namespace> exec -it $(kubectl -n <namespace> get pods -l app=nomad-server -o name | tail -1) -- nomad node status
```

You should be able to see output like this:

```
ID        DC       Name              Class        Drain  Eligibility  Status
132ed55b  default  ip-192-168-44-29  linux-64bit  false  eligible     ready
```

To run realitycheck, you need to clone the repository. Depending on your GitHub setup, you can use one of the following commands:

GitHub Cloud

```
git clone https://github.com/circleci/realitycheck.git
```

GitHub Enterprise

```
git clone https://github.com/circleci/realitycheck.git
git remote set-url origin <YOUR_GH_REPO_URL>
git push
```

Once you have successfully cloned the repository, you can follow it from within your CircleCI server installation. You need to set the following variables. For full instructions please see the repository readme.

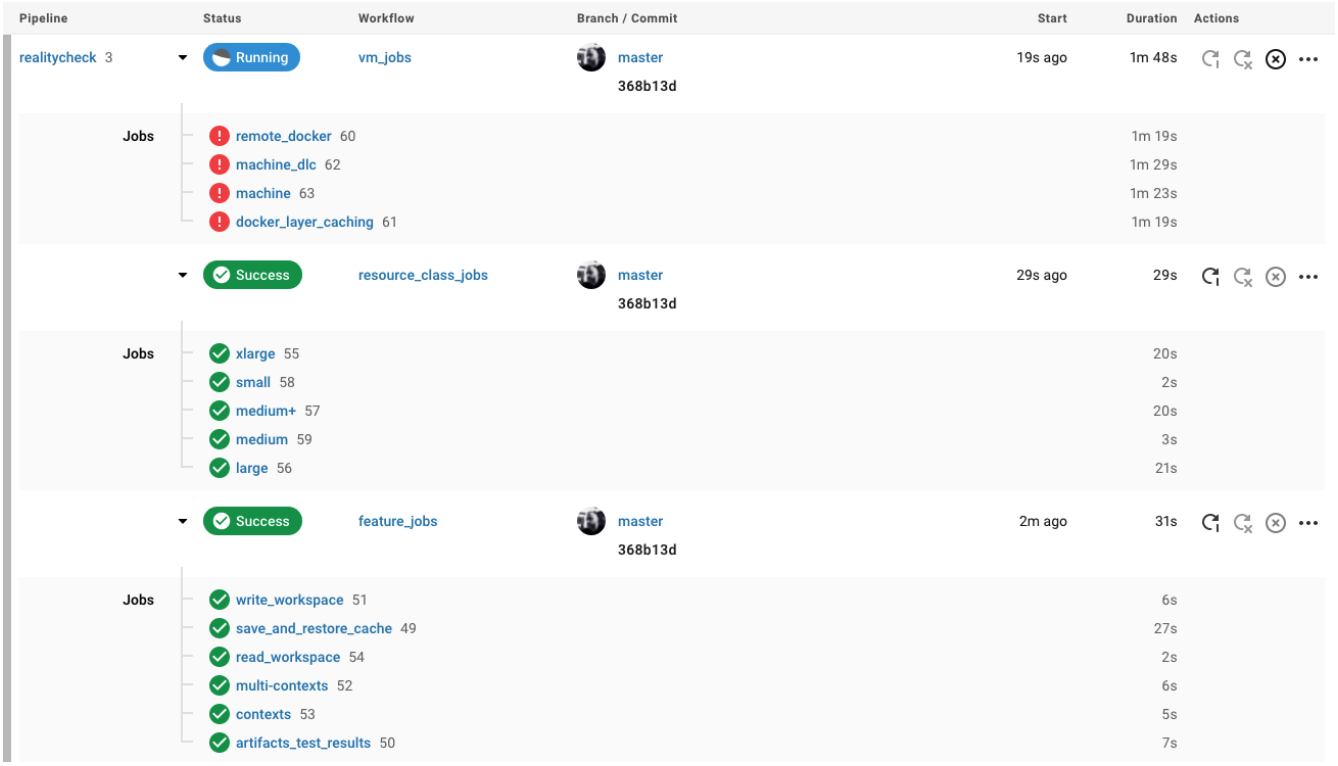
Table 1. Environmental Variables

Name	Value
CIRCLE_HOSTNAME	<YOUR_CIRCLECI_INSTALLATION_URL>
CIRCLE_TOKEN	<YOUR_CIRCLECI_API_TOKEN>

Table 2. Contexts

Name	Environmental Variable Key	Environmental Variable Value
org-global	CONTEXT_END_TO_END_TEST_VAR	Leave blank
individual-local	MULTI_CONTEXT_END_TO_END_VAR	Leave blank

Once you have configured the environmental variables and contexts, rerun the realitycheck tests. You should see the features and resource jobs complete successfully. Your test results should look something like the following:



3. VM service

VM service configures virtual machine and remote docker jobs. You can configure a number of options for VM service, such as scaling rules. VM service is unique to AWS and GCP installations because it relies on specific features of these cloud providers.

GCP

You need additional information about your cluster to complete the next section. Run the following command:

```
gcloud container clusters describe
```

This command returns something like the following, which includes network, region and other details that you need to complete the next section:

```
addonsConfig:
  gcePersistentDiskCsiDriverConfig:
    enabled: true
  kubernetesDashboard:
    disabled: true
  networkPolicyConfig:
    disabled: true
clusterIpv4Cidr: 10.100.0.0/14
createTime: '2021-08-20T21:46:18+00:00'
currentMasterVersion: 1.20.8-gke.900
currentNodeCount: 3
currentNodeVersion: 1.20.8-gke.900
databaseEncryption:
…
```

1. **Create firewall rules**

   Run the following commands to create a firewall rule for VM service in GKE:

   ```
   gcloud compute firewall-rules create "circleci-vm-service-internal-nomad-fw" --network
   "<network>" --action allow --source-ranges "0.0.0.0/0" --rules "TCP:22,TCP:2376"
   ```

   If you have used auto-mode, you can find the Nomad clients CIDR based on the region by referring to the table here.

   ```
   gcloud compute firewall-rules create "circleci-vm-service-internal-k8s-fw" --network "<network>"
   --action allow --source-ranges "<clusterIpv4Cidr>" --rules "TCP:22,TCP:2376"
   ```

   ```
   gcloud compute firewall-rules create "circleci-vm-service-external-fw" --network "<network>"
   --action allow --rules "TCP:54782"
   ```

2. **Create user**

   We recommend you create a unique service account used exclusively by VM Service. The Compute Instance Admin (Beta) role is broad enough to allow VM Service to operate. If you wish to make permissions more granular, you can use the Compute Instance Admin (beta) role documentation as reference.

   ```
   gcloud iam service-accounts create circleci-server-vm --display-name "circleci-server-vm service
   account"
   ```

If your are deploying CircleCI server in a shared VCP, you should create this user in the project in which you intend to run your VM jobs.

3. **Get the service account email address**

```
gcloud iam service-accounts list --filter="displayName:circleci-server-vm service account"
--format 'value(email)'
```

4. **Apply role to service account**

   Apply the Compute Instance Admin (beta) role to the service account:

```
gcloud projects add-iam-policy-binding <YOUR_PROJECT_ID> --member
serviceAccount:<YOUR_SERVICE_ACCOUNT_EMAIL> --role roles/compute.instanceAdmin --condition=None
```

   And

```
gcloud projects add-iam-policy-binding <YOUR_PROJECT_ID> --member
serviceAccount:<YOUR_SERVICE_ACCOUNT_EMAIL> --role roles/iam.serviceAccountUser --condition=None
```

5. **Enable Workload Identity for Service Account**

   This step is required only if you are using Workload Identities for GKE. Steps to enable Workload Identities are provided in Phase 1 - Prerequisites.

```
gcloud iam service-accounts add-iam-policy-binding <YOUR_SERVICE_ACCOUNT_EMAIL> \
    --role roles/iam.workloadIdentityUser \
    --member "serviceAccount:<GCP_PROJECT_ID>.svc.id.goog[circleci-server/vm-service]"
```

6. **Optionally, get JSON Key File**

   If you are using Workload Identities for GKE, this step is not required.

   After running the following command, you should have a file named `circleci-server-vm-keyfile` in your local working directory. You will need this when you configure your server installation.

```
gcloud iam service-accounts keys create circleci-server-vm-keyfile --iam-account
<YOUR_SERVICE_ACCOUNT_EMAIL>
```

7. **Configure Server**

   When using service account keys for configuring access for the VM service, there are two options.

   Option 1: CircleCI creates the Kubernetes Secret.

Add the VM Service configuration to values.yaml. Details of the available configuration options can be found in the Managing Virtual Machines with VM Service guide.

Option 2: Create the Kubernetes Secret yourself.

Instead of providing the service account in your `values.yaml` file, you may create the Kubernetes Secret yourself.

```
kubectl create secret generic vm-service-secret \
  --from-literal=gcp_sa.json=<access-key>
```

VM Service Validation

Apply they changes made to your values.yaml file.

```
namespace=<your-namespace>
helm upgrade circleci-server oci://cciserver.azurecr.io/circleci-server -n $namespace --version
4.0.0 -f <path-to-values.yaml>
```

Once you have configured and deployed CircleCI server, you should validate that VM Service is operational. You can rerun the realitychecker project within your CircleCI installation and you should see the VM Service Jobs complete. At this point, all tests should pass.

4. Runner

Overview

CircleCI runner does not require any additional server configuration. Server ships ready to work with runner. However, you need to create a runner and configure the runner agent to be aware of your server installation. For complete instructions for setting up runner, see the runner documentation.

Runner requires a namespace per organization. Server can have many organizations. If your company has multiple organizations within your CircleCI installation, you need to set up a runner namespace for each organization within your server installation.

Phase 4 - Post installation

Before you begin with the CircleCI server v4.x post installation phase, ensure you have run through Phase 1 – Prerequisites, Phase 2 - Core Services Installation and Phase 3 - Execution Environments Installation.

In the following sections, replace any sections indicated by `< >` with your details.

Backup and restore

Backups of CircleCI server can be created through Velero. You installed Velero in your cluster during the prerequisites installation phase.

Set up backup and restore on GCP

These instructions were sourced from the documentation for the Velero GCP plugin.

1. **Create a GCP bucket**

To reduce the risk of typos, you can set some of the parameters as shell variables. Should you be unable to complete all the steps in the same session, do not forget to reset variables as necessary before proceeding. In the step below, for example, you can define a variable for your bucket name. Replace the `<YOUR_BUCKET>` placeholder with the name of the bucket you want to create for your backups.

+

```
BUCKET=<YOUR_BUCKET>

gsutil mb gs://$BUCKET/
```

1. **Set up permissions for Velero**

If your server installation runs within a GKE cluster, ensure that your current IAM user is a cluster admin for this cluster, as RBAC objects need to be created. More information can be found in the GKE documentation.

a. Set a shell variable for your project ID. Make sure that your `gcloud` CLI points to the correct project by looking at the current configuration:

```
gcloud config list
```

If the project is correct, set the variable:

```
PROJECT_ID=$(gcloud config get-value project)
```

b. Create a service account:

```
gcloud iam service-accounts create velero \
    --display-name "Velero service account"
```

If you run several clusters with Velero, consider using a more specific name for the Service Account besides `velero`, as suggested above.

You can check if the service account has been created successfully by running the following command:

```
gcloud iam service-accounts list
```

c. Next, store the email address for the Service Account in a variable. Modify the command as needed to match the display name you have chosen for your Service Account:

```
SERVICE_ACCOUNT_EMAIL=$(gcloud iam service-accounts list \
  --filter="displayName:Velero service account" \
  --format 'value(email)')
```

Grant the necessary permissions to the Service Account:

```
ROLE_PERMISSIONS=(
    compute.disks.get
    compute.disks.create
    compute.disks.createSnapshot
    compute.snapshots.get
    compute.snapshots.create
    compute.snapshots.useReadOnly
    compute.snapshots.delete
    compute.zones.get
)

gcloud iam roles create velero.server \
    --project $PROJECT_ID \
    --title "Velero Server" \
    --permissions "$(IFS=","; echo "${ROLE_PERMISSIONS[*]}")"

gcloud projects add-iam-policy-binding $PROJECT_ID \
    --member serviceAccount:$SERVICE_ACCOUNT_EMAIL \
    --role projects/$PROJECT_ID/roles/velero.server

gsutil iam ch serviceAccount:$SERVICE_ACCOUNT_EMAIL:objectAdmin gs://${BUCKET}
```

d. Next, ensure that Velero can use this Service Account.

- **Option 1: JSON key file**

  You can simply pass a JSON credentials file to Velero to authorize it to perform actions as the Service Account. To do this, you first need to create a key:

  ```
  gcloud iam service-accounts keys create credentials-velero \
      --iam-account $SERVICE_ACCOUNT_EMAIL
  ```

  After running this command, you should see a file named `credentials-velero` in your local working directory.

- **Option 2: Workload Identities**

  If you are already using Workload Identity in your cluster, you can bind the GCP Service Account you just created to Velero's Kubernetes service account. In this case, the GCP Service Account needs the `iam.serviceAccounts.signBlob` role in addition to the permissions already specified above.

2. **Install and start Velero**

Run one of the following `velero install` commands, depending on how you authorized the service account. This creates a namespace called `velero` and installs all the necessary resources to run Velero.

Backups require restic to operate. When installing Velero, ensure that you have the `--use-restic` flag set.

- **using a JSON key file**

```
velero install \
    --provider gcp \
    --plugins velero/velero-plugin-for-gcp:v1.2.0 \
    --bucket $BUCKET \
    --secret-file ./credentials-velero \
    --use-restic \
    --wait
```

- **using Workload Identities**

```
velero install \
    --provider gcp \
    --plugins velero/velero-plugin-for-gcp:v1.2.0 \
    --bucket $BUCKET \
    --no-secret \
    --sa-annotations iam.gke.io/gcp-service-account=$SERVICE_ACCOUNT_EMAIL \
    --backup-location-config serviceAccount=$SERVICE_ACCOUNT_EMAIL \
    --use-restic \
    --wait
```

For more options on customizing your installation, refer to the Velero documentation.

3. **Verify Velero**

Once Velero is installed on your cluster, check the new `velero` namespace. You should have a Velero deployment and a restic daemonset, for example:

```
$ kubectl get pods --namespace velero
NAME                      READY   STATUS    RESTARTS   AGE
restic-5vlww              1/1     Running   0          2m
restic-94ptv              1/1     Running   0          2m
restic-ch6m9              1/1     Running   0          2m
restic-mknws              1/1     Running   0          2m
velero-68788b675c-dm2s7   1/1     Running   0          2m
```

As restic is a daemonset, there should be one pod for each node in your Kubernetes cluster.

Set up backup and restore with S3-compatible storage

The following steps assume you are using S3-compatible object storage, but not necessarily AWS S3, for your backups.

These instructions were sourced from the Velero docs.

1. **Configure `mc` client**

   To start, configure `mc` to connect to your storage provider:

   ```
   # Alias can be any name as long as you use the same value in subsequent commands
   export ALIAS=my-provider
   mc alias set $ALIAS <YOUR_MINIO_ENDPOINT> <YOUR_MINIO_ACCESS_KEY_ID>
   <YOUR_MINIO_SECRET_ACCESS_KEY>
   ```

   You can verify your client is correctly configured by running `mc ls my-provider` and you should see the buckets in your provider enumerated in the output.

2. **Create a bucket**

   Create a bucket for your backups. It is important that a new bucket is used, as Velero cannot use a preexisting bucket that contains other content.

   ```
   mc mb ${ALIAS}/<YOUR_BUCKET>
   ```

3. **Create a user and policy**

   Create a user and policy for Velero to access your bucket.

   In the following snippet `<YOUR_MINIO_ACCESS_KEY_ID>` and `<YOUR_MINIO_SECRET_ACCESS_KEY>` refer to the credentials used by Velero to access MinIO.

```
# Create user
mc admin user add $ALIAS <YOUR_MINIO_ACCESS_KEY_ID> <YOUR_MINIO_SECRET_ACCESS_KEY>

# Create policy
cat > velero-policy.json << EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:*"
      ],
      "Resource": [
        "arn:aws:s3:::<YOUR_BUCKET>",
        "arn:aws:s3:::<YOUR_BUCKET>/*"
      ]
    }
  ]
}
EOF

mc admin policy add $ALIAS velero-policy velero-policy.json

# Bind user to policy
mc admin policy set $ALIAS velero-policy user=<YOUR_VELERO_ACCESS_KEY_ID>
```

Finally, you add your new user's credentials to a file (`./credentials-velero` in this example) with the following contents:

```
[default]
aws_access_key_id=<YOUR_VELERO_ACCESS_KEY_ID>
aws_secret_access_key=<YOUR_VELERO_SECRET_ACCESS_KEY>
```

4. **Install and start Velero**

Run the following `velero install` command. This creates a namespace called `velero` and installs all the necessary resources to run Velero.

Backups require restic to operate. When installing Velero, ensure that you have the `--use-restic` flag set, as shown below:

```
velero install --provider aws \
  --plugins velero/velero-plugin-for-aws:v1.2.0 \
  --bucket <YOUR_BUCKET> \
  --secret-file ./credentials-velero \
  --use-volume-snapshots=false \
  --use-restic \
  --backup-location-config region=minio,s3ForcePathStyle="true",s3Url=<YOUR_ENDPOINT> \
  --wait
```

5. **Verify Velero**

   Once Velero is installed on your cluster, check the new `velero` namespace. You should have a Velero deployment and a restic daemonset, for example:

```
$ kubectl get pods --namespace velero
NAME                      READY   STATUS    RESTARTS   AGE
restic-5vlww              1/1     Running   0          2m
restic-94ptv              1/1     Running   0          2m
restic-ch6m9              1/1     Running   0          2m
restic-mknws              1/1     Running   0          2m
velero-68788b675c-dm2s7   1/1     Running   0          2m
```

As restic is a daemonset, there should be one pod for each node in your Kubernetes cluster.

Take a backup

Now that Velero is installed on your cluster, you are ready to create your first backup. If you encounter problems, please refer to the troubleshooting section.

- To create the backup, run:

```
K8S_NS=$(helm list -o yaml  | yq '.[].namespace')
CHART=$(helm list -o yaml  | yq '.[].chart' )
REV=$(helm list -o yaml  | yq '.[].revision')
RANDOM_STR=$(cat /dev/urandom | env LC_ALL=C tr -dc 'a-z0-9' | head -c 8)

velero backup create "${K8S_NS}-${RANDOM_STR}" --include-namespaces "${K8S_NS}" --labels "chart--
rev=${CHART}--${REV}"
```

- To restore from a backup, run:

```
# List all existing backups
velero backup get --show-labels

# Restore the specific backup
velero restore create --include-namespaces <circleci-namespace> --from-backup <backup-name>
```

See the Velero documentation or more details.

Email Notifications

Add email notification support by adding the following to `values.yaml`:

```
smtp:
  host: <hostname-of-submission-server>
  user: <username-for-submission-server>
  password: <password-for-submission-server
  port: <mail-port>
```

Managing orbs

Server installations include their own local orb registry. This registry is private to the server installation. All orbs referenced in project configs reference the orbs in the *server* orb registry. You are responsible for maintaining orbs. This includes:

- Copying orbs from the public registry.
- Updating orbs that may have been copied previously.
- Registering your company's private orbs, if you have any.

For more information and steps to complete these tasks, see the Orbs on server guide.

Hardening your cluster

This section provides supplemental information on hardening your Kubernetes cluster.

Network topology

A server installation basically runs three different type of compute instances: The Kubernetes nodes, Nomad clients, and external VMs.

It is highly recommended that you deploy these into separate subnets with distinct CIDR blocks. This will make it easier for you to control traffic between the different components of the system and isolate them from each other.

Best practice is to make as many of the resources as private as possible. If your users will access your CircleCI server installation via VPN, there is no need to assign any public IP addresses at all, as long as you have a working NAT gateway setup. Otherwise, you will need at least one public subnet for the `circleci-proxy` load balancer.

However, in this case, it is also recommended to place Nomad clients and VMs in a public subnet to enable your users to SSH into jobs and scope access via networking rules.

Currently, custom subnetting is not supported for GCP. Custom subnetting support will be available in a future update/release.

An NGINX reverse proxy is placed infront of Kong and exposed as a Kubernetes service named `circleci-proxy`. NGINX is responsible routing the traffic to the following services: `kong`, `vm-service`, `output-processor` and `nomad`.

When using Amazon Certificate Manager (ACM), the name of nginx's service will be `circleci-proxy-acm` instead of `circleci-proxy`. If you have switched from some other method of handling your TLS certificates to using ACM, this change will recreate the loadbalancer and you will have to reroute your associated DNS records for your <domain> and app.<domain>.

Network traffic

This section explains the minimum requirements for a server installation to work. Depending on your workloads, you might need to add additional rules to egress for Nomad clients and VMs. As nomenclature between cloud providers differs, you will probably need to implement these rules using firewall rules and/or security groups.

Where you see "external," this usually means all external IPv4 addresses. Depending on your particular setup, you might be able to be more specific (for example, if you are using a proxy for all external traffic).

The rules explained here are assumed to be stateful and for TCP connections only, unless stated otherwise. If you are working with stateless rules, you need to create matching ingress or egress rules for the ones listed here.

Reverse proxy status

You may wish to check the status of the services routing traffic in your CircleCI server installation and alert if there are any issues. Since we use both Nginx and Kong in CircleCI server, we expose the status pages of both via port 80.

Service	Endpoint
nginx	`/nginx_status`
kong	`/kong_status`

Kubernetes load balancers

Depending on your setup, your load balancers might be transparent (that is, they are not treated as a distinct layer in your networking topology). In this case, you can apply the rules from this section directly to the underlying destination or source of the network traffic. Refer to the documentation of your cloud provider to make sure you understand how to correctly apply networking security rules, given the type of load balancing you are using with your installation.

Ingress

If the traffic rules for your load balancers have not been created automatically, here are their respective ports:

Name	Port	Source	Purpose
circleci-proxy/-acm	80	External	User Interface & Frontend API
circleci-proxy/-acm	443	External	User Interface & Frontend API
circleci-proxy/-acm	3000	Nomad clients	Communication with Nomad clients
circleci-proxy/-acm	4647	Nomad clients	Communication with Nomad clients
circleci-proxy/-acm	8585	Nomad clients	Communication with Nomad clients

Egress

The only type of egress needed is TCP traffic to the Kubernetes nodes on the Kubernetes load balancer ports (30000-32767). This is not needed if your load balancers are transparent.

Common rules for compute instances

These rules apply to all compute instances, but not to the load balancers.

Ingress

If you want to access your instances using SSH, you will need to open port 22 for TCP connections for the instances in question. It is recommended to scope the rule as closely as possible to allowed source IPs and/or only add such a rule when needed.

Egress

You most likely want all of your instances to access internet resources. This requires you to allow egress for UDP and TCP on port 53 to the DNS server within your VPC, as well as TCP ports 80 and 443 for HTTP and HTTPS traffic, respectively. Instances building jobs (that is, the Nomad clients and external VMs) also will likely need to pull code from your VCS using SSH (TCP port 22). SSH is also used to communicate with external VMs, so it should be allowed for all instances with the destination of the VM subnet and your VCS, at the very least.

Kubernetes nodes

Intra-node traffic

By default, the traffic within your Kuberntes cluster is regulated by networking policies. For most purposes, this should be sufficient to regulate the traffic between pods and there is no additional requirement to reduce traffic between Kubernetes nodes any further (it is fine to allow all traffic between Kubernetes nodes).

To make use of networking policies within your cluster, you may need to take additional steps, depending on your cloud provider and setup. Here are some resources to get you started:

- Kuberenetes Network Policy Overview
- Creating a Cluster Network Policy on Google Cloud
- Installing Calico on Amazon EKS

Ingress

If you are using a managed service, you can check the rules created for the traffic coming from the load balancers and the allowed port range. The standard port range for Kubernetes load balancers (30000-32767) should be all that is needed here for ingress. If you are using transparent load balancers, you need to apply the ingress rules listed for load balancers above.

Egress

Port	Destination	Purpose
2376	VMs	Communication with VMs
4647	Nomad clients	Communication with the Nomad clients
all traffic	other nodes	Allow intra-cluster traffic

Nomad Clients

Nomad clients do not need to communicate with each other. You can block traffic between Nomad client instances completely.

Ingress

Port	Source	Purpose
4647	K8s nodes	Communication with Nomad server
64535-65535	External	Rerun jobs with SSH functionality

Egress

Port	Destination	Purpose
2376	VMs	Communication with VMs
3000	VM Service load balancers	Internal communication
4647	Nomad Load Balancer	Internal communication
8585	Output Processor Load Balancer	Internal communication

External VMs

Similar to Nomad clients, there is no need for external VMs to communicate with each other.

Ingress

Port	Source	Purpose
22	Kubernetes nodes	Internal communication
22	Nomad clients	Internal communication
2376	Kubernetes nodes	Internal communication
2376	Nomad clients	Internal communication
54782	External	Rerun jobs with SSH functionality

Egress

You will only need the egress rules for internet access and SSH for your VCS.

Installing server behind a proxy

Depending on your security requirements, you might want to install CircleCI server behind a proxy. Installing behind a proxy gives you the power to monitor and control access between your installation and the broader Internet.

Configuring a proxy happens during Phase 2 - Core services.

Known limitations

- The CircleCI hostname must be added to the no-proxy list because the following services: `output processor` and `vm-service`. The application and build-agent share a no-proxy list and are assumed to be behind the same firewall and therefore cannot have a proxy between them.
- Some additional configuration is required to import orbs when installed behind a proxy. See Orbs on Server docs for more information.
- The JVM only accepts proxies that run over HTTP, not HTTPS, and therefore proxy URIs must be of the form `http://user:password@host:port` rather than `https://user:password@host:port`.
- If your GitHub instance is running outside of the proxied environment (either GitHub.com or GitHub Enterprise), you must ensure that SSH traffic from CircleCI (inside the Kubernetes cluster) and from our Nomad node can reach your instance. Please note the default `checkout` step in a CircleCI job will fail to clone code and our `ssh-keyscan` of GitHub Enterprise will not work. While you may configure an SSH proxy, `ssh-keyscan` can NOT be proxied and instead will require you provide `github.fingerprint` when using GHE.
- If you install server behind a proxy, you may need to provide a custom image for VM service. Visit the CircleCI Linux Image Builder repo for further information.
- If object storage is outside the proxy, no job features that use object storage will work. This includes:
  - Artifacts
  - Test results
  - Cache save and restore
  - Workspaces

  Users can get around this restriction by setting environment variables on their jobs. For example:

```
jobs:
  my-job:
    docker:
      - image: cimg/node:17.2.0
        environment:
          HTTP_PROXY: http://proxy.example.com:3128
          HTTPS_PROXY: http://proxy.example.com:3128
          NO_PROXY: whatever.internal,10.0.1.2
```

It is crucial that these environment variables are set in this specific location because it is the only location that propagates them to the correct service.

Migrate from server v3.x to v4.x

Migrating from server v3.x to v4.x is an *in-place* migration. You will generate a helm-value file, which will be used in a helm command to upgrade the CircleCI server from v3.x to v4.x.

We recommend using a staging environment before completing this process in a production environment. This will allow you to gain a better understanding of the migration process, and give you an idea of how long the migration will take to complete.

Prerequisites

- Your current CircleCI server installation is v3.x.
- You have taken a backup of the v3.x instance, following the instructions at the Backup and Restore guide.
    - If you are using external datastores, they need to be backed up separately.
    - If you are using internal datastores they are backed up using the backup process described in the link above.
- The migration script must be run from a machine with following tools installed:
    - kubectl configured for the server 3.x instance
    - yq
    - helm
    - helm-diff
    - Azure CLI

Migration

To perform an in-place upgrade of CircleCI server from v3.x to v4.x, you will execute a migration script, which will perform the following tasks:

- Export the KOTS config values into a YAML file.
- Modify the exported YAML file to be used with `helm` to migrate to server v4.x.
- Annotate all the kubernetes resources for `helm`.
- Execute postgres db `domain` migration.
- Execute cleanup steps which will delete all the KOTS related resources.
- Display an output message with the next steps.

In CircleCI server v4.x, we have changed the way we store workflow data. Previously we had stored your workflow data in Postgres. We have moved this data into your storage bucket (GCS/S3/MinIO). After migrating, there will be a period where your workflow data will be unavailable. This is due to your existing workflow data being transferred to your storage bucket. The time taken depends on how much existing workflow data you have, however, you may scale the `workflow-conductor` pods that are responsible for the migration to speed up the process. You may scale as much as your cluster will allow. In later steps we will show you how to scale your pods.

The kots-exporter script can also perform the following functions if required:

- Rerun the annotation step only (`-f annotate`)
- Run the kots-annotation cleanup job (`-f kots_cleanup`)
- Display the output message (`-f message`)
- Rerun the PostgreSQL database migration (`-f flyway`)

1. Create docker secret for CircleCI image registry

The migration script requires access to the CircleCI image registry to run the migration function. The image registry is also used in the server v4.x installation. Create the `docker-registry` secret in the same Kubernetes namespace in-which CircleCI server is installed. Image registry credentials will be provided by your CircleCI point of contact.

In a terminal run the following, substituting all values indicated by `< >`:

```
kubectl -n <namespace> create secret docker-registry regcred \
  --docker-server=https://cciserver.azurecr.io \
  --docker-username=<image-registry-username> \
  --docker-password=<image-registry-token> \
  --docker-email=<notification-email-id>
```

2. Clone the repository and run the kots-exporter script

The instructions below guide you through cloning the repository containing the KOTS exporter script. This script generates the `helm-value` file for server 4.x.

1. Run `git clone https://github.com/CircleCI-Public/server-scripts`.
2. Change into the `kots-exporter` directory: `cd server-scripts/kots-exporter`.
3. Run the migration script: `./kots-exporter.sh`.
4. You will be prompted for the following information:
    - Release name of your server 3.x installation - `circleci-server`
    - Kubernetes namespace of your server 3.x installation
    - License String for Server 4.x, which will be provided by your CircleCI point of contact.

5. After the script has completed, you will get a message detailing the next steps to perform, for example:

```
############ Upgrade Commands ################
Before upgrading to 4.0, follow below steps:

Your helm values file with your installation config is here:
- <path>/kots-exporter/output/helm-values.yaml
--------------------------------------------------------------------------
## Progres Chart Upgrade Preparation
Upgrading to CircleCI server v4.x includes upgrading the Postgres chart.
Before upgrading, we need to prepare your Postgres instance.
This is only needed if your Postgres instance is not externalized.

# Collect your postgres user's password and the PVC attached to your postgres instance
export POSTGRESQL_PASSWORD=$(kubectl get secret --namespace <namespace> postgresql -o jsonpath
="{.data.postgresql-password}" | base64 --decode)
export POSTGRESQL_PVC=$(kubectl get pvc --namespace <namespace> -l app.kubernetes.io/instance
=circleci-server,role=primary -o jsonpath="{.items[0].metadata.name}")
# remove the postgres statefulset without terminating your postgres instance
kubectl delete statefulsets.apps postgresql --namespace <namespace> --cascade=orphan
# remove the existing secret containing your postgres password. This will get recreated during
upgrade.
kubectl delete secret postgresql --namespace <namespace>
--------------------------------------------------------------------------
## Helm Diff
The Helm Diff tool is used to verify that the changes between your current install and the upgrade
are expected

# diff command
helm diff upgrade <release-name> -n <namespace> -f <path>/kots-exporter/output/helm-values.yaml
--show-secrets --context 5 oci://cciserver.azurecr.io/circleci-server --version 4.0.0
--------------------------------------------------------------------------
## Helm Upgrade CircleCI Server
helm upgrade <release-name> -n <namespace> -f <path>/kots-exporter/output/helm-values.yaml
oci://cciserver.azurecr.io/circleci-server --version <version-to-upgrade-to> --force

NOTE: After server 3.x to 4.x migration, You must rerun the Nomad terraform with modified value of
'server_endpoint' variable
It should be - <domain-name>:4647
```

If your Postgres instance is not externalized, when upgrading to CircleCI Server v4.0, the Postgres chart is also being upgraded. Please note the command needed before running helm upgrade above.

3. Validate your helm-value file

Once the migration script has completed, a `helm-values.yaml` file will be generated with your existing CircleCI Server 3.x config. This file holds the configuration data you had previously entered in KOTS. Going forward, you will use this file to update/configure your CircleCI Server installation as is standard helm practice.

4. Generate helm-diff output

Next, execute the helm-diff command and review the output.

```
helm diff upgrade <release-name> -n <namespace> -f <path>/kots-exporter/output/helm-values.yaml
--show-secrets --contexts 5 oci://cciserver.azurecr.io/circleci-server --version 4.0.0
```

Review the output generated from the `helm-diff` command using the following to help:

- line highlighted `Yellow`: Kubernetes resources status, for example, `changed`, `added`
- line highlighted `Red`: Deletion, for example, `image`
- line displayed in `Green`: Addition, for example, `imagePullSecret`

Below are the changes you should expect to see in `helm-diff` output:

- `imagePullSecrets` is added into all the Kubernetes resources
- Container images are updated
- Secret environment variables (for example api-token, signing-keys) now reference Kubernetes secrets
- Environment variables for RabbitMQ and MongoDB URIs change
- Environment variables for VM, OUTPUT and NOMAD service uri now reference `<domain_name>:<service_port>`
- Annotations from VM, OUTPUT and NOMAD service resources are deleted
- Github checksum is added as annotation
- Secret and annotation for `distributor-*` deployments are deleted
- Upstream chart `postgresql` is updated
- Upsteam charts will be recreated (delete and create):
    - Prometheus (circleci-server-kube-state-metrics, node-exporter, prometheus-server)
    - MongoDB
    - RabbitMQ
    - Redis (redis-master, redis-slave)

5. Upgrading CircleCI Server 3.x

Once your helm-value file is verified, run the following commands to upgrade the CircleCI server to v4.x.

Our helm registry is stored in an azure private registry. You will be provided a username and token to access the registry.

```
USER_NAME=<token-username>
PASSWORD=<token>
namespace=<your-install-namespace>
helm upgrade circleci-server -n <namespace> -f <path>/kots-exporter/output/helm-values.yaml
oci://cciserver.azurecr.io/circleci-server --version 4.0.0 --force --username $USER_NAME --password
$PASSWORD
```

6. Check upgrade status

Run the following command to check all pods are up and running:

```
kubectl -n <namespace> get pods
```

7. Update DNS setting

Server 4.x migration is a destructive change for your DNS configuration. Server 4.x replaces the need for 4 load balancers and 5 DNS records with a single`load-balancer/external-ip` service, named `circleci-proxy` or `circleci-proxy-acm`. This load balancer only needs to be routed via 2 DNS records, <your-domain> and app.<your-domain>. Separate domains for vm-service, output-processer and nomad are no longer required. Retrieve the external IP/Loadbalancer and update your DNS records accordingly.

```
kubectl -n <namespace> get svc circleci-proxy

# AWS Provider: XXXXX.elb.XXXXX.amazonaws.com
# GCP Provider: XXX.XXX.XXX.XXX
```

The following Kubernetes service objects are renamed/changed:

- circleci-server-traefik (LoadBalancer) → kong (ClusterIP)
- nomad-server-external (LoadBalancer) → nomad-server (ClusterIP)
- output-processor (LoadBalancer) → output-processor (ClusterIP)
- vm-service (LoadBalancer) → vm-service (ClusterIP)

The following Kubernetes service object is added:

- circleci-proxy or circleci-proxy-acm (LoadBalancer)

8. Execute Nomad Terraform

Execute the Nomad Terraform to re-create nomad client where `server_endpoint` value is set to be `<domain>:4647`. You can follow the steps mentioned here. Update the helm value file with generated Certificates and Keys (base64 encoded) for Nomad Sever-Client communication.

9. Validate your migration to server v4.x

Re-run realitycheck on your new server 4.x environment by pushing a fresh commit.

10. Update your team

Once you have successfully run realitycheck, notify your team about the upgrade.