OPERATIONS GUIDE

A guide for administrators of CircleCI Server installations on AWS and private infrastructure.

docs@circleci.com

Version 3.0, 02/09/2021: beta-DRAFT

Overview

CircleCI Server v3.x is a continuous integration and continuous delivery (CI/CD) platform that you can install on your GCP or AWS Kubernetes cluster. Refer to the Changelog for what's new in this release.

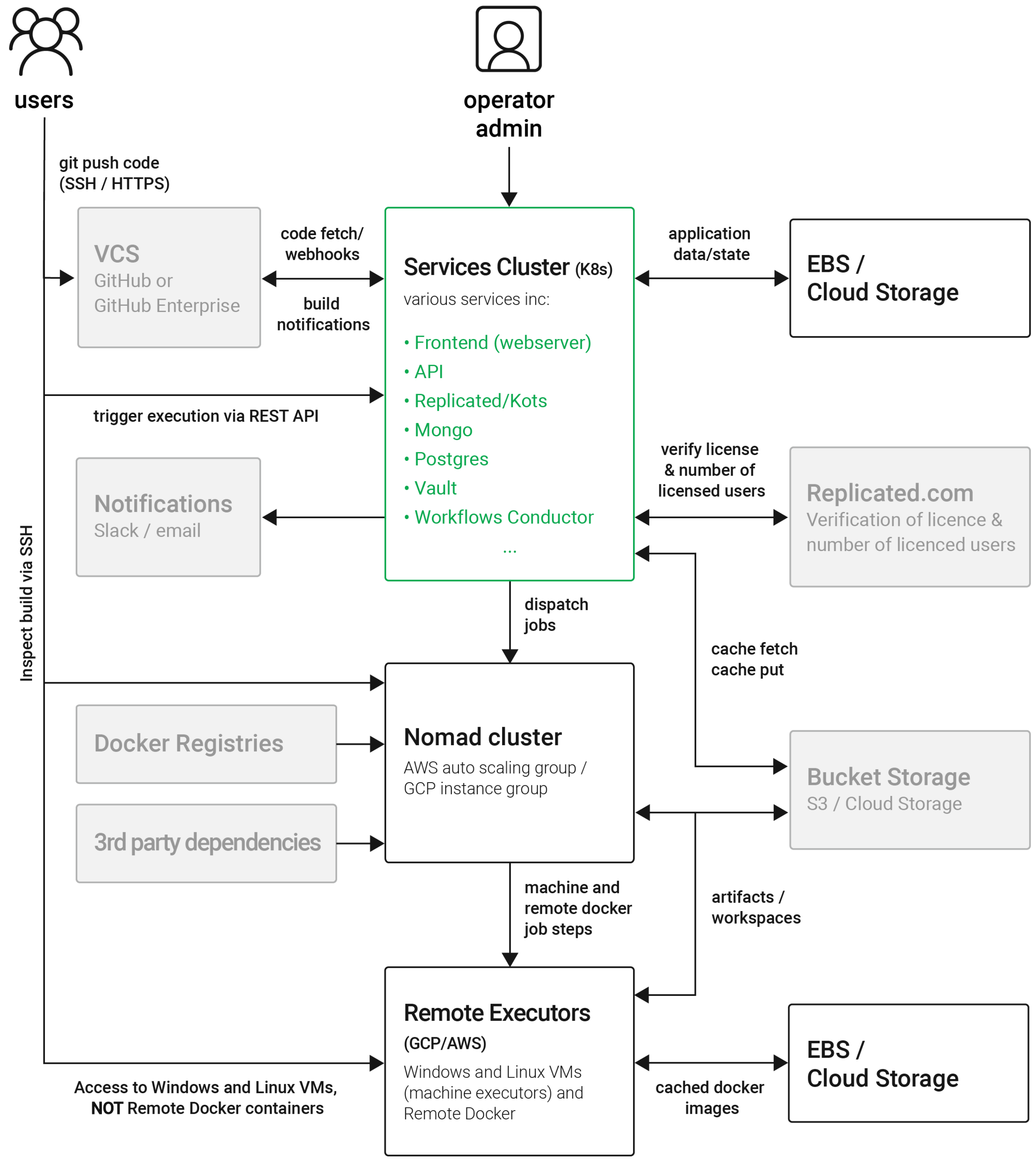


Figure 1. Server v3.x Architecture

Architecture

The image above illustrates CircleCI server v3.x core components. Some notable sections:

- The CircleCI API is a full-featured RESTful API that allows you to access all information and trigger all actions in CircleCI.
- CircleCI consists of two primary components: Services and Nomad Clients. Any number of Nomad Clients execute your jobs and communicate back to the Services.
- All components must access GitHub or your hosted instance of GitHub Enterprise on the network.

Services Cluster

All services that make up the CircleCI server v3.0-beta installation run in pods within your K8s cluster. For a full list of services, along with role information, see the Services Architecture guide.

Build Environment

CircleCI Server v3.0-beta uses Nomad as the primary job scheduler. Refer to our Introduction to Nomad Cluster Operation to learn more about the job scheduler and how to perform basic client and cluster operations.

By default, CircleCI Nomad clients automatically provision compute resources according to the executors configured for each job in a project's `.circleci/config.yml` file.

Nomad Clients

Nomad Clients run without storing state, enabling you to increase or decrease the number of containers as needed.

To ensure enough Nomad clients are running to handle all builds, track the queued builds and increase the number of Nomad Client machines as needed to balance the load. For more on tracking metrics see Monitoring Your Installation.

Each machine reserves two vCPUs and 4GB of memory for coordinating builds. The remaining processors and memory create the containers for running jobs. Larger machines are able to run more containers and are limited by the number of available cores after two are reserved for coordination.

The maximum machine size for a Nomad client is 128GB RAM/ 64 CPUs. Contact your CircleCI account representative to request use of larger machines for Nomad Clients.

The following table describes the ports used on Nomad clients:

Source	Ports	Use
End Users	64535-65535	SSH into builds
Administrators	80 or 443	CCI API Access
Administrators	22	SSH
Services Machine	all traffic, all ports	Internal Comms

Source	Ports	Use
Nomad Clients (including itself)	all traffic, all ports	Internal Comms

GitHub

CircleCI uses GitHub or GitHub Enterprise credentials for authentication, which, in turn, may use SAML or SSH for access. This means CircleCI will inherit the authentication supported by your central SSO infrastructure.

CircleCI does not support changing the URL or backend GitHub instance after it has been set up.

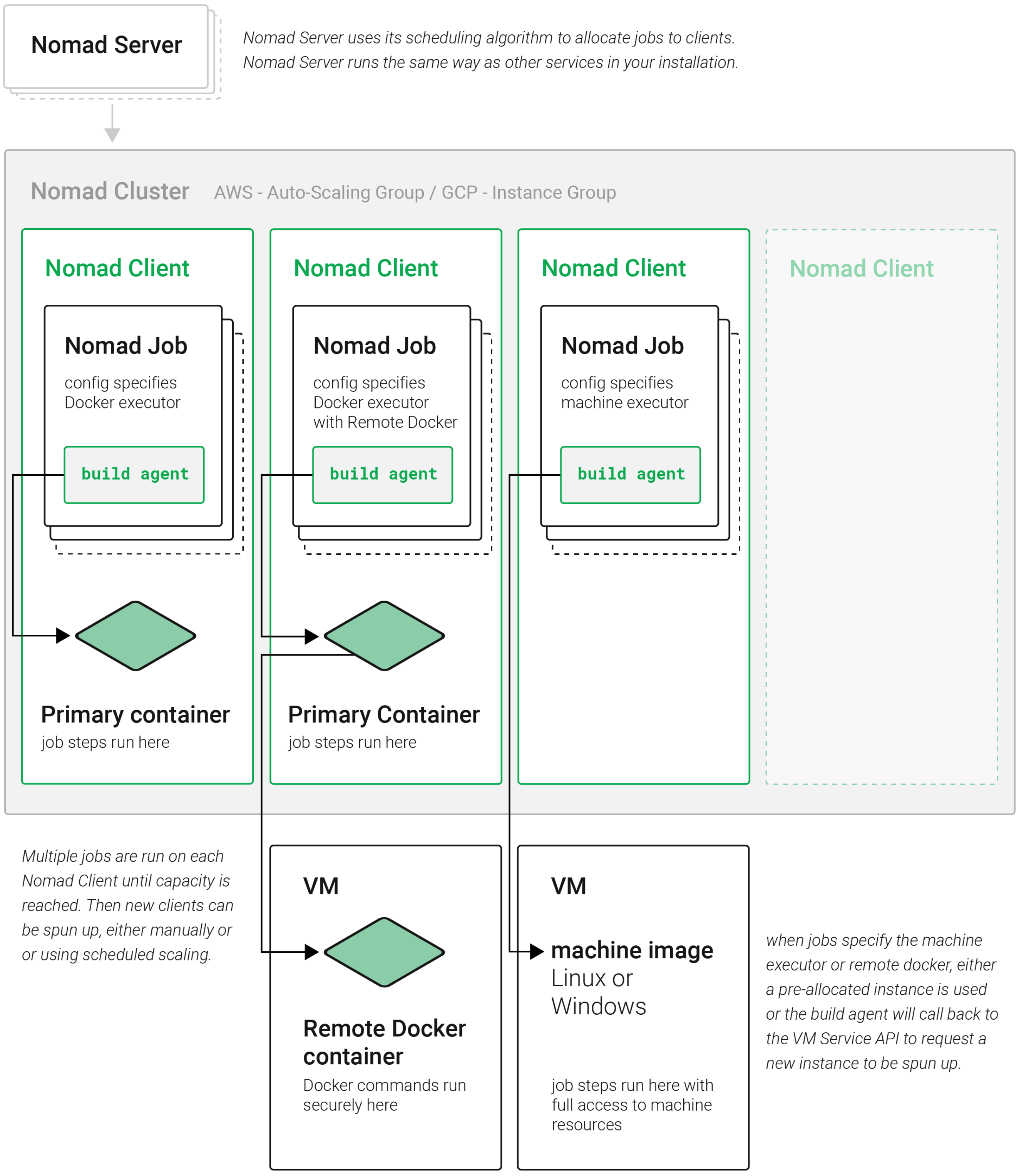
The following table describes the ports used on machines running GitHub to communicate with the Services and Nomad Client instances.

Source	Ports	Use
Services	22	Git Access
Services	80, 443	API Access
Nomad Client	22	Git Access
Nomad Client	80, 443	API Access

Introduction to Nomad Cluster Operation

CircleCI uses Nomad as the primary job scheduler. This guide provides a basic introduction to Nomad along with guidance on how to operate and manage the Nomad Cluster in your CircleCI server v3.x installation.

Basic Terminology and Architecture



Nomad Server
Nomad Server uses its scheduling algorithm to allocate jobs to clients.
Nomad Server runs the same way as other services in your installation.
Nomad Cluster
AWS - Auto-Scaling Group / GCP - Instance Group
Nomad Client
Nomad Job
config specifies Docker executor
build agent
Primary container
job steps run here
Nomad Client
Nomad Job
config specifies Docker executor with Remote Docker
build agent
Primary Container
job steps run here
Nomad Client
Nomad Job
config specifies machine executor
build agent
Nomad Client
Multiple jobs are run on each Nomad Client until capacity is reached. Then new clients can be spun up, either manually or or using scheduled scaling.
VM
Remote Docker container
Docker commands run securely here
VM
machine image
Linux or Windows
job steps run here with full access to machine resources
when jobs specify the machine executor or remote docker, either a pre-allocated instance is used or the build agent will call back to the VM Service API to request a new instance to be spun up.

Figure 2. Nomad Cluster Management

- **Nomad server:** The Nomad server is the brains of the cluster, receiving and allocating jobs to Nomad clients. In CircleCI server v3.x, a Nomad server runs as a service within your cluster.
- **Nomad client:** Nomad clients execute jobs allocated to them by the Nomad server. Usually a Nomad client runs on a dedicated machine (often a VM) in order to fully take the advantage of machine power. You can have multiple Nomad clients to form a cluster and the Nomad server allocates jobs to the cluster using its scheduling algorithm.
- **Nomad jobs:** A Nomad job is a specification, provided by a user, that declares a workload for Nomad. In CircleCI, a Nomad job corresponds to an execution of a CircleCI job. If the job configuration specifies the use of parallelism, say x10 parallelism, then Nomad will run 10 jobs.
- **Build agent:** The build agent is a Go program written by CircleCI that executes steps in a job and reports the results. The build agent is executed as the main process inside a Nomad Job.

Basic Operations

The following sections form a basic guide to operating a Nomad cluster in your installation.

The `nomad` CLI is installed in the `circleci-nomad-0` pod within your cluster, and the commands described below can be used to inspect and manage your Nomad cluster.

You can also access individual Nomad clients if you have provided a valid SSH key when running through the Terraform setup process.

Checking the Jobs Status

To get a status list for all jobs in your cluster, run:

```
kubectl exec circleci-nomad-0 -it -- nomad status
```

The `status` is the most important field in the output, with the following status type definitions:

- `running`: Nomad has started executing the job. This typically means your job in CircleCI is started.
- `pending`: There are not enough resources available to execute the job inside the cluster at this time.
- `dead`: Nomad has finished executing the job. The status becomes `dead` regardless of whether the corresponding CircleCI job/build succeeds or fails.

Checking the cluster status

To get a list of your Nomad clients, run:

```
kubectl exec circleci-nomad-0 -it -- nomad node-status
```

`nomad node-status` reports both Nomad clients that are currently serving (status `active`) and Nomad clients that were taken out of the cluster (status `down`). Therefore, you need to count the number of `active` Nomad clients to know the current capacity of your cluster.

To get more information about a specific client, run the following from that client:

A valid SSH key must be provided during the initial Terraform setup during your installation for these steps to work.

1. SSH into the client
2. Run the following command to get status information for the client:

```
nomad node-status -self
```

These steps will give information such as how many jobs are running on the client and the resource utilization of the client.

Checking logs

As noted in the Nomad jobs section above, a Nomad job corresponds to an execution of a CircleCI job. Therefore, Nomad job logs can sometimes help to understand the status of a CircleCI job if there is a problem. To get logs for a specific job:

1. Run the following to get the Nomad job ID:

```
kubectl exec circleci-nomad-0 -it -- nomad status
```

2. Run the following, substituting the Nomad job ID you got in step 1 to view logs for that job:

```
kubectl exec circleci-nomad-0 -it -- nomad logs -job -stderr <job-id>
```

Be sure to specify the `-stderr` flag as this is where most build agent logs appear.

While the `nomad logs -job` command is useful, the command is not always accurate because the `-job` flag uses a random allocation of the specified job. The term `allocation` is a smaller unit in Nomad Job, which is out of scope for this document. To learn more, please see the official document.

Complete the following steps to get logs from the allocation of the specified job:

1. Get the job ID:

```
kubectl exec circleci-nomad-0 -it -- nomad status
```

2. Get the allocation ID of the job by running

```
kubectl exec circleci-nomad-0 -it -- nomad status <job-id>
```

3. Get the logs from the allocation by running:

```
kubectl exec circleci-nomad-0 -it -- nomad logs -stderr <allocation-id>
```

Scaling the Cluster

By default, your Nomad cluster is set up within an Auto Scaling Group (ASG) for AWS installs, or an Instance Group for GCP installs. To view settings follow the steps for your infrastructure:

AWS

1. Go to your EC2 Dashboard and select Auto Scaling Groups from the left hand menu
2. Select your Nomad Client
3. Select Actions > Edit to set Desired/Minimum/Maximum counts. This defines the number of Nomad Clients to spin up and keep available. Use the Scaling Policy tab to scale up your group automatically at your busiest times, see below for best practices for defining scaling policies. Use nomad job metrics to assist in defining your scaling policies.

GCP

1. Go to the GCP console, select Compute Engine and then Instance Groups
2. Find your Nomad cluster. The prefix will be your `BASENAME` and the name will contain the word `nomad`
3. Select Edit Group
4. You now have the option to either set a number of nodes to have up and running at all times, or you can click Configure Autoscaling to set up autoscaling. For more information on setting a GCP autoscaling policy see the GCP Documentation.

Scaling Policy Best Practices

There is a blog post series wherein CircleCI engineering spent time running simulations of cost savings for the purpose of developing a general set of best practices for Auto Scaling. Consider the following best practices when setting up Auto Scaling:

1. In general, size your cluster large enough to avoid queueing builds. That is, less than one second of queuing for most workloads and less than 10 seconds for workloads run on expensive hardware or at highest parallellism. Sizing to reduce queuing to zero is best practice because of the high cost of developer time. It is difficult to create a model in which developer time is cheap enough for under-provisioning to be cost-effective.
2. Create an Auto Scaling Group / Instance Group with a Step Scaling policy that scales up during the normal working hours of the majority of developers and scales back down at night. Scaling up during the weekday normal working hours and back down at night is the best practice to keep queue times down during peak development, without over provisioning at night when traffic is low. Looking at millions of builds over time, a bell curve during normal working hour emerges for most data sets.

This is in contrast to auto-scaling throughout the day based on traffic fluctuations, because modelling revealed that boot times are actually too long to prevent queuing in real time. Use Amazon’s Step Policy instructions to set this up along with Cloudwatch Alarms.

Shutting Down a Nomad Client

When you want to shut down a Nomad client, you must first set the client to `drain`. In `drain` mode, the client will finish any jobs that have already been allocated but will not be allocated any new jobs.

A valid SSH key must be provided during the initial Terraform setup during your installation for these steps to work.

1. To drain a client, SSH into the client and set the client to drain mode with `node-drain` command as follows:

   ```
   nomad node-drain -self -enable
   ```

2. Then, make sure the client is in drain mode using the `node-status` command:

   ```
   nomad node-status -self
   ```

Alternatively, you can drain a remote node from the Nomad server with the following command, substituting the node ID:

```
kubectl exec circleci-nomad-0 -it -- nomad node-drain -enable -yes <node-id>
```

Scaling Down the Client Cluster

To set up a mechanism for clients to shutdown, first enter `drain` mode, then wait for all jobs to be finished before terminating the client. If your installation is within AWS, you can also configure an ASG Lifecycle Hook that triggers a script for scaling down instances.

The script should use the commands in the section above to do the following:

1. Put the instance in drain mode

   ```
   nomad node-drain -self -enable
   ```

2. Monitor running jobs on the instance and wait for them to finish
3. Terminate the instance

Monitoring your Installation

This section includes information on metrics for monitoring your CircleCI server v3.0-beta installation.

Metrics Overview

Metrics are technical statistical data collected for monitoring and analytics purposes. The data includes basic information, such as CPU or memory usage, as well as more advanced counters, such as number of executed builds and internal errors. Using metrics you can:

- Quickly detect incidents and abnormal behavior
- Dynamically scale compute resources
- Retroactively understand infrastructure-wide issues

How Metrics Work in CircleCI Server v3.x

Telegraf is the main component used for metrics collection in CircleCI server. Telegraf is server software that brokers metrics data emitted by CircleCI services to data monitoring platforms such as Datadog.

Metrics collection in CircleCI Server works as follows:

- Each component sends metrics data to the Telegraf service.
- Telegraf listens on port 8125/UDP and receives data from all components (inputs) and applies configured filters to determine whether data should be kept or dropped.
- For some metric-types, Telegraf keeps data internal and periodically calculates statistical data, such as max, min, mean, stdev, sum.
- Finally, Telegraf sends out data to configured sinks (outputs), such as stdout and Datadog.

It is worth noting that Telegraf can accept multiple input and output types at the same time allowing administrators to configure a single Telegraf instance to collect and forward multiple metrics data sets to Datadog.

Supported Platforms

CircleCI server v3.0-beta has built-in support for forwarding metrics to Datadog.

Datadog

To enable Datadog complete the following steps - you will need your Datadog API key:

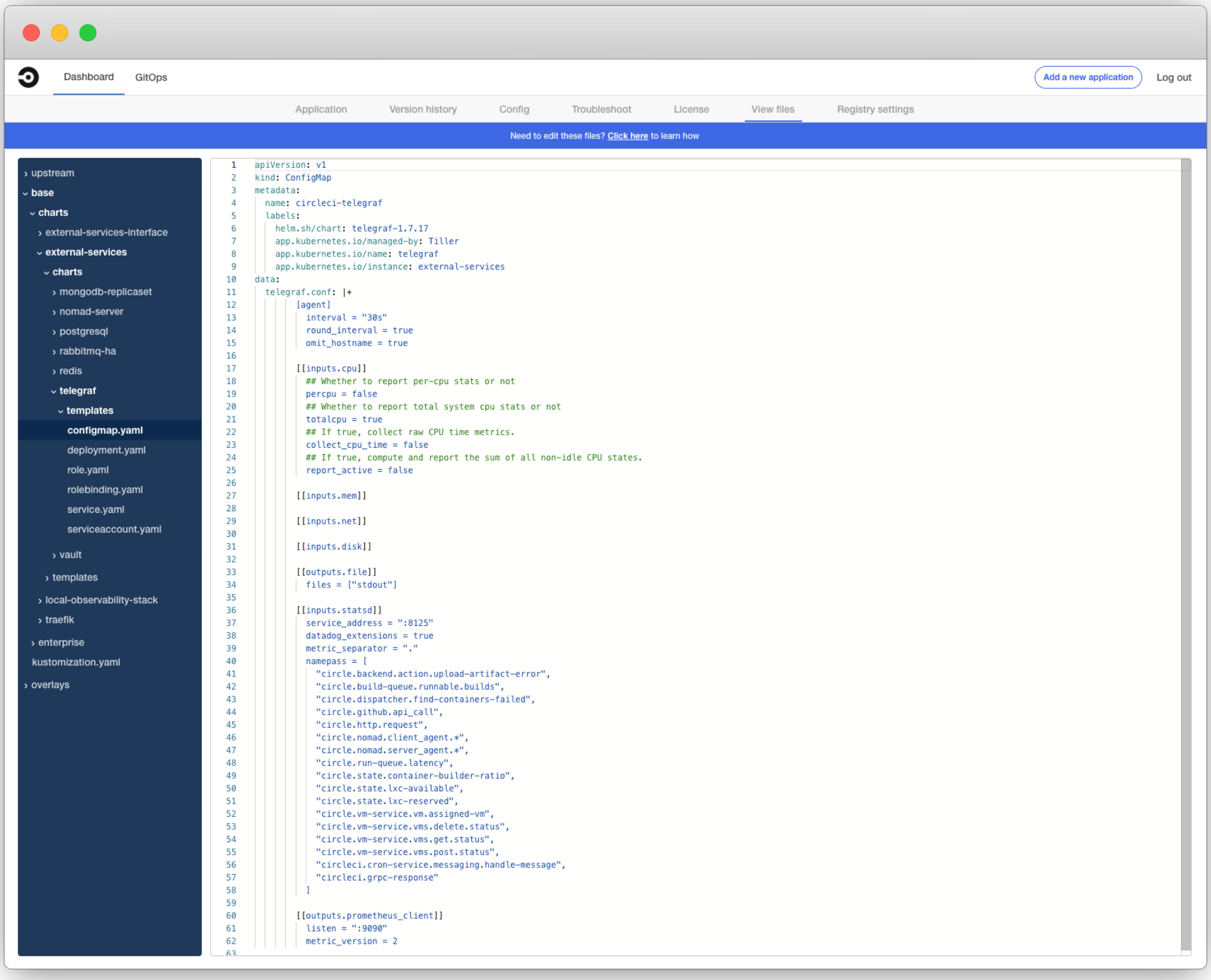
1. Navigate your Admin Dashboard (localhost:8800)
2. Select the **Config** tab
3. Scroll to the **Enable Metrics** section
4. Complete the settings for this section:
    a. Check the box to enable metrics forwarding to Datadog
    b. Enter a metrics prefix (optional)

c. Enter your Datadog API key (required)

Standard Metrics Configuration

Follow these steps to review the standard metrics configuration for your installation:

1. Navigate to the Admin Dashboard
2. Select the **View Files** tab
3. Navigate to the Telegraf Config Map – `base/charts/external-services/charts/telegraf/templates/configmap.yaml`



```
apiVersion: v1
kind: ConfigMap
metadata:
  name: circleci-telegraf
  labels:
    helm.sh/chart: telegraf-1.7.17
    app.kubernetes.io/managed-by: Tiller
    app.kubernetes.io/name: telegraf
    app.kubernetes.io/instance: external-services
data:
  telegraf.conf: |+
        [agent]
          interval = "30s"
          round_interval = true
          omit_hostname = true

        [[inputs.cpu]]
          ## Whether to report per-cpu stats or not
          percpu = false
          ## Whether to report total system cpu stats or not
          totalcpu = true
          ## If true, collect raw CPU time metrics.
          collect_cpu_time = false
          ## If true, compute and report the sum of all non-idle CPU states.
          report_active = false

        [[inputs.mem]]

        [[inputs.net]]

        [[inputs.disk]]

        [[outputs.file]]
          files = ["stdout"]

        [[inputs.statsd]]
          service_address = ":8125"
          datadog_extensions = true
          metric_separator = "."
          namepass = [
            "circle.backend.action.upload-artifact-error",
            "circle.build-queue.runnable.builds",
            "circle.dispatcher.find-containers-failed",
            "circle.github.api_call",
            "circle.http.request",
            "circle.nomad.client_agent.*",
            "circle.nomad.server_agent.*",
            "circle.run-queue.latency",
            "circle.state.container-builder-ratio",
            "circle.state.lxc-available",
            "circle.state.lxc-reserved",
            "circle.vm-service.vm.assigned-vm",
            "circle.vm-service.vms.delete.status",
            "circle.vm-service.vms.get.status",
            "circle.vm-service.vms.post.status",
            "circleci.cron-service.messaging.handle-message",
            "circleci.grpc-response"
          ]

        [[outputs.prometheus_client]]
          listen = ":9090"
          metric_version = 2
```

Figure 3. Metrics Configuration

For detailed information on each element contained in the Telegraph configuration file, see the Telegraf docs.

Custom Metrics Configuration

If your installation requires modifications to the standard metrics setup it is possible to create an overlay to make your changes. CircleCI only supports the standard setup. Therefore, **if you do make changes it is**

important you have engineering capacity to support this going forwards. To create a custom setup, follow these steps:

1. Navigate your Admin Dashboard (localhost:8800)
2. Select the **View Files** tab
3. At the top of the **View Files** window within the blue notification bar click the link to learn how to download a file bundle and follow the instructions
4. Create a kustomize overlay containing the configuration you require
5. Save the overlay in the overlays/midstream folder and follow the in-app instructions to upload

You will see your changes listed under the **Version History** tab.

System Monitoring Metrics

The following sections give an overview of available metrics for your installation.

VM Service and Docker Metrics

VM Service and Docker services metrics are forwarded via Telegraf, a plugin-driven server agent for collecting and reporting metrics.

The following metrics are enabled:

- CPU
- Disk
- Memory
- Networking
- Docker

Nomad Job Metrics

Nomad job metrics are enabled and emitted by the Nomad Server agent. Five types of metrics are reported:

Metric	Description
`circle.nomad.server_agent.poll_failure`	Returns 1 if the last poll of the Nomad agent failed, otherwise it returns 0.
`circle.nomad.server_agent.jobs.pending`	Returns the total number of pending jobs across the cluster.
`circle.nomad.server_agent.jobs.running`	Returns the total number of running jobs across the cluster.
`circle.nomad.server_agent.jobs.complete`	Returns the total number of complete jobs across the cluster.
`circle.nomad.server_agent.jobs.dead`	Returns the total number of dead jobs across the cluster.

When the Nomad metrics container is running normally, no output will be written to standard output or standard error. Failures will elicit a message to standard error.

CircleCI Metrics

`circle.backend.action.upload-artifact-error`	Tracks how many times an artifact has failed to upload.
`circle.build-queue.runnable.builds`	Tracks how many builds flowing through the system are considered runnable.
`circle.dispatcher.find-containers-failed`	Tracks how many 1.0 builds
`circle.github.api_call`	Tracks how many api calls CircleCI is making to github
`circle.http.request`	Tracks the response codes to CircleCi requests
`circle.nomad.client_agent.*``	Tracks nomad client metrics
`circle.nomad.server_agent.*`	Tracks how many nomad servers there are.
`circle.run-queue.latency`	Tracks how long it takes for a runnable build to be accepted.
`circle.state.container-builder-ratio`	Keeps track of how many containers exist per builder (1.0 only).
`circle.state.lxc-available`	Tracks how many containers are available (1.0 only)
`circle.state.lxc-reserved`	Tracks how many containers are reserved/in use (1.0 only).
`circleci.cron-service.messaging.handle-message`	Provides timing and counts for RabbitMQ messages processed by the `cron-service`
`circleci.grpc-response`	Tracks latency over the system grpc system calls.

Tips and Troubleshooting

Check Metrics are Reported

You can check that services/pods are reporting metrics correctly by checking they are being reported to stdout. To do this, examine the logs of the `circleci-telegraf` pod using `kubectl logs` or a tool like stern. To view logs for Telegraf:

1. Run `kubectl get pods` to get a list of services
2. Run `kubectl logs -f circleci-telegraf-<hash>` substituting the hash for your installation

While monitoring the current log stream, perform some actions with your server installation, e.g. logging out/in or running a workflow. These activities should be logged, showing that metrics are being reported.

Most metrics you will see logged will be from the `frontend` pod. However, when you run workflows, you should also see metrics reported by `dispatcher`, `legacy-dispatcher`, `output-processor` and `workflows-conductor`, as well as metrics concerning cpu, memory and disk stats.

Check Logs

You may check the logs by running `kubectl logs circleci-telegraf-<hash> -n <namespace> -f` to confirm your output provider (e.g. influx) is listed in the configured outputs.

Tag Metrics

If you would like to ensure that all metrics in your installation are tagged against an environment you could place the following code in your config:

```
[global_tags]
Env="<staging-circleci>"
```

Read the InfluxDB documentation for default and advanced installation steps.

Authentication

This guide describes how people can access your CircleCI server v3.0-beta installation and authenticate their accounts. CircleCI server currently supports OAuth through GitHub or GitHub Enterprise.

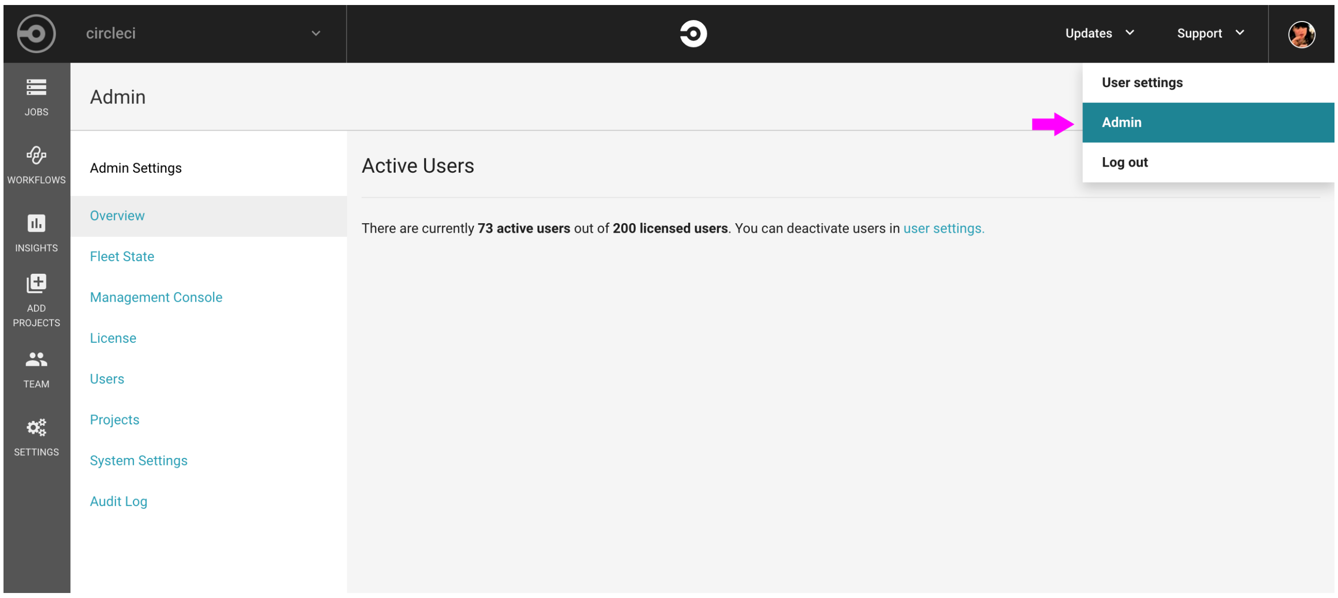
OAuth with GitHub/GitHub Enterprise

The default method for user account authentication in CircleCI server is through GitHub.com/GitHub Enterprise OAuth.

After your installation is up and running, provide users with a link to access the CircleCI application - for example `<your-circleci-hostname>.com` – and they will be prompted to set up an account by running through the GitHub/GitHub Enterprise OAuth flow before being redirected to the CircleCI login screen.

Managing User Accounts in Server Installations

This guide provides information to help system administrators of CircleCI server installations manage accounts for their users. For an overview of user accounts, view the Admin settings overview from the CircleCI app by clicking on your profile in the top right corner and selecting Admin. This overview provides the active user count and the total number of licensed users.



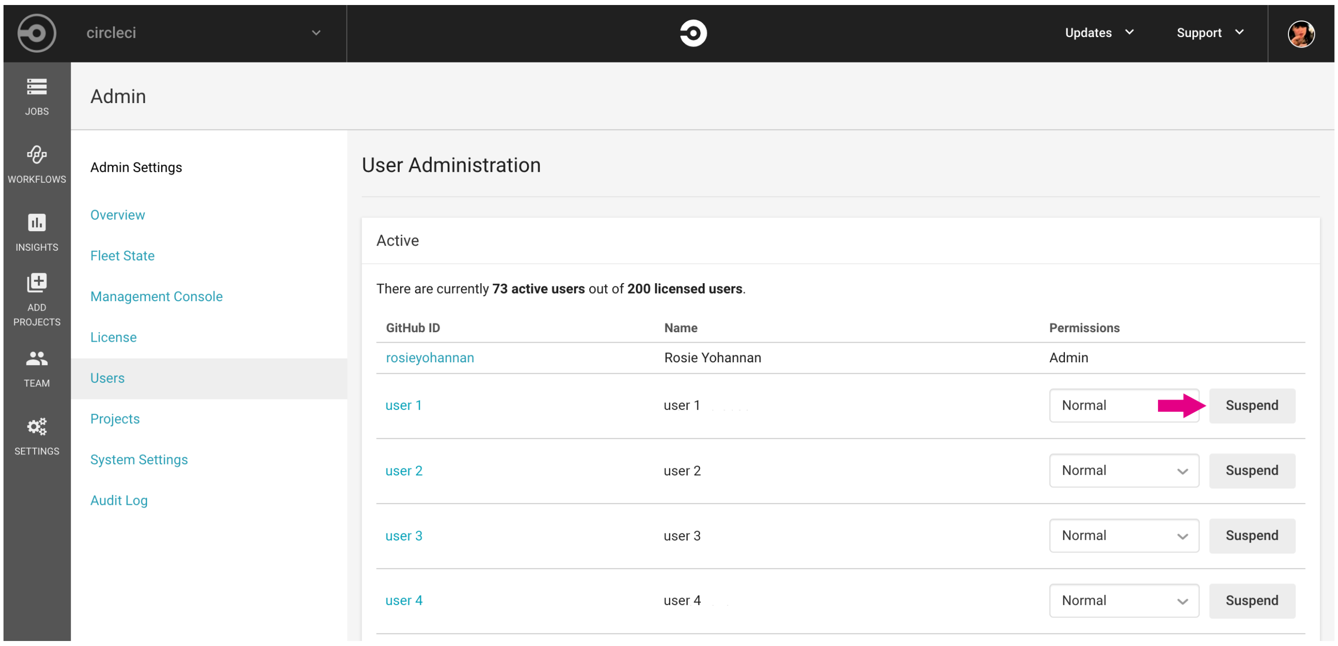
There are currently **73 active users** out of **200 licensed users**. You can deactivate users in user settings.

Figure 4. Admin Settings – Account Overview

Suspending Accounts

When an account is no longer required, you can suspend the account so it will no longer be active and will not count against your license quota. To suspend an account:

1. Navigate to your CircleCI Admin Settings
2. Select Users from the Admin Settings menu
3. Scroll to locate the account in either the Active or Inactive window
4. Click `Suspend` next to the account name and the account will appear in the Suspended window



User Administration

Active

There are currently **73 active users** out of **200 licensed users**.

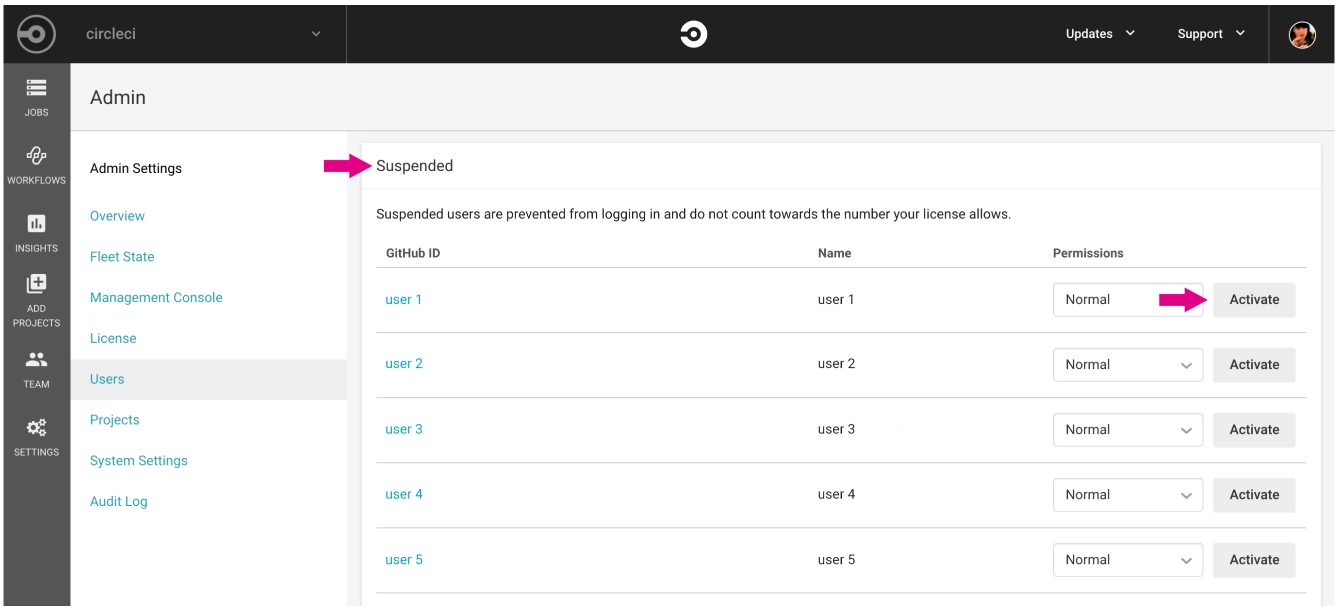
GitHub ID	Name	Permissions	
rosieyohannan	Rosie Yohannan	Admin	
user 1	user 1	Normal	Suspend
user 2	user 2	Normal	Suspend
user 3	user 3	Normal	Suspend
user 4	user 4	Normal	Suspend

Figure 5. Suspending an Account

Reactivating a Suspended User Account

To reactivate an account that has been suspended:

1. Navigate to your CircleCI Admin Settings
2. Select Users from the Admin Settings menu
3. View the Suspended window
4. Click on `Activate` next to the User you wish to grant access and the account will appear in the Active window



Suspended

Suspended users are prevented from logging in and do not count towards the number your license allows.

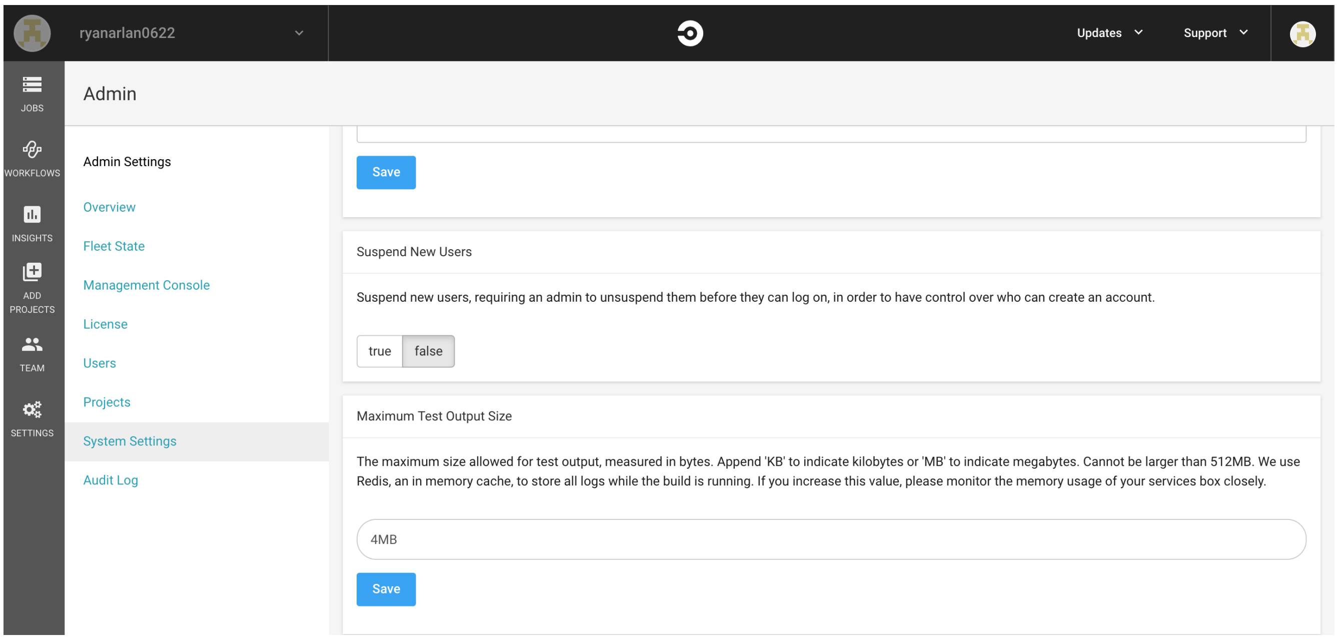
GitHub ID	Name	Permissions	
user 1	user 1	Normal	Activate
user 2	user 2	Normal	Activate
user 3	user 3	Normal	Activate
user 4	user 4	Normal	Activate
user 5	user 5	Normal	Activate

Figure 6. Rectivate Existing Users

Controlling Account Access

Any user associated with your GitHub.com or GitHub Enterprise organization can create a user account for your CircleCI installation. In order to control who has access, you can automatically suspend **all** new users, requiring an administrator to activate them before they can log in. To access this feature:

1. Navigate to your CircleCI Admin Settings
2. Select System Settings from the Admin Settings menu
3. Set Suspend New Users to `True`



Suspend New Users

Suspend new users, requiring an admin to unsuspend them before they can log on, in order to have control over who can create an account.

true [✓ false]

Maximum Test Output Size

The maximum size allowed for test output, measured in bytes. Append 'KB' to indicate kilobytes or 'MB' to indicate megabytes. Cannot be larger than 512MB. We use Redis, an in memory cache, to store all logs while the build is running. If you increase this value, please monitor the memory usage of your services box closely.

4MB

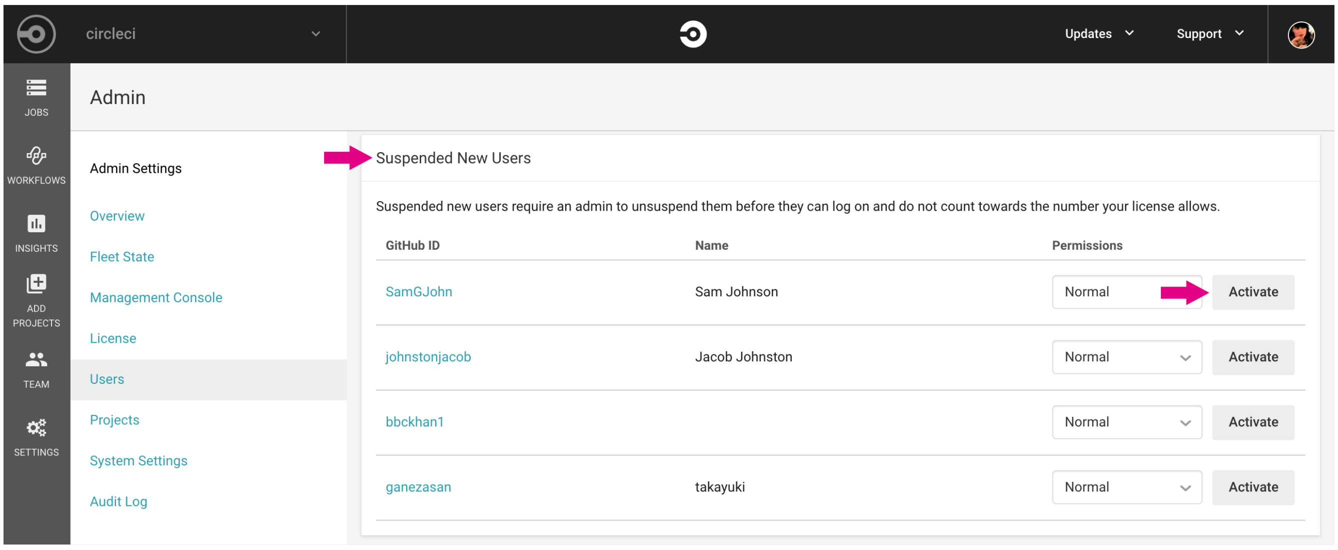
Save

Figure 7. Auto Suspend New Users

Activating a Suspended New User Account

To activate an **new** account that was automatically suspended, and allow the associated user access to your installation of CircleCI Server:

1. Navigate to your CircleCI Admin Settings
2. Select Users from the Admin Settings menu
3. View the Suspended New Users window
4. Click on `Activate` next to the User you wish to grant access and the account will appear in the Active window



Suspended New Users

Suspended new users require an admin to unsuspend them before they can log on and do not count towards the number your license allows.

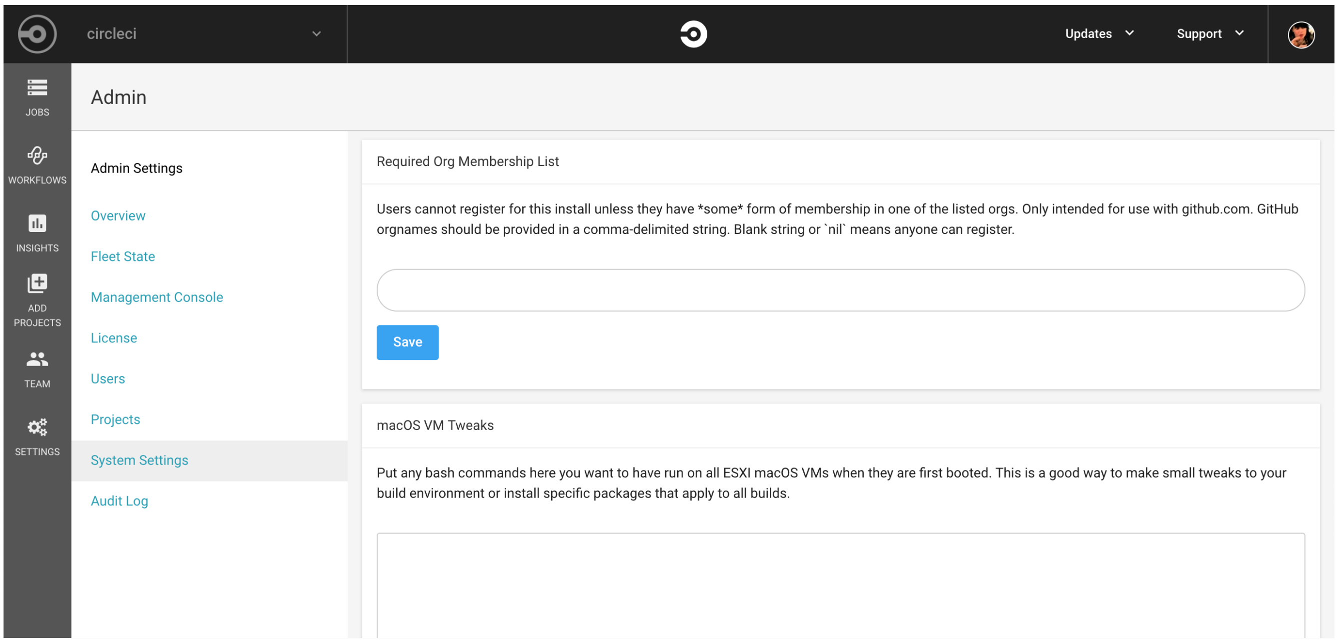
GitHub ID	Name	Permissions	
SamGJohn	Sam Johnson	Normal	Activate
johnstonjacob	Jacob Johnston	Normal	Activate
bbckhan1		Normal	Activate
ganezasan	takayuki	Normal	Activate

Figure 8. Activate a Suspended New User

Limit User Registrations by GitHub Organization

When using github.com, you can limit who can register with your CircleCI install to people with **some** connection to your approved organizations list. To access this feature:

1. Navigate to your CircleCI Admin Settings page
2. Select System Settings from the Admin Setting menu
3. Scroll down to Required Org Membership List
4. Enter the organization(s) you wish to approve. If entering more than one organization, use a comma delimited string



Required Org Membership List

Users cannot register for this install unless they have *some* form of membership in one of the listed orgs. Only intended for use with github.com. GitHub orgnames should be provided in a comma-delimited string. Blank string or `nil` means anyone can register.

macOS VM Tweaks

Put any bash commands here you want to have run on all ESXI macOS VMs when they are first booted. This is a good way to make small tweaks to your build environment or install specific packages that apply to all builds.

Figure 9. Organization Membership

Any form of organization membership is within the scope of this approval feature, and it does not stop users from running builds associated with other organizations they may belong to.

Full User List

To view a full list of users for your CircleCI Server installation, first SSH into your Services machine, and then run:

```
circleci dev-console
(circle.model.user/where { :$and [{:sign_in_count {:$gte 0}}, {:login {:$ne nil}}]} :only [:login])
```

Deleting a User

If you need to remove a user from your installation of CircleCI Server, you will need to SSH into the services machine first and then delete using the following command, substituting the user's GitHub username:

```
circleci dev-console
(circle.http.api.admin-commands.user/delete-by-login-vcs-type! "github-username-of-user" :github)
```

Orbs

This guide describes Orbs and how to manage them. Server installations include their own local orb registry. This registry is private to the server installation. All orbs referenced in 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 prior, and registering your companies private orbs if they exist.

Managing Orbs

Orbs are accessed via the CircleCI CLI. Providing a local repository location using the `--host` option allows you to access your local server orbs vs the public cloud orbs. For example, if your server installation is located at `http://circleci.somehostname.com`, then you can run orb commands local to that orb repository by passing `--host http://cirlceci.somehostname.com.`

For a list of public orbs, visit the orb directory or you can run:

```
circleci orb list
```

For a list of orbs registered in your local server orb repository:

```
circleci orb list --host <your server installation domain>
```

To import a public orb to your local server orb repository:

```
circleci orb import ns[/orb[@version]] --host <your server installation domain>
```

To update a public orb in your local server orb repository with a new version:

```
Circleci orb import ns[/orb[@version]] --host <your server installation domain>
```

For more Orb documentation please refer to the Orb docs for the cloud product.

Troubleshooting server v3.x installations

This document describes an initial set of troubleshooting steps to take if you are having problems with your CircleCI server v3.x installation. If your issue is not addressed below, you can generate a support bundle and contact your CircleCI account team.

Start Admin Console

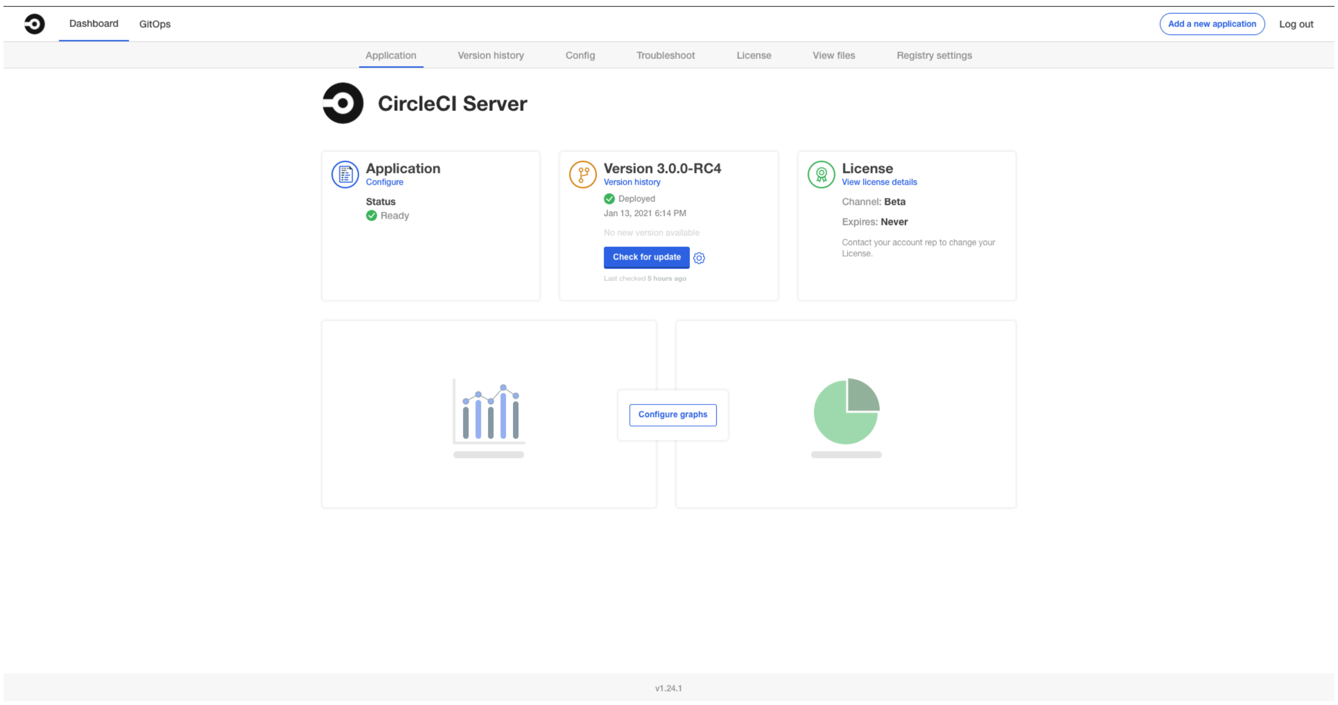
To check the Server status and download support bundle, you can you run the the following command to start Admin console:

```
$ kubectl kots admin-console --namespace <your-namespace>
```

Allows you to access Admin console over bastion host using SSH port forwarding.

```
$ ssh -i ~/.ssh/<Private Key of bastion instance> -NL 8800:localhost:8800 ubuntu@<BASTION_IP>
```

Open your browser and access http://localhost:8800 to see the Admin console



Application

Status
✓ Ready

Version 3.0.0-RC4

✓ Deployed
Jan 13, 2021 6:14 PM
No new version available
Last checked 5 hours ago

License

Channel: Beta
Expires: Never
Contact your account rep to change your License.

Figure 10. Server v3.x Admin Console

Generating a Support Bundle

To download a support bundle, select the **Troubleshoot** tab from the admin console menu bar, and then click **Analyze CircleCI Server**. CircleCI engineers will often request support bundles to help diagnose and fix any problems you are experiencing.

Check Pods status

Run the below command on bastion instance, and if you see pods aren't ready, please check the detailed logs of those pods. Note: please check the `READY` column as well as `STATUS`, because, even if the `STATUS` is `Running`, pods are not ready to server user requests. Some pods may take some time to become ready.

```
$ kubectl get pods -n <name_space>
NAME                                                             READY   STATUS      RESTARTS    AGE
api-service-5c8f557548-zjbsj                                     1/1     Running     0           6d20h
audit-log-service-77c478f9d5-5dfzv                               1/1     Running     0           6d20h
builds-service-v1-5f8568c7f5-62h8n                               1/1     Running     0           6d20h
circleci-mongodb-0                                               1/1     Running     0           6d20h
circleci-nomad-0                                                 1/1     Running     6           6d20h
...
```

To show only pods with other than `Running` status, you can use `--field-selector` option.

```
$ kubectl get pods --field-selector status.phase!=Running -n circleci-eks
NAME                                     READY    STATUS       RESTARTS    AGE
local-observability-stack-loki-stack-test  0/1      Error        0           5d22h
```

To show detail settings and status of pods

```
$ kubectl describe pods <pod_name> -n <name_space>
```

To show logs of pods

```
$ kubectl logs <pod_name> -n <name_space>
```

To restart specific pods, the easiest way is remove the pod, then Kubernetes automatically recreates the pod again.

```
kubectl delete pod <pod_name> -n <name_space> --now
```

Debug Queuing Builds

If your Services component is fine, but builds are not running, or all builds are queueing, follow the steps below.

1. Check Legacy Dispatcher Logs for Errors

Run `kubectl logs -l app=circle-legacy-dispatcher -n <name_space> -f`, then making a new build. If you see log output that is free of errors you may continue on the next step.

The following output indicates that legacy-dispatcher is up and running correctly:

```
Jan 20 06:14:59.119:+0000 INFO circle.backend.build.usage-queue got usage-queue event;
Jan 20 06:14:59.129:+0000 INFO circle.backend.build.usage-queue forwarded to run-queue; build-
name=circleci/realitycheck/21;
Jan 20 06:14:59.129:+0000 INFO circle.backend.build.usage-queue publishing event;
```

2. Check Picard-Dispatcher Logs for Errors

Run the `kubectl logs -l app=dispatcher -n <name_space>` command. A healthy `picard-dispatcher` should output the following:

```
2021-01-20T06:14:59.277+0000 [] [async-thread-macro-1] INFO  picard-dispatcher.core popped build
from build queue;
2021-01-20T06:14:59.281+0000 [] [async-thread-macro-1] INFO  circle.model.build Performed
compression of config-str;
2021-01-20T06:14:59.347+0000 [] [async-thread-macro-1] INFO  picard-dispatcher.tasks Build agent
image;
2021-01-20T06:14:59.347+0000 [] [async-thread-macro-1] INFO  picard-dispatcher.tasks Build is using
resource class; build-id=6007ca6352718c10b1227a57; build-url
=https://<SERVER_DOMAIN>/gh/circleci/realitycheck/21; resource-class={:id "d1.medium", :availability
:general, :ui {:cpu 2.0, :ram 4096, :class :medium, :name "Docker Medium"}, :outer {:cpu 2.0, :ram
4120}}; vcs-url=https://github.com/circleci/realitycheck;
2021-01-20T06:14:59.348+0000 [] [async-thread-macro-1] INFO  picard-dispatcher.tasks Computed tasks;
2021-01-20T06:15:42.005+0000 [] [main] INFO  picard-dispatcher.init Still running...;
2021-01-20T06:16:42.006+0000 [] [main] INFO  picard-dispatcher.init Still running...;
```

The output should be filled with the above messages. If it is a slow day and builds are not happening very often, the output will appear as follows:

```
2021-01-20T06:16:42.006+0000 [] [main] INFO  picard-dispatcher.init Still running...;
```

As soon as you run a build, you should see the above message to indicate that it has been dispatched to the scheduler. If you do not see the above output or you have a stack trace in the picard-dispatcher container, contact support@circleci.com.

If you run builds and do not see messages mentioned above in log outputs from picard-dispatcher, it often indicates that a job is getting lost between the legacy dispatcher and the picard dispatcher.

Delete two pods and recreate them to re-establish the connection between the two containers. Then wait for the two pods to start up.

```
$ kubectl delete pod <legacy-dispatcher_pod_name> <dispatcher_pod_name> -n <name_space> now
$ kubectl get pods -n <name_space> | grep dispatcher
```

3. Check Picard-Scheduler Logs for Errors

Run `kubectl logs -l app=schedulerer -n <name_space>`. The `picard-scheduler` schedules jobs and sends them to Nomad through a direct connection. It does not actually handle queuing of the jobs in CircleCI.

A healthy `picard-scheduler` should output the following:

```
2021-01-20T06:14:59.553+0000 [] [pool-43-thread-13] INFO  schedulerer.core got task message; task-
name=6007ca6352718c10b1227a57-0-build-circleci-realitycheck-21; task={:disk-mb 0, :task-id
"6007ca6352718c10b1227a57-0-build", :resource_class_id "d1.medium", :name "6007ca6352718c10b1227a57-
0-build-circleci/realitycheck/21", :build-agent-image "1.0.48635-2552da21", :ram 4120,
:event_processor_uri "http://output.<BASENAME>.circleci.internal:8585", :resource-class-id
"d1.medium", :ram-mb 4120, :executor "docker", :build_agent_image "1.0.48635-2552da21", :metadata
{:build_url "https://<SERVER_DOMAIN>/gh/circleci/realitycheck/21", :env "development", :debug
false}, :cpus 2.0}; version=0.1.53171
2021-01-20T06:14:59.554+0000 [] [pool-43-thread-13] INFO  schedulerer.scheduler posting task to
nomad;
2021-01-20T06:14:59.567+0000 [] [pool-43-thread-13] INFO  schedulerer.scheduler posted to nomad;
2021-01-20T06:15:07.896+0000 [] [main] INFO  schedulerer.core Still running...;
```

4. Check Nomad Node Status

Check to see if there are any Nomad nodes by running `kubectl exec circleci-nomad-0 -n <namespace> – nomad node status -allocs` command and viewing the following output:

```
ID        DC       Name             Class         Drain  Eligibility  Status  Running Allocs
21dc1d47  default  ip-10-0-10-134   linux-64bit   false  eligible     ready   0
```

If you do not see any Nomad clients listed, please consult our <nomad#,Introduction to Nomad Cluster Operation> for more detailed information on managing and troubleshooting the nomad server.

DC in the output stands for `datacenter` and will always be `default`, and should be left as such. It doesn't affect or break anything. The things that are the most important are Drain, Status, and Allocs columns.

- **Drain** - If `Drain` is `true` then CircleCI will **not** route jobs to that nomad client. It is possible to change this value by running the following command:

```
$ kubectl exec circleci-nomad-0 -n circleci-eks -- nomad node drain -enable -yes <Nomad Node ID>
```

If you set Drain to `true`, it will finish the jobs that were currently running and then stop accepting builds. After the number of allocations reaches 0, it is safe to terminate instance. If `Drain` is set to `false` it means the node is accepting connections and should be getting builds.

- **Status** - If Status is `ready` then it is ready to accept builds and should be wired up correctly. If it is not wired up correctly it will not show `ready` and it should be investigated because a node that is not showing `ready` in the Status will not accept builds.
- **Allocs** - Allocs is a term used to refer to builds. Hence, the number of Running Allocs is the number of

builds running on the corresponding single node. This number indicates whether builds are routed. If all of the nodes have Running Allocs, but your job is still queued, that means you do not have enough capacity and you need to add more Nomad nodes to your fleet.

If the outputs look good, but your builds are still queued, then continue to the next step.

5. Check Job Processing Status

Run `kubectl exec circleci-nomad-0 -n <namespace> – nomad status` command to view the jobs that are currently being processed. It should list the status of each job as well as the ID of jobs, as follows:

```
$ kubectl exec circleci-nomad-0 -n <namespace> -- nomad status
ID                                                         Type   Priority  Status   Submit Date
6007ca6352718c10b1227a57-0-build-circleci-realitycheck-21  batch  50        running  2021-01-
20T06:15:00Z
```

Once a job has completed, the Status changes to `dead`. If the status is `running`, the job is currently running. Those running jobs should appear as running in the CircleCI app builds dashboard. If it is not appearing in the UI, there may be a problem with output-processor. Run `kubectl logs -l app=output-processor --tail=10000 -n <name_space>` and check the logs for any obvious stack traces.

- If the job is in `pending` state constantly with no allocations being made, run `kubectl exec circleci-nomad-0 -n <namespace> – nomad status JOB_ID` command to see where Nomad is stuck and then refer to standard Nomad Cluster error documentation for information.
- If the job is running/dead but the CircleCI app shows nothing:
  - Check Nomad job logs by running `kubectl exec circleci-nomad-0 -n <namespace> – nomad logs -stderr -job JOB_ID` command.
  - Run `kubectl logs <output-processor_pod_name> -n <name_space>` command to see whether output-processor gives more clues about the errors on the Nomad side.

Why do my jobs stay in `queued` status and never successfully run?

This applies especially if `nomad logs` command shows messages similar to the one below.

```
{"error":"rpc error: code = Unavailable desc = grpc: the connection is
unavailable","level":"warning","msg":"error fetching config, retrying","time":"2018-04-
17T18:47:01Z"}
```

Check port `8585` by the following command on a Nomad client. If the output is empty, the nomad client connects to the output-processor. If the output shows numbers, please check this number on the exit code section in `man curl`.

```
$ curl -s output.<BASENAME>.circleci.internal:8585 -o /dev/null || echo $?
```

Why is the cache failing to unpack?

If a `restore_cache` step is failing for one of your jobs, it is worth checking the size of the cache - you can view the cache size from the CircleCI Jobs page within the `restore_cache` step. We recommend keeping cache sizes under 500MB – this is our upper limit for corruption checks because above this limit check times would be excessively long. Larger cache sizes are allowed but may cause problems due to a higher chance of decompression issues and corruption during download. To keep cache sizes down, consider splitting into multiple distinct caches.

CircleCI Server Services Architecture

This document outlines the services within a CircleCI Server v3.0-beta cluster. This is provided both to give an overview of service operation, and to help with troubleshooting in the event of service outages. Supplementary notes and a key are provided below the following table.

Notes

- Database migrator services are listed here with a low failure severity as they only run at startup, however:

  ⚠ If migrator services are down at startup connected services will fail.

- Some services can be externalized (marked with * here) and managed to suit your requirements. External locations for these services can be set from the admin console either during installation or after.

key

Icon	Description
✓	Failure has a minor affect on production - no loss of data or functioning.
⚠	Failure might cause issues with some jobs, but no loss of data.
🧯	Failure can cause loss of data, corruption of jobs/workflows, major loss of functionality.

Services, Roles, Failure Modes

Pod	Role	What happens if it fails?	Failure severity
`api-service`	Provides a GraphQL API that provides much of the data to render the web frontend.	Many parts of the UI (e.g. Contexts) will fail completely.	🧯
`audit-log-service`	Persists audit log events to blob storage for long term storage.	Some events may not be recorded.	✓
`circleci-mongodb-0`	Mongo data store.	Potential total data loss. All running builds will fail and the UI will not work.	🧯
`circleci-nomad-0`	Nomad primary service.	No jobs will run.	🧯
`circleci-postgresql`	Basic `postgresql` with enhancements for creating required databases when containers are launched.	Potential total data loss. All running builds will fail and the UI will not work.	🧯
`circleci-rabbitmq-0`	Runs the RabbitMQ server. Most of our services use RabbitMQ for queueing.	Potential total data loss. All running builds will fail and the UI will not work.	🧯
`circleci-redis-master-0` and `circleci-redis-slave`	The Redis key/value store.	Lose output from currently-running job steps. API calls out to GitHub may also fail.	⚠
`circleci-telegraf`	This is the statsd forwarding agent that our local services write to and can be configured to forward to an external metrics service.	Metics will stop working but jobs will continue to run.	✓
`circleci-vault-0`	Instance of Hashicorp's Vault – an encryption service that provides key-management, secure storage, and other encryption related services. Used to handle the encryption and key store for the `contexts-service`.	`contexts-service` will stop working, and all jobs that use `contexts-service` will fail.	⚠

Pod	Role	What happens if it fails?	Failure severity
`contexts-service`	Stores and provides encrypted contexts.	All builds using Contexts will fail.	⚠
`cron-service`	Triggers scheduled workflows.	Scheduled workflows will not run.	⚠
`dispatcher`	Splits a job into tasks and sends them to `schedulerer` to be run.	No jobs will be sent to Nomad, the run queue will increase in size but there should be no meaningful loss of data.	⚠
`domain-service`	Stores and provides information about our domain model.	Workflows will fail to start and some REST API calls may fail causing `500` errors in the CircleCI UI. If LDAP authentication is in use, all logins will fail.	🧯
`exim`	Mail Transfer Agent (MTA) used to send all outbound mail SMTP.	No email notifications will be sent.	✓
`external-dns`	Responsible for setting up (internal) DNS records for installation, used for communication between Nomad clients and the cluster, as well as the public DNS record for the installation itself (i.e., the frontend).	Nomad will be unable to communicate with the front-end.	🧯
`federations-service`	Stores user identities (LDAP).	If LDAP authentication is in use, all logins will fail and some REST API calls might fail.	🧯 only if LDAP in use
`frontend`	CircleCI web app and www-api proxy.	The UI and REST API will be unavailable and no jobs will be triggered by GitHub/Enterprise. Running builds will be OK but no updates will be seen.	⚠

Pod	Role	What happens if it fails?	Failure severity
`grafana`	Part of the local observability stack. Provides a frontend service to view dashboards, metrics, and logs for the installation.	If metrics are forwarded to another service, this service is unaffected.	✓
`legacy-dispatcher`	Critical service needed to run jobs. Responsible for part of the dispatching process to Nomad.	User will not be able to run jobs.	🧯
`legacy-notifier`	Ensures changes to a job's status reach the front-end.	Jobs will still run, but what is reflected in the UI may not reflect what is actually occurring.	🧯
`loki-0`	Log aggregator which collects logs from all running services for consumption by grafana.	If metrics are forwarded to another service, this service is unaffected.	✓
`nomad-gc`	Nomad garbage collector. Ensures Nomad resources are freed and cleaned up when they are no longer needed.	Nomad resources may not be cleaned up until this pod is restarted.	✓
`nomad-metrics`	Collects and forwards metrics to Nomad clients.	Metrics may not be sent to Nomad until this pod is restarted.	✓
`opencensus-agent`	Collector of traces used for debugging only.	Failure is uncritical.	✓
`output-processor`	Receives job output & status updates and writes them to MongoDB. Also provides an API to running jobs to access caches, workspaces, store caches, workspaces, artifacts, & test results.	All running builds will either fail or be left in an unfixable, inconsistent state. There will also be data loss in terms of step output, test results and artifacts.	🧯
`permissions-service`	Provides the CircleCI permissions interface.	Workflows will fail to start and some REST API calls may fail, causing 500 errors in the UI.	⚠
`prometheus`	Collects Prometheus metrics used for event monitoring.	Metrics may not be collected until this pod is restarted.	✓

Pod	Role	What happens if it fails?	Failure severity
`scheduler`	Sends tasks to `server-nomad` to run. \	No jobs will be sent to Nomad, the run queue will increase in size but there should be no meaningful loss of data.	⚠
`server-troubleshooter`	Responsible for creating support bundles.	Support bundles may not be created until this pod is restarted.	✓
`slanger`	Provides real-time events to the CircleCI app.	Live UI updates will stop but hard refreshes will still work.	✓
`test-results`	Parses test result files and stores data.	There will be no test failure or timing data for jobs, but this will be back-filled once the service is restarted.	✓
`vm-gc`	VM Garbage Collector — periodically check for stale `machine` and remote Docker instances and request that `vm-service` remove them.	Old vm-service instances might not be destroyed until this service is restarted.	✓
`vm-scaler`	Periodically requests that `vm-service` provision more instances for running `machine` and remote Docker jobs.	VM instances for `machine` and Remote Docker might not be provisioned causing you to run out of capacity to run jobs with these executors.	⚠
`vm-service`	Inventory of available `vm-service` instances, and provisioning of new instances.	Jobs that use `machine` or remote Docker will fail.	⚠
`workflows-conductor`	Coordinates and provides information about workflows.	No new workflows will start, currently running workflows might end up in an inconsistent state, and some REST and GraphQL API requests will fail.	🧯