Monitoring and troubleshooting self-hosted runners

In this article

You can monitor your self-hosted runners to view their activity and diagnose common issues.

Mac Windows Linux

Using repository-level self-hosted runners

You may not be able to create a self-hosted runner for an organization-owned repository.

Enterprise owners and organization owners can disable the ability to create self-hosted runners at the repository level. For more information, see "Enforcing policies for GitHub Actions in your enterprise" and "Disabling or limiting GitHub Actions for your organization."

Checking the status of a self-hosted runner

A self-hosted runner can be located in either your repository, organization, or enterprise account settings on GitHub. To manage a self-hosted runner, you must have the following permissions, depending on where the self-hosted runner was added:

- **User repository**: You must be the repository owner.
- **Organization**: You must be an organization owner.
- **Organization repository**: You must be an organization owner, or have admin access to the repository.
- **Enterprise account**: You must be an enterprise owner.

1. In your organization or repository, navigate to the main page and click **Settings**.
2. In the left sidebar, click **Actions**, then click **Runners**.

3. Under "Runners", you can view a list of registered runners, including the runner's name, labels, and status.

   The status can be one of the following:

   - **Idle**: The runner is connected to GitHub Enterprise Cloud and is ready to execute jobs.
   - **Active**: The runner is currently executing a job.
   - **Offline**: The runner is not connected to GitHub Enterprise Cloud. This could be because the machine is offline, the self-hosted runner application is not running on the machine, or the self-hosted runner application cannot communicate with GitHub Enterprise Cloud.

Troubleshooting network connectivity

Checking self-hosted runner network connectivity

You can use the self-hosted runner application's `run` script with the `--check` parameter to check that a self-hosted runner can access all required network services on GitHub.com.

In addition to `--check`, you must provide two arguments to the script:

- `--url` with the URL to your GitHub repository, organization, or enterprise. For example, `--url https://github.com/octo-org/octo-repo`.
- `--pat` with the value of a personal access token (classic), which must have the `workflow` scope, or a fine-grained personal access token with workflows read and write access . For example, `--pat ghp_abcd1234`. For more information, see "Managing your personal access tokens."

For example:

```
./run.sh --check --url URL --pat ghp_abcd1234
```


```
./run.sh --check --url URL --pat ghp_abcd1234
```


```
run.cmd --check --url https://github.com/YOUR-ORG/YOUR-REPO --pat GHP_ABCD1234
```

The script tests each service, and outputs either a `PASS` or `FAIL` for each one. If you have any failing checks, you can see more details on the problem in the log file for the check. The log files are located in the `_diag` directory where you installed the runner application, and the path of the log file for each check is shown in the console output of the script.

If you have any failing checks, you should also verify that your self-hosted runner machine meets all the communication requirements. For more information, see "About self-hosted runners."

Disabling TLS certificate verification

By default, the self-hosted runner application verifies the TLS certificate for GitHub Enterprise Cloud. If you encounter network problems, you may wish to disable TLS certificate verification for testing purposes.

To disable TLS certification verification in the self-hosted runner application, set the

`GITHUB_ACTIONS_RUNNER_TLS_NO_VERIFY` environment variable to `1` before configuring and running the self-hosted runner application.

```
export GITHUB_ACTIONS_RUNNER_TLS_NO_VERIFY=1
./config.sh --url https://github.com/YOUR-ORG/YOUR-REPO --token
./run.sh
```


```
export GITHUB_ACTIONS_RUNNER_TLS_NO_VERIFY=1
./config.sh --url https://github.com/YOUR-ORG/YOUR-REPO --token
./run.sh
```


```
[Environment]::SetEnvironmentVariable('GITHUB_ACTIONS_RUNNER_TLS_NO_VERIFY', '1')
./config.cmd --url https://github.com/YOUR-ORG/YOUR-REPO --token
./run.sh
```

Warning: Disabling TLS verification is not recommended since TLS provides privacy and data integrity between the self-hosted runner application and GitHub Enterprise Cloud. We recommend that you install the GitHub Enterprise Cloud certificate in the operating system certificate store for your self-hosted runner. For guidance on how to install the GitHub Enterprise Cloud certificate, check with your operating system vendor.

Reviewing the self-hosted runner application log files

You can monitor the status of the self-hosted runner application and its activities. Log files are kept in the `_diag` directory where you installed the runner application, and a new log is generated each time the application is started. The filename begins with _Runner__, and is followed by a UTC timestamp of when the application was started.

For detailed logs on workflow job executions, see the next section describing the _Worker__ files.

Reviewing a job's log file

The self-hosted runner application creates a detailed log file for each job that it processes. These files are stored in the `_diag` directory where you installed the runner application, and the filename begins with _Worker__.

Using journalctl to check the self-hosted runner application service

For Linux-based self-hosted runners running the application using a service, you can use `journalctl` to monitor their real-time activity. The default systemd-based service uses the following naming convention: `actions.runner.<org>-<repo>.<runnerName>.service`. This name is truncated if it exceeds 80 characters, so the preferred way of finding the service's name is by checking the *.service* file. For example:

```
$ cat ~/actions-runner/.service
actions.runner.octo-org-octo-repo.runner01.service
```

If this fails due to the service being installed elsewhere, you can find the service name in the list of running services. For example, on most Linux systems you can use the `systemctl` command:

```
$ systemctl --type=service | grep actions.runner
actions.runner.octo-org-octo-repo.hostname.service loaded active running GitHub
Actions Runner (octo-org-octo-repo.hostname)
```

You can use `journalctl` to monitor the real-time activity of the self-hosted runner:

```
sudo journalctl -u actions.runner.octo-org-octo-repo.runner01.service -f
```

In this example output, you can see `runner01` start, receive a job named `testAction`, and then display the resulting status:

```
Feb 11 14:57:07 runner01 runsvc.sh[962]: Starting Runner listener with startup
type: service
Feb 11 14:57:07 runner01 runsvc.sh[962]: Started listener process
Feb 11 14:57:07 runner01 runsvc.sh[962]: Started running service
Feb 11 14:57:16 runner01 runsvc.sh[962]: √ Connected to GitHub
Feb 11 14:57:17 runner01 runsvc.sh[962]: 2020-02-11 14:57:17Z: Listening for Jobs
Feb 11 16:06:54 runner01 runsvc.sh[962]: 2020-02-11 16:06:54Z: Running job:
testAction
Feb 11 16:07:10 runner01 runsvc.sh[962]: 2020-02-11 16:07:10Z: Job testAction
completed with result: Succeeded
```

To view the `systemd` configuration, you can locate the service file here: `/etc/systemd/system/actions.runner.<org>-<repo>.<runnerName>.service`. If you want to customize the self-hosted runner application service, do not directly modify this file. Follow the instructions described in "Configuring the self-hosted runner application as a service."

Using `launchd` to check the self-hosted runner application service

For macOS-based self-hosted runners running the application as a service, you can use `launchctl` to monitor their real-time activity. The default launchd-based service uses the following naming convention: `actions.runner.<org>-<repo>.<runnerName>`. This name is truncated if it exceeds 80 characters, so the preferred way of finding the service's name is by checking the *.service* file in the runner directory:

```
% cat ~/actions-runner/.service
/Users/exampleUsername/Library/LaunchAgents/actions.runner.octo-org-octo-
repo.runner01.plist
```

The `svc.sh` script uses `launchctl` to check whether the application is running. For example:

```
$ ./svc.sh status
status actions.runner.example.runner01:
/Users/exampleUsername/Library/LaunchAgents/actions.runner.example.runner01.plist
Started:
379 0 actions.runner.example.runner01
```

The resulting output includes the process ID and the name of the application's `launchd` service.

To view the `launchd` configuration, you can locate the service file here: `/Users/exampleUsername/Library/LaunchAgents/actions.runner.<repoName>.<runnerName>.service`. If you want to customize the self-hosted runner application service, do not directly modify this file. Follow the instructions described in "Configuring the self-hosted runner application as a service."

Using PowerShell to check the self-hosted runner application service

For Windows-based self-hosted runners running the application as a service, you can use PowerShell to monitor their real-time activity. The service uses the naming convention `GitHub Actions Runner (<org>-<repo>.<runnerName>)`. You can also find the service's name by checking the *.service* file in the runner directory:

```
PS C:\actions-runner> Get-Content .service
actions.runner.octo-org-octo-repo.runner01.service
```

You can view the status of the runner in the Windows *Services* application (`services.msc`). You can also use PowerShell to check whether the service is running:

```
PS C:\actions-runner> Get-Service "actions.runner.octo-org-octo-
repo.runner01.service" | Select-Object Name, Status
Name                                                Status
----                                                ------
actions.runner.octo-org-octo-repo.runner01.service    Running
```

You can use PowerShell to check the recent activity of the self-hosted runner. In this example output, you can see the application start, receive a job named `testAction`, and then display the resulting status:

```
PS C:\actions-runner> Get-EventLog -LogName Application -Source
ActionsRunnerService

   Index Time          EntryType   Source                 InstanceID Message
   ----- ----          ---------   ------                 ---------- -------
     136 Mar 17 13:45  Information ActionsRunnerService          100 2020-03-17
13:45:48Z: Job Greeting completed with result: Succeeded
     135 Mar 17 13:45  Information ActionsRunnerService          100 2020-03-17
13:45:34Z: Running job: testAction
     134 Mar 17 13:41  Information ActionsRunnerService          100 2020-03-17
13:41:54Z: Listening for Jobs
     133 Mar 17 13:41  Information ActionsRunnerService          100 û Connected
to GitHub
     132 Mar 17 13:41  Information ActionsRunnerService            0 Service
started successfully.
     131 Mar 17 13:41  Information ActionsRunnerService          100 Starting
Actions Runner listener
     130 Mar 17 13:41  Information ActionsRunnerService          100 Starting
Actions Runner Service
     129 Mar 17 13:41  Information ActionsRunnerService          100 create event
log trace source for actions-runner service
```

Monitoring the automatic update process

We recommend that you regularly check the automatic update process, as the self-hosted runner will not be able to process jobs if it falls below a certain version threshold. The self-hosted runner application automatically updates itself, but note that this process does not include any updates to the operating system or other software; you will need to separately manage these updates.

You can view the update activities in the _Runner__ log files. For example:

```
[Feb 12 12:37:07 INFO SelfUpdater] An update is available.
```

In addition, you can find more information in the *SelfUpdate* log files located in the

`_diag` directory where you installed the runner application.

Troubleshooting containers in self-hosted runners

Checking that Docker is installed

If your jobs require containers, then the self-hosted runner must be Linux-based and needs to have Docker installed. Check that your self-hosted runner has Docker installed and that the service is running.

You can use `systemctl` to check the service status:

```
$ sudo systemctl is-active docker.service
active
```

If Docker is not installed, then dependent actions will fail with the following errors:

```
[2020-02-13 16:56:10Z INFO DockerCommandManager] Which: 'docker'
[2020-02-13 16:56:10Z INFO DockerCommandManager] Not found.
[2020-02-13 16:56:10Z ERR  StepsRunner] Caught exception from step:
System.IO.FileNotFoundException: File not found: 'docker'
```

Checking the Docker permissions

If your job fails with the following error:

```
dial unix /var/run/docker.sock: connect: permission denied
```

Check that the self-hosted runner's service account has permission to use the Docker service. You can identify this account by checking the configuration of the self-hosted runner in `systemd`. For example:

```
$ sudo systemctl show -p User actions.runner.octo-org-octo-repo.runner01.service
User=runner-user
```

Checking which Docker engine is installed on the runner

If your build fails with the following error:

```
Error: Input required and not supplied: java-version
```

Check which Docker engine is installed on your self-hosted runner. To pass the inputs of an action into the Docker container, the runner uses environment variables that might contain dashes as part of their names. The action may not able to get the inputs if the Docker engine is not a binary executable, but is instead a shell wrapper or a link (for example, a Docker engine installed on Linux using `snap`). To address this error, configure your self-hosted runner to use a different Docker engine.

To check if your Docker engine was installed using `snap`, use the `which` command. In the following example, the Docker engine was installed using `snap`:

```
$ which docker
/snap/bin/docker
```