Migrating from GitLab with GitHub Actions Importer

In this article

Learn how to use GitHub Actions Importer to automate the migration of your GitLab pipelines to GitHub Actions.

Legal notice

About migrating from GitLab with GitHub Actions Importer

The instructions below will guide you through configuring your environment to use GitHub Actions Importer to migrate GitLab pipelines to GitHub Actions.

Prerequisites

- A GitLab account or organization with pipelines and jobs that you want to convert to GitHub Actions workflows.
- Access to create a GitLab personal access token for your account or organization.
- An environment where you can run Linux-based containers, and can install the necessary tools.
  - Docker is installed and running.
  - GitHub CLI is installed.

> **Note**: The GitHub Actions Importer container and CLI do not need to be installed on the same server as your CI platform.

Limitations

There are some limitations on migrating processes automatically from GitLab pipelines to GitHub Actions with GitHub Actions Importer.

- Automatic caching in between jobs of different workflows is not supported.

- The `audit` command is only supported when using an organization account. However, the `dry-run` and `migrate` commands can be used with an organization or user account.

Manual tasks

Certain GitLab constructs must be migrated manually. These include:

- Masked project or group variable values
- Artifact reports

For more information on manual migrations, see "Migrating from GitLab CI/CD to GitHub Actions."

Installing the GitHub Actions Importer CLI extension

1. Install the GitHub Actions Importer CLI extension:

   ```
   Bash
   ```

   ```
   gh extension install github/gh-actions-importer
   ```

2. Verify that the extension is installed:

   ```
   $ gh actions-importer -h
   Options:
     -?, -h, --help  Show help and usage information

   Commands:
     update     Update to the latest version of GitHub Actions Importer.
     version    Display the version of GitHub Actions Importer.
     configure  Start an interactive prompt to configure credentials used to
   authenticate with your CI server(s).
     audit      Plan your CI/CD migration by analyzing your current CI/CD
   footprint.
     forecast   Forecast GitHub Actions usage from historical pipeline
   utilization.
     dry-run    Convert a pipeline to a GitHub Actions workflow and output its
   yaml file.
     migrate    Convert a pipeline to a GitHub Actions workflow and open a pull
   request with the changes.
   ```

Configuring credentials

The `configure` CLI command is used to set required credentials and options for GitHub Actions Importer when working with GitLab and GitHub.

1. Create a GitHub personal access token (classic). For more information, see "Managing your personal access tokens."

   Your token must have the `workflow` scope.

   After creating the token, copy it and save it in a safe location for later use.

2. Create a GitLab personal access token. For more information, see Personal access tokens in the GitLab documentation.

Your token must have the `read_api` scope.

After creating the token, copy it and save it in a safe location for later use.

3. In your terminal, run the GitHub Actions Importer `configure` CLI command:

```
gh actions-importer configure
```

The `configure` command will prompt you for the following information:

- For "Which CI providers are you configuring?", use the arrow keys to select `GitLab`, press `Space` to select it, then press `Enter`.
- For "Personal access token for GitHub", enter the value of the personal access token (classic) that you created earlier, and press `Enter`.
- For "Base url of the GitHub instance", press `Enter` to accept the default value (`https://github.com`).
- For "Private token for GitLab", enter the value for the GitLab personal access token that you created earlier, and press `Enter`.
- For "Base url of the GitLab instance", enter the URL of your GitLab instance, and press `Enter`.

An example of the output of the `configure` command is shown below.

```
$ gh actions-importer configure
✔ Which CI providers are you configuring?: GitLab
Enter the following values (leave empty to omit):
✔ Personal access token for GitHub: ***************
✔ Base url of the GitHub instance: https://github.com
✔ Private token for GitLab: ***************
✔ Base url of the GitLab instance: http://localhost
Environment variables successfully updated.
```

4. In your terminal, run the GitHub Actions Importer `update` CLI command to connect to GitHub Packages Container registry and ensure that the container image is updated to the latest version:

```
gh actions-importer update
```

The output of the command should be similar to below:

```
Updating ghcr.io/actions-importer/cli:latest...
ghcr.io/actions-importer/cli:latest up-to-date
```

Perform an audit of GitLab

You can use the `audit` command to get a high-level view of all pipelines in a GitLab server.

The `audit` command performs the following steps:

1. Fetches all of the projects defined in a GitLab server.
2. Converts each pipeline to its equivalent GitHub Actions workflow.
3. Generates a report that summarizes how complete and complex of a migration is possible with GitHub Actions Importer.

Prerequisites for the audit command

In order to use the `audit` command, you must have a personal access token configured with a GitLab organization account.

Running the audit command

To perform an audit of a GitLab server, run the following command in your terminal, replacing `my-gitlab-namespace` with the namespace or group you are auditing:

```
gh actions-importer audit gitlab --output-dir tmp/audit --namespace my-gitlab-namespace
```

Inspecting the audit results

The files in the specified output directory contain the results of the audit. See the `audit_summary.md` file for a summary of the audit results.

The audit summary has the following sections.

Pipelines

The "Pipelines" section contains a high-level statistics regarding the conversion rate done by GitHub Actions Importer.

Listed below are some key terms that can appear in the "Pipelines" section:

- **Successful** pipelines had 100% of the pipeline constructs and individual items converted automatically to their GitHub Actions equivalent.
- **Partially successful** pipelines had all of the pipeline constructs converted, however, there were some individual items that were not converted automatically to their GitHub Actions equivalent.
- **Unsupported** pipelines are definition types that are not supported by GitHub Actions Importer.
- **Failed** pipelines encountered a fatal error when being converted. This can occur for one of three reasons:
  - The pipeline was misconfigured and not valid in Bamboo.
  - GitHub Actions Importer encountered an internal error when converting it.
  - There was an unsuccessful network response that caused the pipeline to be inaccessible, which is often due to invalid credentials.

Build steps

The "Build steps" section contains an overview of individual build steps that are used across all pipelines, and how many were automatically converted by GitHub Actions Importer.

Listed below are some key terms that can appear in the "Build steps" section:

- A **known** build step is a step that was automatically converted to an equivalent action.
- An **unknown** build step is a step that was not automatically converted to an equivalent action.
- An **unsupported** build step is a step that is either:
  - Fundamentally not supported by GitHub Actions.
  - Configured in a way that is incompatible with GitHub Actions.

- An **action** is a list of the actions that were used in the converted workflows. This can be important for:
  - If you use GitHub Enterprise Server, gathering the list of actions to sync to your instance.
  - Defining an organization-level allowlist of actions that are used. This list of actions is a comprehensive list of actions that your security or compliance teams may need to review.

Manual tasks

The "Manual tasks" section contains an overview of tasks that GitHub Actions Importer is not able to complete automatically, and that you must complete manually.

Listed below are some key terms that can appear in the "Manual tasks" section:

- A **secret** is a repository or organization-level secret that is used in the converted pipelines. These secrets must be created manually in GitHub Actions for these pipelines to function properly. For more information, see "Using secrets in GitHub Actions."
- A **self-hosted runner** refers to a label of a runner that is referenced in a converted pipeline that is not a GitHub-hosted runner. You will need to manually define these runners for these pipelines to function properly.

Files

The final section of the audit report provides a manifest of all the files that were written to disk during the audit.

Each pipeline file has a variety of files included in the audit, including:

- The original pipeline as it was defined in GitHub.
- Any network responses used to convert the pipeline.
- The converted workflow file.
- Stack traces that can be used to troubleshoot a failed pipeline conversion.

Additionally, the `workflow_usage.csv` file contains a comma-separated list of all actions, secrets, and runners that are used by each successfully converted pipeline. This can be useful for determining which workflows use which actions, secrets, or runners, and can be useful for performing security reviews.

Forecast potential build runner usage

You can use the `forecast` command to forecast potential GitHub Actions usage by computing metrics from completed pipeline runs in your GitLab server.

Running the forecast command

To perform a forecast of potential GitHub Actions usage, run the following command in your terminal, replacing `my-gitlab-namespace` with the namespace or group you are forecasting. By default, GitHub Actions Importer includes the previous seven days in the forecast report.

```
gh actions-importer forecast gitlab --output-dir tmp/forecast --namespace my-
gitlab-namespace
```

Forecasting an entire namespace

To forecast an entire namespace and all of its subgroups, you must specify each subgroup in the `--namespace` argument or `NAMESPACE` environment variable.

For example:

```
gh actions-importer forecast gitlab --namespace my-gitlab-namespace my-gitlab-
namespace/subgroup-one my-gitlab-namespace/subgroup-two ...
```

Inspecting the forecast report

The `forecast_report.md` file in the specified output directory contains the results of the forecast.

Listed below are some key terms that can appear in the forecast report:

- The **job count** is the total number of completed jobs.
- The **pipeline count** is the number of unique pipelines used.
- **Execution time** describes the amount of time a runner spent on a job. This metric can be used to help plan for the cost of GitHub-hosted runners.
    - This metric is correlated to how much you should expect to spend in GitHub Actions. This will vary depending on the hardware used for these minutes. You can use the GitHub Actions pricing calculator to estimate the costs.
- **Queue time** metrics describe the amount of time a job spent waiting for a runner to be available to execute it.
- **Concurrent jobs** metrics describe the amount of jobs running at any given time. This metric can be used to define the number of runners you should configure.

Additionally, these metrics are defined for each queue of runners in GitLab. This is especially useful if there is a mix of hosted or self-hosted runners, or high or low spec machines, so you can see metrics specific to different types of runners.

Perform a dry-run migration of a GitLab pipeline

You can use the `dry-run` command to convert a GitLab pipeline to its equivalent GitHub Actions workflow.

Running the dry-run command

You can use the `dry-run` command to convert a GitLab pipeline to an equivalent GitHub Actions workflow. A dry-run creates the output files in a specified directory, but does not open a pull request to migrate the pipeline.

To perform a dry run of migrating your GitLab pipelines to GitHub Actions, run the following command in your terminal, replacing `my-gitlab-project` with the URL of your GitLab project, and `my-gitlab-namespace` with the namespace or group you are performing a dry run for.

```
gh actions-importer dry-run gitlab --output-dir tmp/dry-run --namespace my-
gitlab-namespace --project my-gitlab-project
```

Inspecting the converted workflows

You can view the logs of the dry run and the converted workflow files in the specified output directory.

If there is anything that GitHub Actions Importer was not able to convert automatically,

such as unknown build steps or a partially successful pipeline, you might want to create custom transformers to further customize the conversion process. For more information, see "Extending GitHub Actions Importer with custom transformers."

Perform a production migration of a GitLab pipeline

You can use the `migrate` command to convert a GitLab pipeline and open a pull request with the equivalent GitHub Actions workflow.

Running the migrate command

To migrate a GitLab pipeline to GitHub Actions, run the following command in your terminal, replacing the following values:

- `target-url` value with the URL for your GitHub Enterprise Cloud repository
- `my-gitlab-project` with the URL for your GitLab project
- `my-gitlab-namespace` with the namespace or group you are migrating

```
gh actions-importer migrate gitlab --target-url https://github.com/:owner/:repo -
-output-dir tmp/migrate --namespace my-gitlab-namespace --project my-gitlab-
project
```

The command's output includes the URL to the pull request that adds the converted workflow to your repository. An example of a successful output is similar to the following:

```
$ gh actions-importer migrate gitlab --target-url https://github.com/octo-
org/octo-repo --output-dir tmp/migrate --namespace octo-org --project monas-
project
[2022-08-20 22:08:20] Logs: 'tmp/migrate/log/actions-importer-20220916-
014033.log'
[2022-08-20 22:08:20] Pull request: 'https://github.com/octo-org/octo-
repo/pull/1'
```

Inspecting the pull request

The output from a successful run of the `migrate` command contains a link to the new pull request that adds the converted workflow to your repository.

Some important elements of the pull request include:

- In the pull request description, a section called **Manual steps**, which lists steps that you must manually complete before you can finish migrating your pipelines to GitHub Actions. For example, this section might tell you to create any secrets used in your workflows.
- The converted workflows file. Select the **Files changed** tab in the pull request to view the workflow file that will be added to your GitHub Enterprise Cloud repository.

When you are finished inspecting the pull request, you can merge it to add the workflow to your GitHub Enterprise Cloud repository.

Reference

This section contains reference information on environment variables, optional arguments, and supported syntax when using GitHub Actions Importer to migrate from GitLab.

Using environment variables

GitHub Actions Importer uses environment variables for its authentication configuration. These variables are set when following the configuration process using the `configure` command. For more information, see the "Configure credentials for GitHub Actions Importer" section.

GitHub Actions Importer uses the following environment variables to connect to your GitLab instance:

- `GITHUB_ACCESS_TOKEN`: The personal access token (classic) used to create pull requests with a converted workflow (requires the `workflow` scope).
- `GITHUB_INSTANCE_URL`: The URL to the target GitHub instance (for example, `https://github.com`).
- `GITLAB_ACCESS_TOKEN`: The GitLab personal access token used to view GitLab resources.
- `GITLAB_INSTANCE_URL`: The URL of the GitLab instance.
- `NAMESPACE`: The namespaces or groups that contain the GitLab pipelines.

These environment variables can be specified in a `.env.local` file that is loaded by GitHub Actions Importer when it is run.

Using optional arguments

There are optional arguments you can use with the GitHub Actions Importer subcommands to customize your migration.

`--source-file-path`

You can use the `--source-file-path` argument with the `forecast`, `dry-run`, or `migrate` subcommands.

By default, GitHub Actions Importer fetches pipeline contents from source control. The `--source-file-path` argument tells GitHub Actions Importer to use the specified source file path instead.

For example:

```
gh actions-importer dry-run gitlab --output-dir output/ --namespace my-gitlab-
namespace --project my-gitlab-project --source-file-path path/to/.gitlab-ci.yml
```

If you would like to supply multiple source files when running the `forecast` subcommand, you can use pattern matching in the file path value. The following example supplies GitHub Actions Importer with any source files that match the `./tmp/previous_forecast/jobs/*.json` file path.

```
gh actions-importer forecast gitlab --output-dir output/ --namespace my-gitlab-
namespace --project my-gitlab-project --source-file-path
./tmp/previous_forecast/jobs/*.json
```

`--config-file-path`

You can use the `--config-file-path` argument with the `audit`, `dry-run`, and `migrate` subcommands.

By default, GitHub Actions Importer fetches pipeline contents from source control. The `--config-file-path` argument tells GitHub Actions Importer to use the specified source files instead.

The `--config-file-path` argument can also be used to specify which repository a converted reusable workflow should be migrated to.

Audit example

In this example, GitHub Actions Importer uses the specified YAML configuration file to perform an audit.

```
gh actions-importer audit gitlab --output-dir path/to/output/ --namespace my-
gitlab-namespace --config-file-path path/to/gitlab/config.yml
```

To audit a GitLab instance using a configuration file, the file must be in the following format, and each `repository_slug` value must be unique:

```yaml
source_files:
  - repository_slug: namespace/project-name
    path: path/to/.gitlab-ci.yml
  - repository_slug: namespace/some-other-project-name
    path: path/to/.gitlab-ci.yml
```

Dry run example

In this example, GitHub Actions Importer uses the specified YAML configuration file as the source file to perform a dry run.

The pipeline is selected by matching the `repository_slug` in the configuration file to the value of the `--namespace` and `--project` options. The `path` is then used to pull the specified source file.

```
gh actions-importer dry-run gitlab --namespace my-gitlab-namespace --project my-
gitlab-project-name --output-dir ./output/ --config-file-path
./path/to/gitlab/config.yml
```

Specify the repository of converted reusable workflows

GitHub Actions Importer uses the YAML file provided to the `--config-file-path` argument to determine the repository that converted reusable workflows are migrated to.

To begin, you should run an audit without the `--config-file-path` argument:

```
gh actions-importer audit gitlab --output-dir ./output/
```

The output of this command will contain a file named `config.yml` that contains a list of all the composite actions that were converted by GitHub Actions Importer. For example, the `config.yml` file may have the following contents:

```yaml
reusable_workflows:
  - name: my-reusable-workflow.yml
    target_url: https://github.com/octo-org/octo-repo
    ref: main
```

You can use this file to specify which repository and ref a reusable workflow or composite action should be added to. You can then use the `--config-file-path` argument to provide the `config.yml` file to GitHub Actions Importer. For example, you can use this file when running a `migrate` command to open a pull request for each unique repository defined in the config file:

```
gh actions-importer migrate gitlab --project my-project-name --output-dir output/
--config-file-path config.yml --target-url https://github.com/my-org/my-repo
```

Supported syntax for GitLab pipelines

The following table shows the type of properties GitHub Actions Importer is currently able to convert. For more details about how GitLab pipeline syntax aligns with GitHub Actions, see "Migrating from GitLab CI/CD to GitHub Actions".

GitLab Pipelines	GitHub Actions	Status
`after_script`	`jobs.<job_id>.steps`	Supported
`auto_cancel_pending_pipelines`	`concurrency`	Supported
`before_script`	`jobs.<job_id>.steps`	Supported
`build_timeout` or `timeout`	`jobs.<job_id>.timeout-minutes`	Supported
`default`	Not applicable	Supported
`image`	`jobs.<job_id>.container`	Supported
`job`	`jobs.<job_id>`	Supported
`needs`	`jobs.<job_id>.needs`	Supported
`only_allow_merge_if_pipeline_succeeds`	`on.pull_request`	Supported
`resource_group`	`jobs.<job_id>.concurrency`	Supported
`schedule`	`on.schedule`	Supported
`script`	`jobs.<job_id>.steps`	Supported
`stages`	`jobs`	Supported
`tags`	`jobs.<job_id>.runs-on`	Supported
`variables`	`env`, `jobs.<job_id>.env`	Supported
Run pipelines for new commits	`on.push`	Supported
Run pipelines manually	`on.workflow_dispatch`	Supported
`environment`	`jobs.<job_id>.environment`	Partially supported
`include`	Files referenced in an `include` statement are merged into a single job graph before being transformed.	Partially supported
`only` or `except`	`jobs.<job_id>.if`	Partially supported
`parallel`	`jobs.<job_id>.strategy`	Partially supported
`rules`	`jobs.<job_id>.if`	Partially supported

`services`	`jobs.<job_id>.services`	Partially supported
`workflow`	`if`	Partially supported

For information about supported GitLab constructs, see the `github/gh-actions-importer` repository.

Environment variables syntax

GitHub Actions Importer uses the mapping in the table below to convert default GitLab environment variables to the closest equivalent in GitHub Actions.

GitLab	GitHub Actions
`CI_API_V4_URL`	`${{ github.api_url }}`
`CI_BUILDS_DIR`	`${{ github.workspace }}`
`CI_COMMIT_BRANCH`	`${{ github.ref }}`
`CI_COMMIT_REF_NAME`	`${{ github.ref }}`
`CI_COMMIT_REF_SLUG`	`${{ github.ref }}`
`CI_COMMIT_SHA`	`${{ github.sha }}`
`CI_COMMIT_SHORT_SHA`	`${{ github.sha }}`
`CI_COMMIT_TAG`	`${{ github.ref }}`
`CI_JOB_ID`	`${{ github.job }}`
`CI_JOB_MANUAL`	`${{ github.event_name == 'workflow_dispatch' }}`
`CI_JOB_NAME`	`${{ github.job }}`
`CI_JOB_STATUS`	`${{ job.status }}`
`CI_JOB_URL`	`${{ github.server_url }}/${{ github.repository }}/actions/runs/${{ github.run_id }}`
`CI_JOB_TOKEN`	`${{ github.token }}`
`CI_NODE_INDEX`	`${{ strategy.job-index }}`
`CI_NODE_TOTAL`	`${{ strategy.job-total }}`
`CI_PIPELINE_ID`	`${{ github.repository}}/${{ github.workflow }}`
`CI_PIPELINE_IID`	`${{ github.workflow }}`
`CI_PIPELINE_SOURCE`	`${{ github.event_name }}`
`CI_PIPELINE_TRIGGERED`	`${{ github.actions }}`
`CI_PIPELINE_URL`	`${{ github.server_url }}/${{ github.repository }}/actions/runs/${{ github.run_id }}`
`CI_PROJECT_DIR`	`${{ github.workspace }}`

`CI_PROJECT_ID`	`${{ github.repository }}`
`CI_PROJECT_NAME`	`${{ github.event.repository.name }}`
`CI_PROJECT_NAMESPACE`	`${{ github.repository_owner }}`
`CI_PROJECT_PATH_SLUG`	`${{ github.repository }}`
`CI_PROJECT_PATH`	`${{ github.repository }}`
`CI_PROJECT_ROOT_NAMESPACE`	`${{ github.repository_owner }}`
`CI_PROJECT_TITLE`	`${{ github.event.repository.full_name }}`
`CI_PROJECT_URL`	`${{ github.server_url }}/${{ github.repository }}`
`CI_REPOSITORY_URL`	`${{ github.event.repository.clone_url }}`
`CI_RUNNER_EXECUTABLE_ARCH`	`${{ runner.os }}`
`CI_SERVER_HOST`	`${{ github.server_url }}`
`CI_SERVER_URL`	`${{ github.server_url }}`
`CI_SERVER`	`${{ github.actions }}`
`GITLAB_CI`	`${{ github.actions }}`
`GITLAB_USER_EMAIL`	`${{ github.actor }}`
`GITLAB_USER_ID`	`${{ github.actor }}`
`GITLAB_USER_LOGIN`	`${{ github.actor }}`
`GITLAB_USER_NAME`	`${{ github.actor }}`
`TRIGGER_PAYLOAD`	`${{ github.event_path }}`
`CI_MERGE_REQUEST_ASSIGNEES`	`${{ github.event.pull_request.assignees }}`
`CI_MERGE_REQUEST_ID`	`${{ github.event.pull_request.number }}`
`CI_MERGE_REQUEST_IID`	`${{ github.event.pull_request.number }}`
`CI_MERGE_REQUEST_LABELS`	`${{ github.event.pull_request.labels }}`
`CI_MERGE_REQUEST_MILESTONE`	`${{ github.event.pull_request.milestone }}`
`CI_MERGE_REQUEST_PROJECT_ID`	`${{ github.repository }}`
`CI_MERGE_REQUEST_PROJECT_PATH`	`${{ github.repository }}`
`CI_MERGE_REQUEST_PROJECT_URL`	`${{ github.server_url }}/${{ github.repository }}`
`CI_MERGE_REQUEST_REF_PATH`	`${{ github.ref }}`
`CI_MERGE_REQUEST_SOURCE_BRANCH_NAME`	`${{ github.event.pull_request.head.ref }}`
`CI_MERGE_REQUEST_SOURCE_BRANCH_SHA`	`${{ github.event.pull_request.head.sha}}`

`CI_MERGE_REQUEST_SOURCE_PROJECT_ID`	`${{ github.event.pull_request.head.repo.full_name }}`
`CI_MERGE_REQUEST_SOURCE_PROJECT_PATH`	`${{ github.event.pull_request.head.repo.full_name }}`
`CI_MERGE_REQUEST_SOURCE_PROJECT_URL`	`${{ github.event.pull_request.head.repo.url }}`
`CI_MERGE_REQUEST_TARGET_BRANCH_NAME`	`${{ github.event.pull_request.base.ref }}`
`CI_MERGE_REQUEST_TARGET_BRANCH_SHA`	`${{ github.event.pull_request.base.sha }}`
`CI_MERGE_REQUEST_TITLE`	`${{ github.event.pull_request.title }}`
`CI_EXTERNAL_PULL_REQUEST_IID`	`${{ github.event.pull_request.number }}`
`CI_EXTERNAL_PULL_REQUEST_SOURCE_REPOSITORY`	`${{ github.event.pull_request.head.repo.full_name }}`
`CI_EXTERNAL_PULL_REQUEST_TARGET_REPOSITORY`	`${{ github.event.pull_request.base.repo.full_name }}`
`CI_EXTERNAL_PULL_REQUEST_SOURCE_BRANCH_NAME`	`${{ github.event.pull_request.head.ref }}`
`CI_EXTERNAL_PULL_REQUEST_SOURCE_BRANCH_SHA`	`${{ github.event.pull_request.head.sha }}`
`CI_EXTERNAL_PULL_REQUEST_TARGET_BRANCH_NAME`	`${{ github.event.pull_request.base.ref }}`
`CI_EXTERNAL_PULL_REQUEST_TARGET_BRANCH_SHA`	`${{ github.event.pull_request.base.sha }}`

Legal notice

Portions have been adapted from https://github.com/github/gh-actions-importer/ under the MIT license: