# GitHub Actions Best Practice

«Automate your workflow from idea to production»

GitHub Actions (2.0) (https://github.com/features/actions) went into public access on November 13, 2019 at GitHub Universe. A simple concept, executed very well. Nat Friedman, the CEO of GitHub, did a writeup here (<https://github.blog/2019-08-08-github-actions-now-supports-ci-cd/>) back in August on what the new GitHub Actions feature will contain.

Despite the complexity this might entail, GitHub’s engineering team have developed a remarkably simple API: each Action runs in a sandbox in a virtual machine, and is described as a YML file which is stored directly in the repository. Actions are stored in Git repos, and can either live standalone within a repository or right alongside a repository’s source code.

With all that said, let’s take a look at some of the best practices we can use while writing code to live in this new environment.

## GitHub Actions Best Practices

‍

### 1. Keep your Actions minimal

Actions’ virtual machines have high bandwidth and are reasonably fast, but the longer an action takes to set up and run, the more time you spend waiting. Additionally, plans for GitHub Actions virtual machines are limited to a hard cap of free minutes per month -- 2,000 for the free plan.

A few seconds’ difference might not seem much when you’re first putting together an Action or workflow, but those seconds can add up quickly, depending on the event that triggers their usage. Therefore, one of the most important best practices to consider when creating a new Action is how to keep it as light as possible.

For example, if your Action runs in a container, make sure to use a light Docker image, such as alpine or alpine-node, and install as little as possible to keep down the time your Action is running, from initial boot up to having finishing its run.

This is important whether you are developing a standalone Action or building a CI/CD workflow, since your Action is set up and run in a clean environment every time, meaning all dependencies have to be downloaded and installed every time it’s run.

### 2. Don’t install dependencies unnecessarily

In keeping with the previous best practice, avoid installing dependencies where you can. There are a few different ways you can do this, but they boil down to two key strategies. First, if you’re publishing a standalone Action (and working on a Node-based project), publish the entire node\_modules folder in it.

Second, make sure to take advantage of GitHub’s caching mechanism wherever you can. You can look it up here, but the gist of it is that you need to provide a key This applies to both standalone Actions and also to Actions that run as part of a CI workflow.

‍

### 3. Never hardcode secrets

Secrets Video

‍One of the most powerful features of GitHub Actions is its encrypted secret handling. You can securely store secrets inside your repository’s settings, and then provide them as inputs or environment variables to your Actions at any time. It automatically redact any secrets that get logged on purpose or on accident (although the GitHub docs also recommend avoiding logging secrets, because the automated redaction not 100% accurate, especially when a secret is composed of structured data).

You can read more about secrets here, but the thrust of this best practice is obvious: instead of manually hardcoding secrets into your workflow (whether it’s private or public), set them manually in the repository settings and access them using environment variables or step inputs.

### 4. Limit environment variables to the narrowest possible scope

Hello World

Image from GitHub Actions documentation.

‍

GitHub allows you to specify variables in the YML file for the Action or Workflow at any scope. For example, you are able to specify an environment variable at the workflow level that any job or step can access. However, if you specify an environment variable at the job or step level, the levels above that, like workflow, won’t be able to access it. Additionally, environment variables sharing a name that are declared with a wider scope are overridden by variables with a tighter scope.

Accordingly, the best practice here is to prevent polluting the global environment context as much as possible by always declaring environment variables with the narrowest possible scope. This makes it easier to reason about what’s going on in a specific step or job, because the environment variables needed are right next to the work being done.

This becomes essential when using workflows that combine a number of Actions, jobs, and steps, as the number of environment variables can rise quickly. You can read further on environment variables in the GitHub docs here.

### 5. Ensure every repository contains a CI/CD workflow

If you’re relying on GitHub Actions as a CI/CD pipeline, you may want to ensure that every repository within your organization has a GitHub Actions workflow in place (and, possibly, in sync). It’s easy enough to manually check a few repositories, but across a significant number of repos, you expose yourself to human error and experience reduced visibility into which repositories are properly integrated and which are not.

You may want to consider using a tool like Datree to help you enforce that each repository within your organization or project contains a .github/workflows folder, which will help you keep track of what is being deployed properly. You can also take this a step further and verify that the workflows are in sync or using the correct servers by comparing Action files across repositories.

### 6. Store authors in Action metadata to promote code ownership

Metadata about actions is stored within the YML file that defines it. There is a lot of metadata you can store about an Action, including but not limited to inputs required by the action, outputs, branding, the entry point of the action, the author of the action, and more.

There are two reasons you might want to include the author of an action in this file. The first use case is for public actions -- obviously you’ll want to attribute the action to yourself for that sweet, sweet internet karma, and also in case anybody might have questions that could be directed to you. This use case is obvious.

But let’s say you’re working on a private Action for your company -- in this case, the internet karma argument doesn’t apply anymore, because it’ll never see the light of day. However, it’s still important to include an author to be in charge of maintaining the action and answer questions about it (this is similar to how the Chromium team does code ownership).

### 7. Don’t use self-hosted runners in a public repository

GitHub allows you to specify whether an Action will be run on a GitHub-hosted runner or a self-hosted runner via the `runs-on` property in the YML file that defines it.

GitHub documentation mentions the security risk in rather blasé fashion here, but the point bears stressing.

If you’re working on a private Action, self-hosted runners are fine and sometimes ideal, since they allow you to host the machines the Action runs on in-house. The upside to this is that Actions can run on machines can be more secure, performant, and optimized according to your needs.

However, the downside is that if you’re working on a public Action, somebody could fork it and submit a pull request for a workflow containing malicious code. That malicious code will then be executed by the Action on your self-hosted machine, and could easily escape its sandbox, invade your network, and do all sorts of Bad Things™️ -- even if you’re using Docker containers.

If your Action becomes popular it can be exposed to thousands of developers, and it only takes one to ruin your day. So please, play it safe and never use a self-hosted runner in a public GitHub action.

## Conclusion

We talked about Actions in two different contexts: writing individual Actions to be published to GitHub publicly and integrated for convenience within a repository, and writing CI/CD workflows containing Actions.

Some of these best practices are simply to be kept in mind when developing new Actions, and you’ll want to spread awareness of them within your team.

Others, however -- such as preventing secrets from being hardcoded, ensuring every repository contains a CI/CD workflow, and storing authors within Action metadata -- are better enforced programmatically. Sometimes, this is for security reasons and in other cases it’s because doing so can reduce the possibility of human error and improve organizational consistency.

We’ve built Datree to make exactly this kind of programmatic enforcement possible.

If you have questions about how you can use it to improve your coding practices, feel free to reach out and our team will be happy to answer any questions.