Using GitHub Actions to Run, Test, Build, and Deploy Docker Containers

9 minute read Updated: May 17, 2023

James Olaogun

GitHub Actions is a flexible tool that enables developers to automate a variety of processes, including developing, testing, and deploying, right from their GitHub repositories. The automation of Docker containers is no exception since GitHub Actions also enables developers to automate the process of developing containerized applications. As a result, developers can save time and focus on improving the overall quality of their software.

In this article, you'll learn how to use Github Actions to run, test, build, and deploy Docker containers using GitHub Actions.

What Is GitHub Actions?

As previously stated, GitHub Actions is a platform for automating software workflows directly from a GitHub repository. With GitHub Actions, you can create workflows that are triggered by events, such as a push to the main branch or the creation of a new pull request in your GitHub repo. These workflows then run a series of actions, such as checking the code for syntax errors, building a Docker image, or deploying to a hosting platform.

In this tutorial, you'll be making use of the GitHub Actions starter workflow examples. These examples can be used as a starting point for your automation

How to Run, Test, Build, and Deploy Docker Containers Using GitHub Actions

Before you begin this tutorial, you'll need the following:

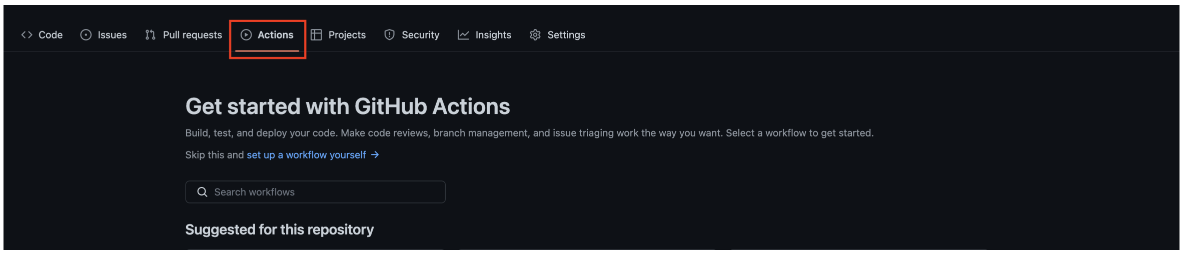
- GitHub account
- Basic knowledge of Docker and Docker Compose
- Basic knowledge of YAML files

Once these prerequisites are met, it's time to begin. You'll start by creating a sample workflow, and then set a runner for the workflow. After that, you'll learn how to set up GitHub Actions locally and then how to set up the build and test stage. Finally, you'll execute the workflow by running the action.

Create a Workflow

The first thing you need to do is create a workflow using GitHub Actions. This workflow defines the steps that GitHub Actions should take when triggered, including checking out the code, building a Docker container, running tests, and deploying the application.

To create a workflow, log into your GitHub account and navigate to the repo you want to automate. Select **Actions** from the navigation bar, which will take you to the **Actions** page:



Get started with GitHub Actions

Build, test, and deploy your code. Make code reviews, branch management, and issue triaging work the way you want. Select a workflow to get started.

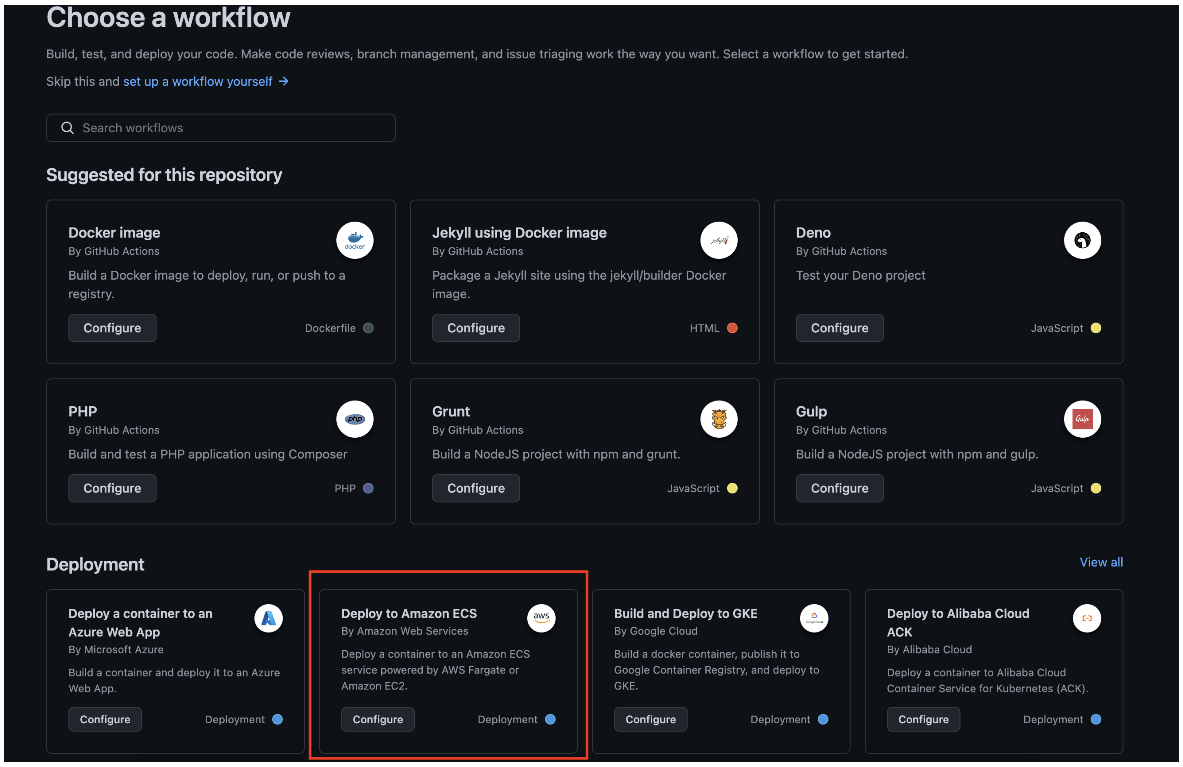
Skip this and set up a workflow yourself →

Suggested for this repository

GitHub Actions

At this point, you have two options: you can either select any of the workflow examples/templates or create a new one from scratch. Here, the **Deploy to**

Container Service (Amazon ECS).


Choose a workflow

Build, test, and deploy your code. Make code reviews, branch management, and issue triaging work the way you want. Select a workflow to get started.

Skip this and set up a workflow yourself →

Search workflows

Suggested for this repository

Docker image
By GitHub Actions
Build a Docker image to deploy, run, or push to a registry.
Configure Dockerfile

Jekyll using Docker image
By GitHub Actions
Package a Jekyll site using the jekyll/builder Docker image.
Configure HTML

Deno
By GitHub Actions
Test your Deno project
Configure JavaScript

PHP
By GitHub Actions
Build and test a PHP application using Composer
Configure PHP

Grunt
By GitHub Actions
Build a NodeJS project with npm and grunt.
Configure JavaScript

Gulp
By GitHub Actions
Build a NodeJS project with npm and gulp.
Configure JavaScript

Deployment View all

Deploy a container to an Azure Web App
By Microsoft Azure
Build a container and deploy it to an Azure Web App.
Configure Deployment

Deploy to Amazon ECS
By Amazon Web Services
Deploy a container to an Amazon ECS service powered by AWS Fargate or Amazon EC2.
Configure Deployment

Build and Deploy to GKE
By Google Cloud
Build a docker container, publish it to Google Container Registry, and deploy to GKE.
Configure Deployment

Deploy to Alibaba Cloud ACK
By Alibaba Cloud
Deploy a container to Alibaba Cloud Container Service for Kubernetes (ACK).
Configure Deployment

Deploy to Amazon ECS example template

The GitHub workflow configuration is always in YAML format, and you'll see many of the following popular parent key-value pairs in your workflows:

- `name` defines a unique name for the workflow.
- `on` specifies the trigger for the workflow, such as when a push is made to the repository or a pull request is opened.
- `jobs` contains one or more jobs that make up the workflow.
- `steps` specifies a list of steps to run in a job.
- `env` defines environment variables that will be used in the workflow.
- `runs-on` specifies the type of runner to use for a job.

Set Up a Runner

Once you've created your workflow, you need to set up a runner. A runner, in this context, is the environment or operating system that processes the actions when

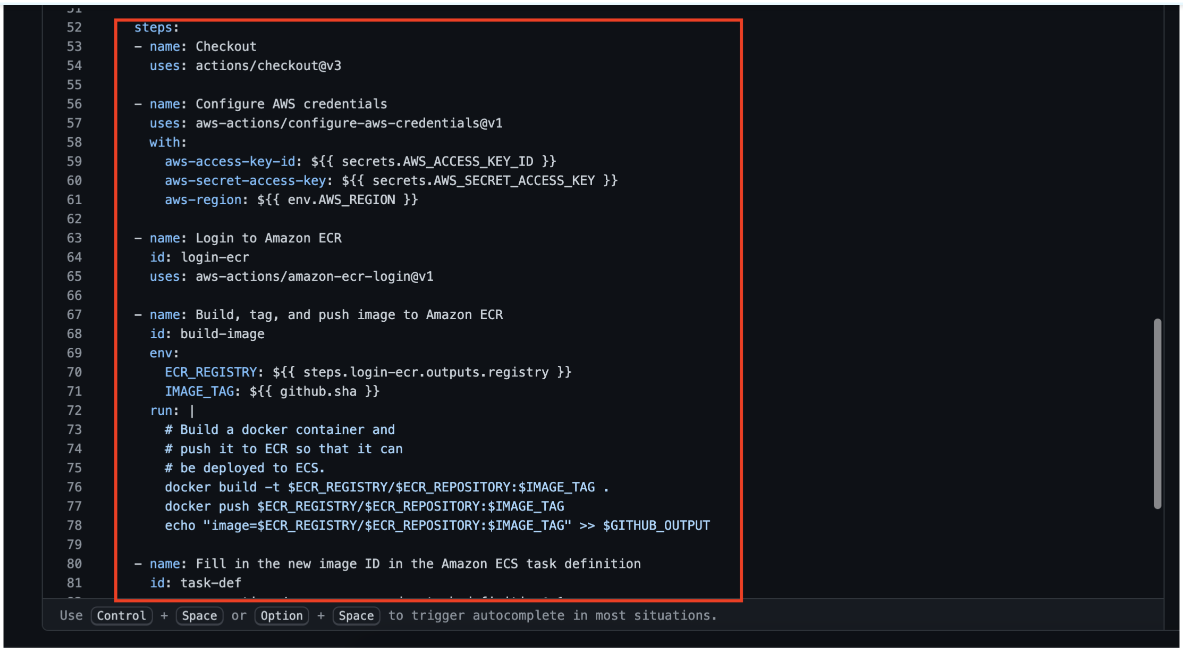
To set up a runner, open your workflow YAML file. Following the **Deploy to Amazon ECS** workflow template, you should see the `jobs` key with a `deploy` child key, followed by the `runs-on` key. The `runs-on` key is what defines the runner to be used for executing the job. See the following code snippet as an example:

```
jobs:
  deploy:
    name: Deploy
    runs-on: ubuntu-latest
    environment: production

    steps:
    - name: Checkout
      uses: actions/checkout@v3
…
```

In this code snippet, the runner is the `ubuntu-latest` runner, which is a GitHub-hosted runner provided by GitHub Actions. Aside from `ubuntu-latest`, there are several other GitHub-hosted runners, such as `windows-latest`, `macos-latest`, and `centos-8`, that you can use in GitHub Actions.

When the workflow is triggered, GitHub Actions will allocate an available runner of the specified type `ubuntu-latest`. The runner will then execute each step defined in the `steps` key, starting with the `Checkout` step. You can review the `steps` section in the **Deploy to Amazon ECS** as an example:



```yaml
steps:
- name: Checkout
  uses: actions/checkout@v3

- name: Configure AWS credentials
  uses: aws-actions/configure-aws-credentials@v1
  with:
    aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
    aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
    aws-region: ${{ env.AWS_REGION }}

- name: Login to Amazon ECR
  id: login-ecr
  uses: aws-actions/amazon-ecr-login@v1

- name: Build, tag, and push image to Amazon ECR
  id: build-image
  env:
    ECR_REGISTRY: ${{ steps.login-ecr.outputs.registry }}
    IMAGE_TAG: ${{ github.sha }}
  run: |
    # Build a docker container and
    # push it to ECR so that it can
    # be deployed to ECS.
    docker build -t $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG .
    docker push $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG
    echo "image=$ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG" >> $GITHUB_OUTPUT

- name: Fill in the new image ID in the Amazon ECS task definition
  id: task-def
```

Use Control + Space or Option + Space to trigger autocomplete in most situations.

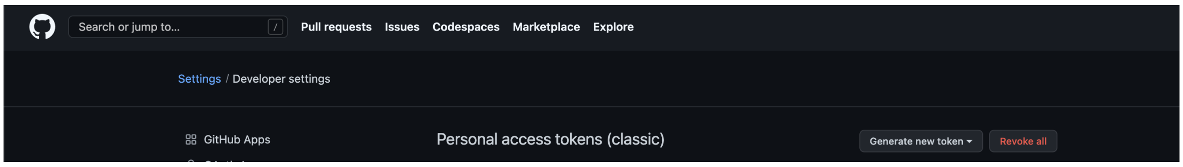
Deploy to Amazon ECS steps section

Set Up GitHub Actions Locally

Considering the limited build minutes that GitHub Actions provides, it's recommended that you conduct a test run of the workflow locally as the workflow will fail to complete if the allotted build time is exceeded. Additionally, setting up GitHub Actions locally allows you to test your workflow locally before you push it to your GitHub repository. This helps ensure that your workflow is working as expected and will run successfully on a GitHub Actions runner.

To set up GitHub Actions locally, you need to first clone or pull the latest changes from your GitHub repository that contains the workflow you want to run locally. Then change the directory to the root directory of the code.

Then install Act on your local machine. Act is a tool that will enable you to run your GitHub Actions locally. After the installation, connect your GitHub token to Act by logging into your GitHub account and navigating to **Settings > Developer settings > Personal access tokens (classic)**. You can either use an existing token if you still have access to it or generate a new token:

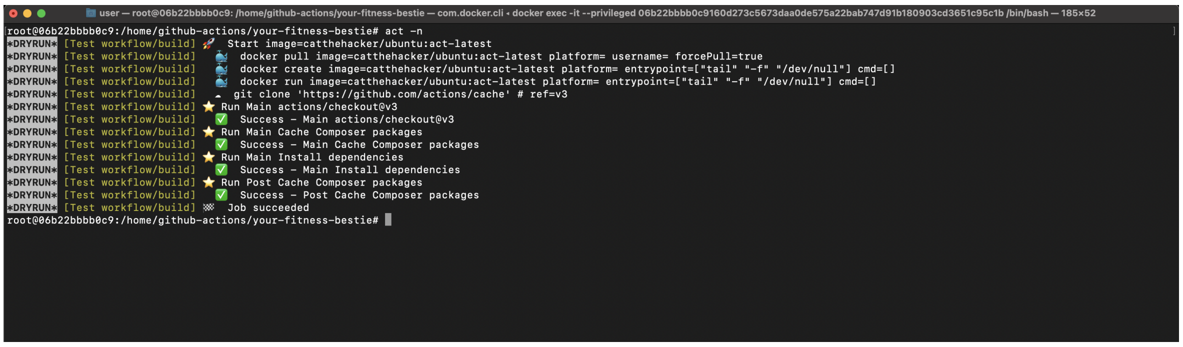


Personal access tokens (classic)

```
act -s GITHUB_TOKEN=
```

Make sure to replace ` **`with your generated token. If it is your first time running the command, you will be asked to select the default image you want to use with`** act`. You can select the “medium” image.

When finished, clone your GitHub repo with the workflow file if you haven’t, and proceed to use Act to run your GitHub Actions locally by running the `act -n` command to dry run the workflow. You should see the run log:



```
user — root@06b22bbbb0c9: /home/github-actions/your-fitness-bestie — com.docker.cli ◂ docker exec -it --privileged 06b22bbbb0c9160d273c5673daa0de575a22bab747d91b180903cd3651c95c1b /bin/bash — 185×52
root@06b22bbbb0c9:/home/github-actions/your-fitness-bestie# act -n
*DRYRUN* [Test workflow/build] 🚀  Start image=catthehacker/ubuntu:act-latest
*DRYRUN* [Test workflow/build]   🐳  docker pull image=catthehacker/ubuntu:act-latest platform= username= forcePull=true
*DRYRUN* [Test workflow/build]   🐳  docker create image=catthehacker/ubuntu:act-latest platform= entrypoint=["tail" "-f" "/dev/null"] cmd=[]
*DRYRUN* [Test workflow/build]   🐳  docker run image=catthehacker/ubuntu:act-latest platform= entrypoint=["tail" "-f" "/dev/null"] cmd=[]
*DRYRUN* [Test workflow/build]   ☁  git clone 'https://github.com/actions/cache' # ref=v3
*DRYRUN* [Test workflow/build] ⭐ Run Main actions/checkout@v3
*DRYRUN* [Test workflow/build]   ✅  Success - Main actions/checkout@v3
*DRYRUN* [Test workflow/build] ⭐ Run Main Cache Composer packages
*DRYRUN* [Test workflow/build]   ✅  Success - Main Cache Composer packages
*DRYRUN* [Test workflow/build] ⭐ Run Main Install dependencies
*DRYRUN* [Test workflow/build]   ✅  Success - Main Install dependencies
*DRYRUN* [Test workflow/build] ⭐ Run Post Cache Composer packages
*DRYRUN* [Test workflow/build]   ✅  Success - Post Cache Composer packages
*DRYRUN* [Test workflow/build] 🏁  Job succeeded
root@06b22bbbb0c9:/home/github-actions/your-fitness-bestie# █
```

Test workflow run log

You can also run other commands, including the command that runs a specific job, lists all actions for all events, or runs a specific event. You can check out the documentation to see a list of all the available commands.

Set Up the Build Stage

Now that you’ve gotten your GitHub Actions to work locally, you can use it to debug and test your workflow locally, make any necessary changes to the workflow, and rerun it until you’re satisfied with the results before pushing your code to GitHub.

Once each step defined in the `steps` key of your workflow file is tested and working as expected, you need to run your `docker-compose` file or build your Docker images. To do that, add a new value to the `steps` key with a `name` , `id` , and `run` key. It may also have an `env` key if the Docker image requires a passkey.

```
- name: Build docker images
  Id: build-image
  run: |
    echo ---Building images and starting up docker---
    {{docker build [image-url] or docker-compose -f \
    [docker-compose file] up -d }}
    echo ---Containers up—
```

Following the **Deploy to Amazon ECS** workflow template that's been used here, the build step can be found on line 67:

```
- name: Build, tag, and push image to Amazon ECR
      id: build-image
      env:
        ECR_REGISTRY: ${{ steps.login-ecr.outputs.registry }}
        IMAGE_TAG: ${{ github.sha }}
      run: |
        # Build a docker container and
        # push it to ECR so that it can
        # be deployed to ECS.
        docker build -t $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG .
        docker push $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG
        echo "image=$ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG" >> \
        $GITHUB_OUTPUT
```

This template's build step uses the `env` key since `ECR_REGISTRY` requires a login and SHA key.

Set Up the Test Stage

Once you're done setting up the build stage, the next thing you need to do is set up the test stage. Because the **Deploy to Amazon ECS** workflow template is primarily a deployment template, the test stage isn't factored into it. However, the test stage is the recommended phase since it helps you verify the functionality and connections of your applications (including the workflow file) before pushing to GitHub to run the workflow actions.

To set up the test stage, you can either create a test folder in your application or in a different folder outside your application, then develop all your test cases. Afterward, ensure your test folder is containerized in the same network as your application. If you're using Docker Compose, GitHub Actions will run the workflow in a user-defined network. And for communication to happen, your

The purpose of this network is to provide a Docker network for the containers defined in the **`docker-compose`** file, allowing them to communicate with each other and with the host system. Containers in separate networks cannot communicate, which means that API test cases won't be able to get a response from the services in other networks. You can learn more about creating a **`docker-compose`** file with a network in this article.

When you're done adding the test cases folder to the same container network, you need to add a new value to the **`steps`** key of your workflow file. The new value will contain a **`name`**, **`id`**, and **`run`** key:

```
- name: Run test cases
   id: run-test-cases
  run: |
    echo --- Running test cases ---
    docker-compose -f  -p  \
    up --build --exit-code-from
    echo --- Completed test cases ---
```

In this code, **`is the name of your `docker-compose` file,`** is the project name, and **`[`** is the container name.

As an alternative to using a **`docker-compose`** file, you can leverage Earthly, which lets you define your containers and their dependencies, as well as specify your entire build process, including testing and deployment, in one **Earthfile**.

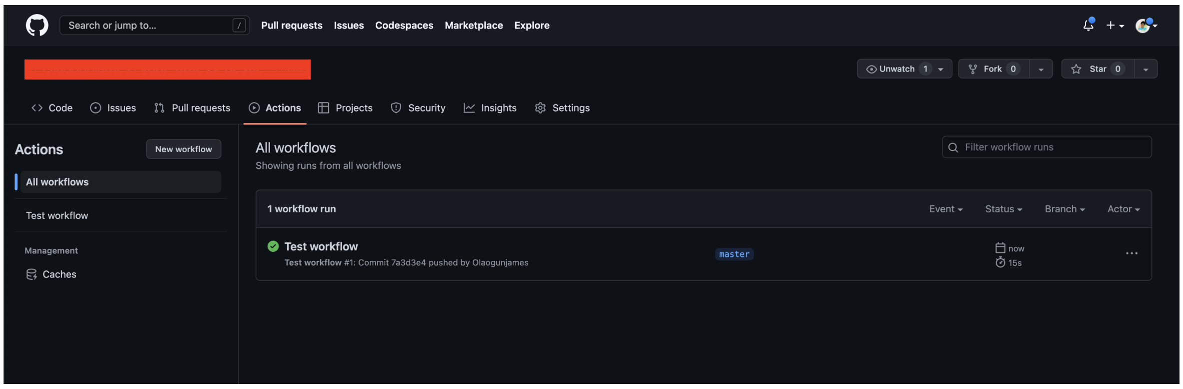
In addition, Earthly allows you to define reusable builds that you can use across different machines, making it easier to collaborate and enabling you to run your builds anywhere.

Moreover, Earthly caches your build components, making the process efficient and fast. Check out the official documentation to learn more about getting started with Earthly.

Run the Action

repository. The workflow will be triggered based on the branches you set it to monitor on the `on` key of your workflow file.

You can also monitor the progress of the workflow by visiting the **Actions** tab in your repository on GitHub and selecting a workflow that has previously been run or the currently running workflow:



[redacted]

Actions

All workflows

Test workflow

Management

Caches

All workflows
Showing runs from all workflows

1 workflow run

Test workflow
Test workflow #1: Commit 7a3d3e4 pushed by Olaogunjames
master
now
15s

Workflow history

Conclusion

In this tutorial, you learned how to create a new workflow, edit an existing workflow, set up the runner for your workflow, and locally set up and work with GitHub Actions. At this point, you should be confident that you can build GitHub Actions for your projects and speed up the development processes.

After reading this article, it's recommended that you explore and experiment with the various actions, events, and workflows available in the GitHub Actions ecosystem to further enhance and automate your development processes.

If you want to see how Github Actions stacks up against other CI services, check out our comparison of [CI Free Tiers](#).

Earthly makes CI/CD super simple

Fast, repeatable CI/CD with an instantly familiar syntax – like Dockerfile and Makefile had a baby.

Go to our homepage to learn more

James Olaogun is an innovative software developer who loves to collaborate to achieve a desired goal.

Writers at Earthly work closely with our talented editors to help them create high quality tutorials. This article was edited by:

Bala Priya C

Bala Priya C

Bala is a technical writer who enjoys creating long-form content. Her areas of interest include math and programming. She shares her learning with the developer community by authoring tutorials, how-to guides, and more.

Published: May 17, 2023

You may also enjoy

Introducing Earthly: build automation for the container era

3 minute read

We live in an era of continuous delivery, containers, automation, rich set of programming languages, varying code structures (mono/poly-repos) and open-sour...

Hello world! We have partnered up with some cool people in Silicon Valley [^1] to fix the world of CI. So today we are launching Earthly CI, the world's fir...

The world deserves better builds

4 minute read

Hello, developers of planet Earth! Earlier this year, we at Earthly embarked on a journey to bring better builds to the world. We started with a deep belief...

Getting a Repeatable Build, Every Time

28 minute read

I wanted to sit down and write about all the tricks we learned and that we used every day to help make builds more manageable in the absence of Earthly.

Better Dependency Management in Python

6 minute read

Can We Build Better?

4 minute read

Have you ever had a test fail in the build but not locally? I have. Have you ever then burnt half a day pushing small changes and waiting for your build to ...

Earthly Switches to Open-source

10 minute read

EARTHLY
Super Simple Builds

The Platform Values of Earthly

10 minute read

As creators of a new approach to build automation, we have always strived to create products that we ourselves would have wished we had. While this may sound...