I have tried to keep the notes as simple as possible with the screenshots from the course and a ton of reference materials. If you benefitted from this note in any manner, please consider buying me a coffee here so that I can churn caffeine to notes for the next course 🙂 Also, do not forget to put the next ML course you would like me to cover in the comment section here

GitHub Actions

Event Triggers

There are several events that can trigger GitHub actions.
Reference: Events that trigger workflows - GitHub Docs

Note: GitHub handles issues and pull_request similarly. So, if you want to trigger an action for a pull request comment, use `issue_comment`.

wandb-cicd / .github / workflows / trigger.yaml in main

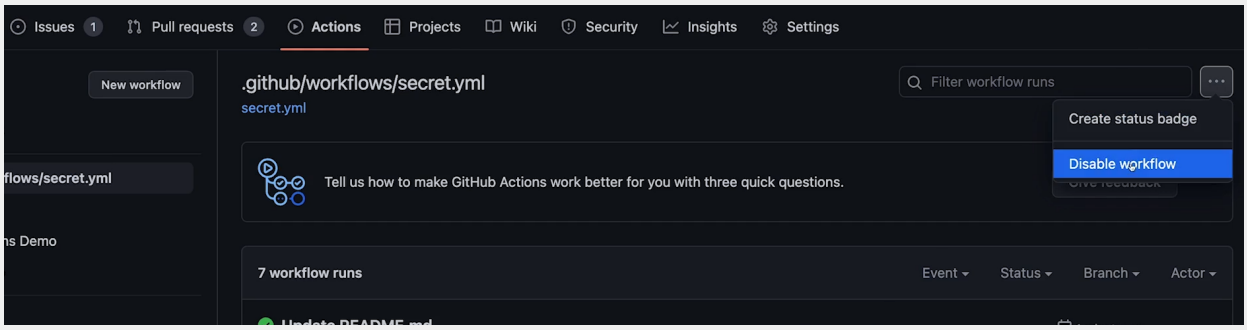
Edit new file | Preview

```yaml
name: trigger-demo
on:
  push:
    branches:
      - main
  pull_request:
  workflow_dispatch:

jobs:
  trigger-demo:
    runs-on: ubuntu-latest
    steps:
      - run: echo "hello"
```

Above workflow triggers `trigger-demo` workflow if someone pushes `push` or manually starts the workflow `workflow_dispatch` in the `main` branch. The workflow will utilize the latest ubuntu `ubuntu-latest` and print hello `run: echo “hello”`

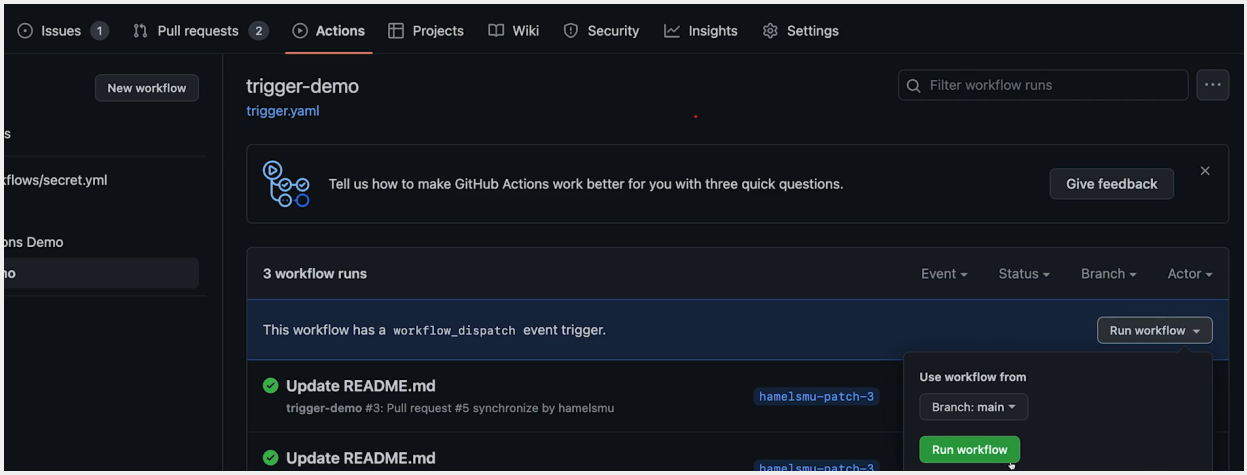
GitHub allows you to block PRs that do not pass the test.

GitHub Actions can also be manually disabled (shown below)



You can also manually run the disabled workflow by clicking `Run workflow` (look for a blue banner) if the `workflow_dispatch` setting is used. You can choose any branch to run the

workflow manually. There are also ways to set parameters to make it behave like your own (sort of) application.



Also, notice above that each run is followed by what triggered it. For instance, `trigger-demo #3` is triggered by `Pull request #5` by `hamelsmu`. If you manually run the workflow, you will see `Manually run by <username>`.

Setting up GitHub Actions environment

Typically, workflows follow a pattern → (i) clone the repo to the environment `uses: actions/checkout@v3` [GitHub - actions/checkout: Action for checking out a repo]() , (ii) install requirements/ dependencies and (iii) run code

```yaml
name: GitHub Actions Demo

on: [push]

jobs:
  my-first-job:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: hello
        run: |
          echo "hello world"
      - name: run python script
        run: |
          pip install -r requirements.txt
          python ci.py
```

To set up an environment variable, one may simply use `env` like below

```yaml
env:
  PUSHER: ${{ github.event.pusher.name }}
  EMAIL: ${{ github.event.pusher.email }}
```

PUSHER and **EMAIL** are two environment variables.
You may also use default environment variables
https://docs.github.com/en/actions/learn-github-actions/variables#default-environment-variables

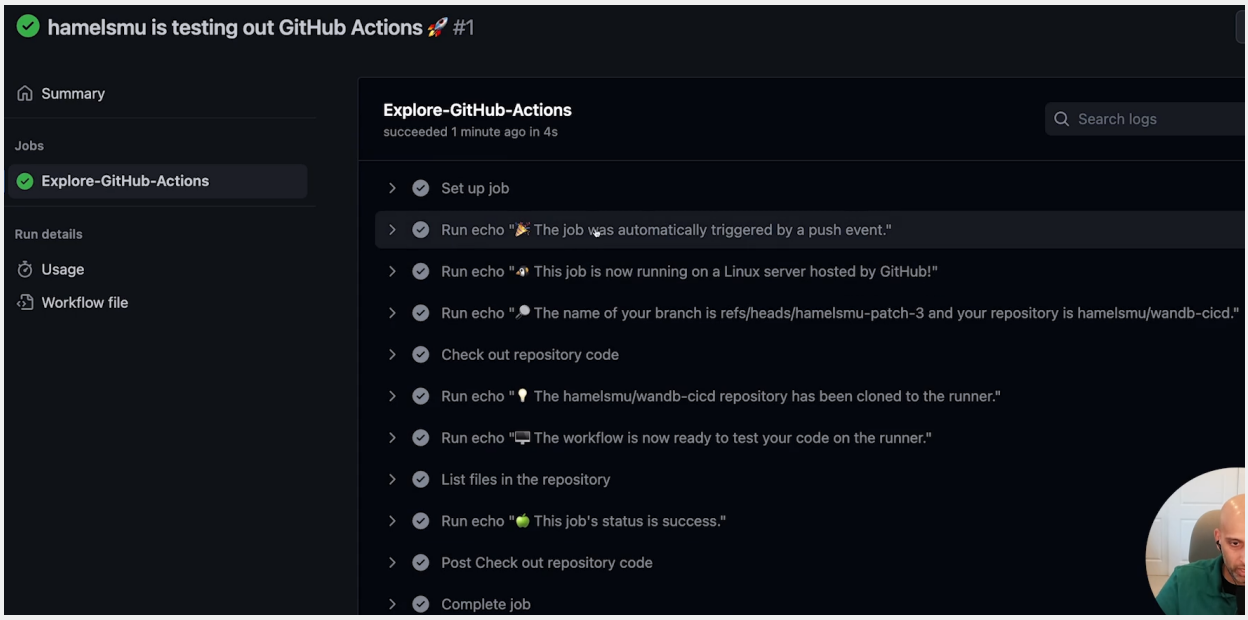
Special Variables

Reference: Variables - GitHub Docs + Quickstart for GitHub Actions

Special variables, aka **context**, are represented by ${{ <special_variable> }}. Think of them as JSON objects. Also, some of these are event dependent.
To learn more: Contexts - GitHub Docs

```yaml
name: GitHub Actions Demo
run-name: ${{ github.actor }} is testing out GitHub Actions 🚀
on: [push]
jobs:
  Explore-GitHub-Actions:
    runs-on: ubuntu-latest
    steps:
      - run: echo "🎉 The job was automatically triggered by a ${{ github.event_name }} event."
      - run: echo "🐧 This job is now running on a ${{ runner.os }} server hosted by GitHub!"
      - run: echo "🔎 The name of your branch is ${{ github.ref }} and your repository is ${{ github.repository }}."
      - name: Check out repository code
        uses: actions/checkout@v3
      - run: echo "💡 The ${{ github.repository }} repository has been cloned to the runner."
      - run: echo "🖥️ The workflow is now ready to test your code on the runner."
      - name: List files in the repository
        run: |
          ls ${{ github.workspace }}
      - run: echo "🍏 This job's status is ${{ job.status }}."
```

If you run the above workflow, you shall receive an output like below. Note all the special variables are replaced by the respective values during the run. Did you notice an apple? 🙂



You may use `toJSON()` which is a [Expressions - GitHub Docs]() to print out the JSON object. Think of them as a limited programming language inside GitHub Actions.
For example `${{ toJSON(<github.event>) }}`

Using variables inside GitHub Actions

Reference: [Defining outputs for jobs - GitHub Docs]()

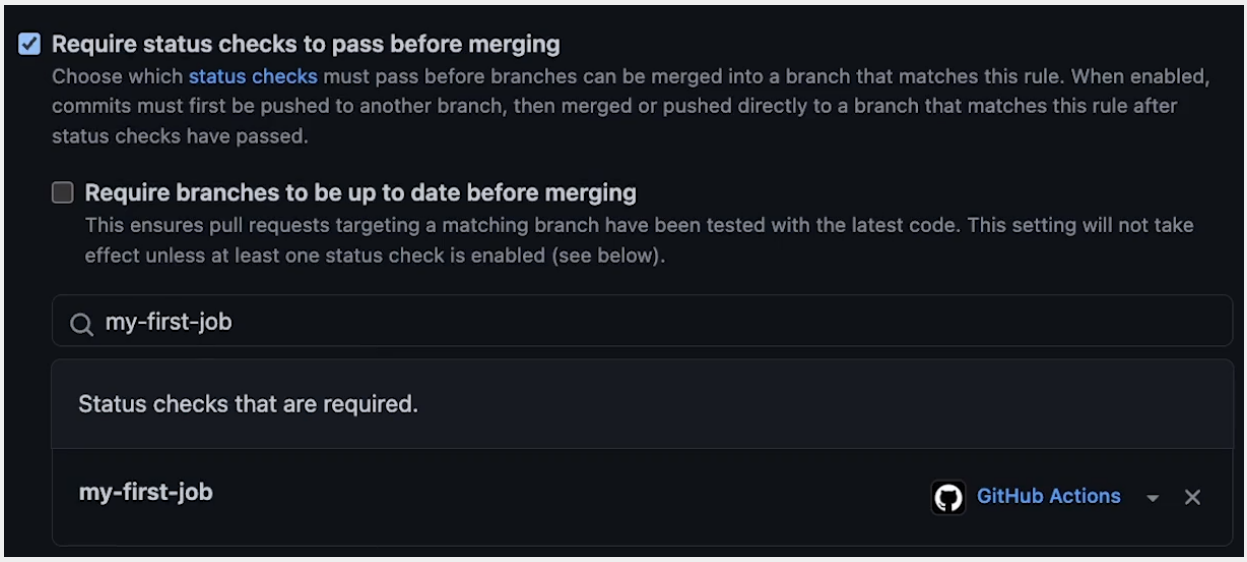
One of the handy use cases would be to grab an output in a certain step and use that in the subsequent steps.
To do so, one needs to set an `id` as shown below

```yaml
name: GitHub Actions Demo
on: [push]

jobs:
  io:
    runs-on: ubuntu-latest
    steps:
      - name: set-var
        id: set-var
        shell: python
        run: |
          import os
          with open(os.environ['GITHUB_OUTPUT'], 'a') as f:
            print(f'VAL=true', file=f)
      - name: print value
        run: echo "the value is ${{ steps.set-var.outputs.VAL }}"

```

Now, you can grab onto the `id` in the following steps.
For example, in the above snippet, VAL is set to true in `set-var` step and is echoed in the `print value` step by using special variable `steps.<id>.outputs.<variable_name>`

Branch Protection Rule

Reference: [Managing protected branches - GitHub Docs]
To enable branch protection rules, go to Setting → Branches → Branch protection rules → Add branch protection rule
Pro tip: Use the name of the job instead of the workflow name to create a branch protection rule: for instance, disable merging without the test being passed.



Testing ML Code

Introduction

Testing can be broadly classified into two categories:

- Unit tests: Test individual components of your code
- Smoke tests: Test the code as a whole for stability (ex: running the training iteration with a tiny dataset to make sure the ML pipeline is bug-free)

Pytest is an excellent framework to start with.

Testing ML code walkthrough

Once you point your Pytest module to a particular directory, it will look for all the files and the functions inside them with the prefix `test`

main wandb-cicd / test-example / test_metrics.py

Code Blame 39 lines (33 loc) · 1.38 KB

```python
def compute_val(met, x1, x2):
    learn = TstLearner()
    for i in range(3):
        learn.pred,learn.yb = x1[vals[i]:vals[i+1]],(x2[vals[i]:vals[i+1]],)
        met.accumulate(learn)
    return met.value

def test_metrics():
    x1a = torch.ones(20,1,1,1) # predicting background pixels
    x1b = torch.clone(x1a)*0.3
    x1c = torch.clone(x1a)*0.1
    x1 = torch.cat((x1a,x1b,x1c),dim=1)   # Prediction: 20xClass0
    x2 = torch.zeros(20,1,1)              # Target: 20xClass0

    assert compute_val(utils.BackgroundIOU(), x1, x2) == 1.
    road_iou = compute_val(utils.RoadIOU(), x1, x2)
    assert math.isnan(road_iou)


    x1b = torch.ones(20,1,1,1) # predicting road pixels
    x1a = torch.clone(x1a)*0.3
    x1c = torch.clone(x1a)*0.1
    x1 = torch.cat((x1a,x1b,x1c),dim=1)   # Prediction: 20xClass1
    x2 = torch.ones(20,1,1)               # Target: 20xClass1
    background_iou = compute_val(utils.BackgroundIOU(), x1, x2)
    assert math.isnan(background_iou)
```

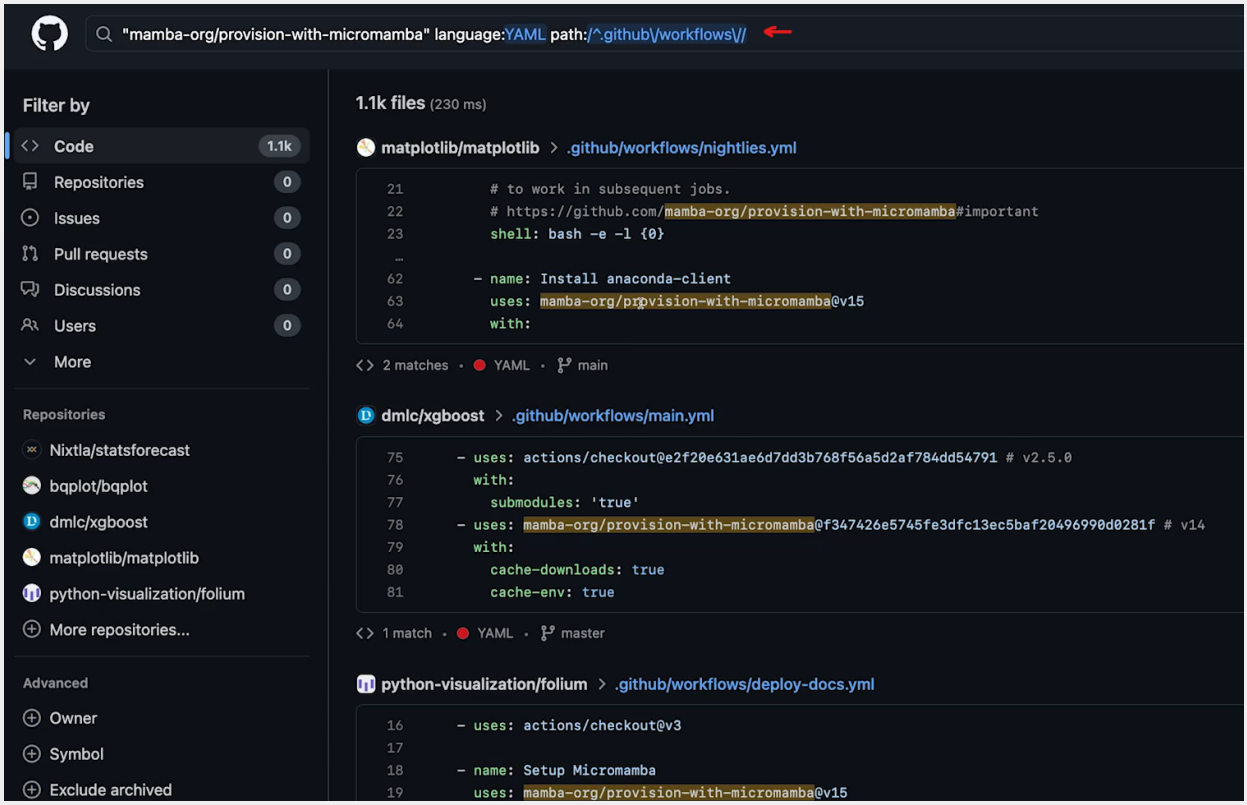
You can run such tests using GitHub Actions which bolsters high quality code maintenance regime. To be able to integrate tests like these, you will need to make sure the environment variable (if any) is set correctly and all the dependencies are installed correctly in the GitHub Actions environment you are using. You may use `pip` or `conda` to install such dependencies.

Pro tip: `mamba` might install certain dependencies faster than `conda` and caching dependencies

References

- https://github.com/conda-incubator/setup-miniconda#example-6-mamba
- https://github.com/mamba-org/provision-with-micromamba

<u>Pro tip</u>: Search repositories that use `mamba` in their workflow file to learn how to use it effectively in your project



To override the default Python version GitHub is using in your environment, you can set the following

```
steps:
- uses: actions/checkout@v3
- uses: actions/setup-python@v4
  with:
    python-version: '3.10'
- run: python my_script.py
```

You can also use `defaults` to apply a common setting for ALL the jobs in the workflow

`defaults`

Use `defaults` to create a `map` of default settings that will apply to all jobs in the workflow. You can also set default settings that are only available to a job. For more information, see `jobs.<job_id>.defaults`.

When more than one default setting is defined with the same name, GitHub uses the most specific default setting. For example, a default setting defined in a job will override a default setting that has the same name defined in a workflow.

You can also use `cache` to cache the files for the installed libraries inside actions and mount them into the actions runtime so that you do not have to install them every time the workflow runs.

Sample workflow file that uses above-discussed concepts is below:

```yaml
name: "Unit & Smoke Tests - mamba"
on:
  pull_request:
    paths:
      - test-example/**

  workflow_dispatch:

defaults:
  run:
    shell: bash -el {0}

jobs:
  test:
    runs-on: ubuntu-latest
    steps:

      - name: Copy Repository Contents
        uses: actions/checkout@main

      - name: Install Conda environment with Micromamba
        uses: mamba-org/provision-with-micromamba@main
        with:
          environment-file: test-example/conda-environment.yaml
          environment-name: course
          cache-env: true
          cache-downloads: true

      - run: |
          micromamba activate course
          pip install -U pytest
          pytest test-example/
        env:
          WANDB_API_KEY: ${{ secrets.WANDB_API_KEY }}
```

GitHub API

GitHub API and Actions

GitHub Actions let you run arbitrary code within GitHub. However, GitHub API lets you do powerful tasks such as making comments, applying labels, and many more.

Recommended ways to use GitHub API:

- Ghapi: third-party Python-based and maintained by fast.ai → ghapi
- Octokit: Official JavaScript based and maintained by GitHub → octokit/rest.js

GitHub API

GitHub helps to interact with GitHub to automate certain CI/CD pipelines.

To use GitHub API in GitHub actions, you will need to set specific permissions in your workflow file

```yaml
name: ghapi
on: pull_request

permissions:
  contents: read
  issues: write
  pull-requests: write

jobs:
  ghapi:
    runs-on: ubuntu-latest
    steps:
    - name: install dependencies
      run: pip install ghapi

    - name: Make a comment
      shell: python
      run: |
        import os
        from ghapi.core import GhApi
        owner,repo = os.environ['REPO'].split('/')
        api = GhApi(owner=owner, repo=repo)
```

Note that the above workflow runs all the Python commands in a Python shell. A best practice is to put all the codes in a .py file and execute the file by calling the Python command.

GitHub Actions come pre-loaded with a secret token which lets it do certain activities in GitHub, For GitHub API to make use of it and do similar activities, you will need to dump `secrets.GITHUB_TOKEN` to a certain environment variable for the API to consume. For instance, GhApi looks for the token in the `GITHUB_TOKEN` environment variable by default. Also, other dumped environment variables can be used by GhApi.

```yaml
env:
  NUMBER: ${{ github.event.number }}
  GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
  REPO: ${{ github.repository }}
```

If you find any functionalities missing in the third-party GitHub API, you may defer to using the official API octokit/rest.js

Even if you do not know much of JavaScript, you should be okay using it with built boilerplates. You will need to install npm and set up a script with some sort of authentication to use it. You may use environment variables like the previous one to set up a token for authentication.

A simple script using octokit/rest.js that creates a comment on an issue is shown below

```javascript
const { Octokit } = require("@octokit/rest");
// install with npm install @octokit/rest

const octokit = new Octokit({
auth: process.env.GITHUB_TOKEN
});

// Create an issue commment
octokit.issues.createComment({
    issue_number: 6,
    owner: 'hamelsmu',
    repo: 'wandb-cicd',
    body: "Hello! I'm making a comment from `octokit.js!`"
});
```

You may also use GitHub Script to execute octokit/rest.js without requiring any authentication setup or importing any specific libraries. However, you will need to set GitHub Actions up with the required actions permission Assigning permissions to jobs - GitHub Docs

Below is an example to comment on an issue using GitHub Script.

Comment on an issue

```yaml
on:
  issues:
    types: [opened]

jobs:
  comment:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/github-script@v6
        with:
          script: |
            github.rest.issues.createComment({
              issue_number: context.issue.number,
              owner: context.repo.owner,
              repo: context.repo.repo,
              body: '👋 Thanks for reporting!'
            })
```

GitHub Actions for ML

Using W&B API

Reference: https://docs.wandb.ai/guides

To be able to use Weights & Biases (wandb) API, you will need wandb API key.
The model registry helps you organize your model artifacts and version them.

Most of this section is better explained in the course video.

Promoting a model to the registry

To promote a model to the registry, you may use the steps below

Promote a model to the registry

This is also might be something you could do as part of a CI/CD process.

```
path = 'av-team/model-registry/BDD Semantic Segmentation'
```

```
run_path = 'av-team/mlops-course-001/2wpuuc70' # this is the run id
api = wandb.Api()
run = api.run(run_path)
```

Get the model from the run and promote it to the registry

```
art = [a for a in run.logged_artifacts() if a.type == 'model']

if art:
    assert len(art) == 1, 'More then 1 artifact of type model!'
    art[0].link(path, aliases=['new-model-demo'])
```

We can see a model in the registry like this:

```
api = wandb.Api()
versions = api.artifact_versions('model', path)
```

```
for v in versions:
    print(f'Model version: {v.version} with tags: {v.aliases}')
```

```
Model version: v3 with tags: ['new-model-demo', 'latest']
Model version: v2 with tags: ['prod candidate', 'demo']
Model version: v1 with tags: ['staging', 'production']
Model version: v0 with tags: []
```

Here are a few questions you might need to answer yourself before starting to build a full end-to-end CI/CD pipeline for an ML project.

Problem: When Do We Do A Training Run?

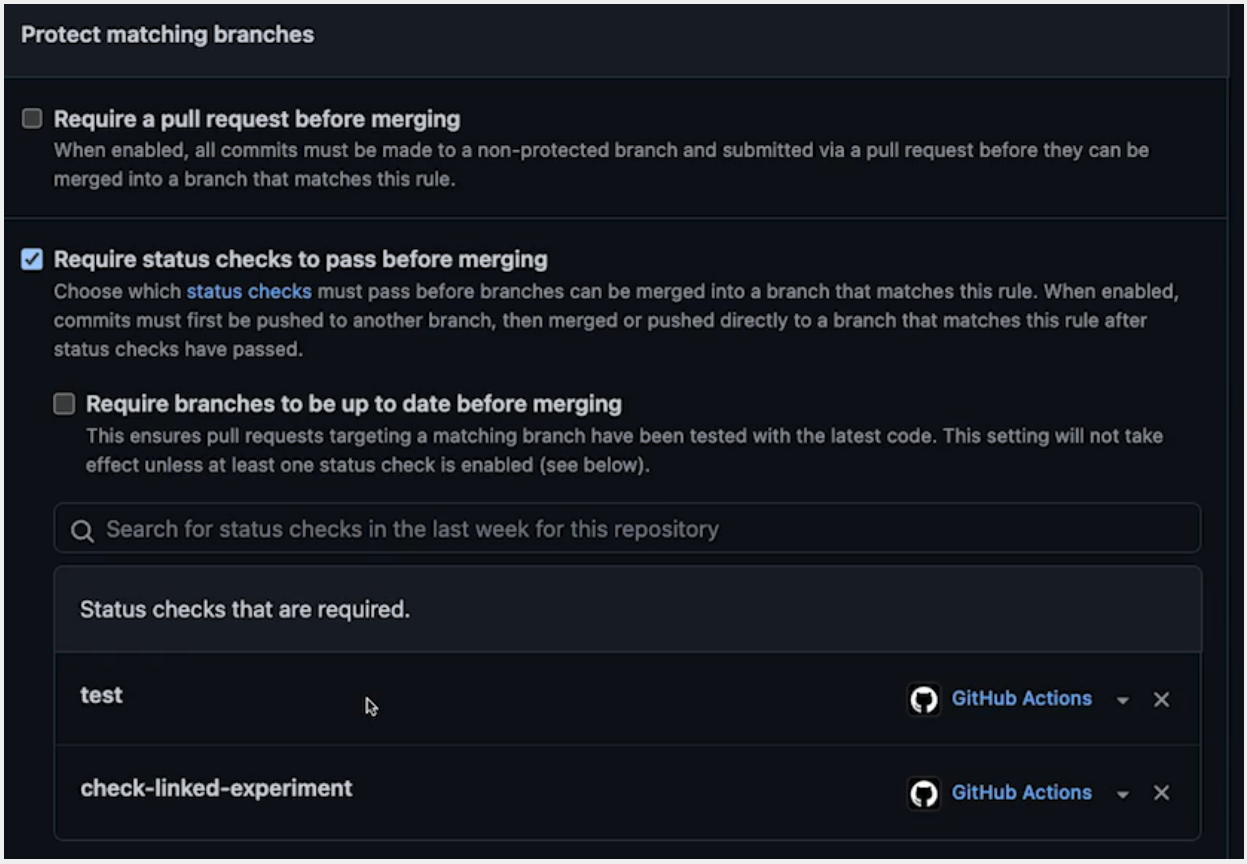
- Before you open the PR?
- After you open the PR?
- On every push/commit?
- What if you make a trivial change? Do you need to redo the whole training run?
  - Consider time/expense of running model

… <u>There is no right answer</u>! We can "manually" link runs to PRs to get started!

Branch protecton with W&B API

Branch protection works on any workflows that create check runs. The GitHub API has check runs (the green/yellow/red) dot on Pull Requests. An easy way to create a check run is by creating events triggered by "pull_request" in Actions.

For instance, the following shows two status checks enabled before merging that perform check runs.

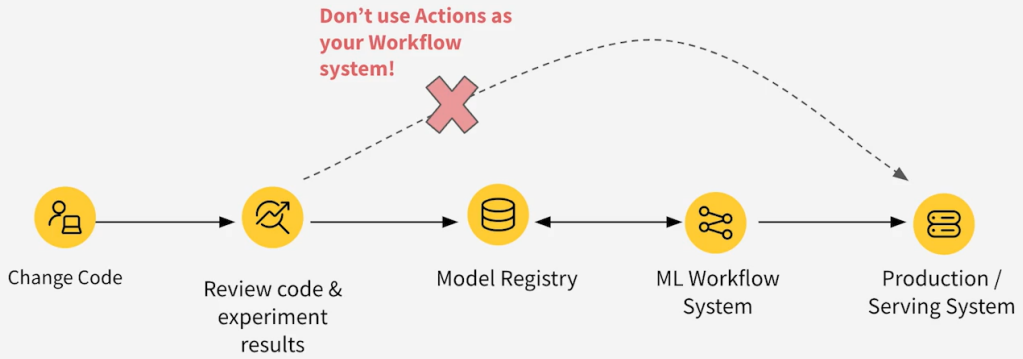


<u>Pro tip</u>: One GitHub Action cannot be run by another GitHub Action. One of the design patterns is to create something like a label from one GitHub Action which triggers another GitHub Action that checks for the label.

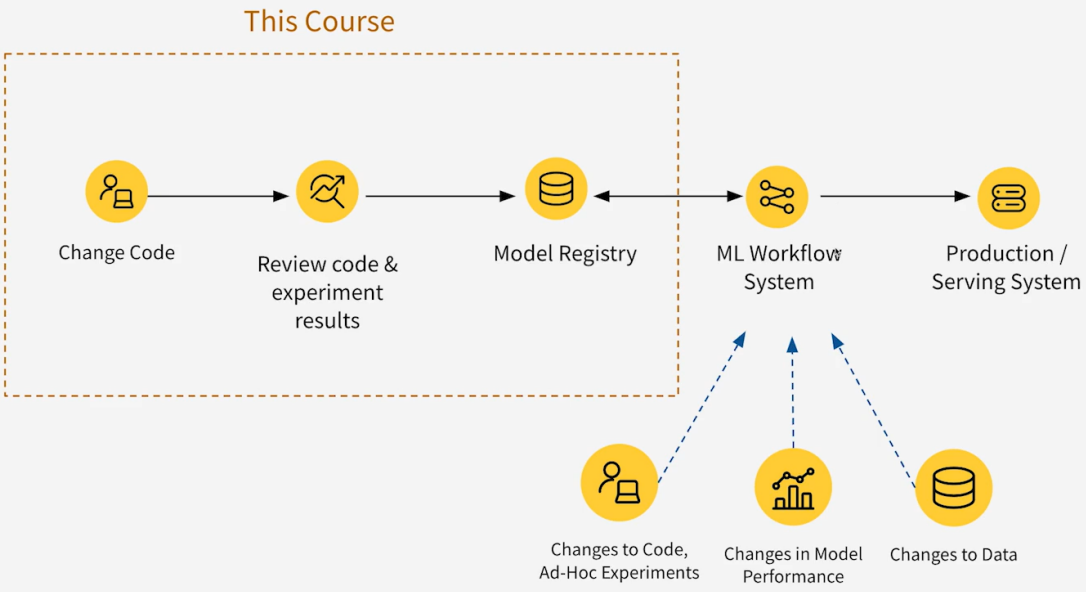
Deployment with GitOps

General workflow

Deployment With GitOps 🚀



Reminder: GitOps is Only A Small Part



Reference: [Deployments - GitHub Docs]

Advanced topics for Independent Study

- Finding and using actions from others. Ex: marketplace
- How to create your own third-party actions.
    - Ex: repo2docker
- “External” event triggers: workflow dispatch, repository dispatch
- Self-hosted runners
- Reusable workflows
    - Ex: fastai/workflows
- Caching

Other resources

- W&B Fully Connected - blog with a regular fresh dose of ML and MLOps knowledge
- W&B docs - the go-to place for anything W&B
- W&B Discord - the place where you can hang out with the W&B community