Open in app Get started

tds Published in Towards Data Science

Mark Douthwaite Follow
Nov 5, 2020 · 13 min read · Listen

Save

GETTING STARTED

A Gentle Introduction: Automating Machine Learning Pipelines

Erlend Ekseth on Unsplash

Deployment is hard

Deploying software regularly and reliably is hard. Deploying software that utilises Machine Learning (ML) models regularly and reliably can be harder still. At the end of

That's where automation can come in very handy: *careful* automation of ML pipelines can massively boost your productivity by allowing you to rapidly iterate on a pipeline in order to account for new business logic or modelling changes, while also ensuring those changes meet key performance criteria *before* going into service with your stakeholders/customers.

Sounds cool, right? But do *you* need automated ML pipelines? Well that question largely boils down to the following question

- Can *you* (as an ML practitioner) *quickly* and *confidently* release changes 'into production' with a single `git` commit?

In other words: can you release updates to your model and pipeline quickly and confidently? Do you have a systematic process for evaluating and testing the behaviour of your model *and* pipeline (including business logic, transformations etc) for each update you make?

If the answer to these sorts of questions is 'no' (or you're unsure!), then this post is for you!

What you'll learn

Here's what you should get from this post:

- An understanding of some of the key ideas and motivations behind the movement focussed on improving the level of reliability and automation in ML systems (often referred to as MLOps).
- An understanding of how to automate a basic ML pipeline using GitHub Actions and Google Cloud. You'll be using free services (or services with free tiers/introductory offers), so it shouldn't cost you a dime!
- A demo project that lets you deploy a Scikit-Learn Pipeline as a 'production ready' Serverless function! It's a template project too, so you can use it as the basis for your own ML projects!

As always, for those of you wanting to get stuck in to the code, check out the GitHub

A repository providing demo code for deploying a lightweight Scikit-Learn based ML pipeline modelling heart disease

github.com

If you know what MLOps is all about and just want to follow the tutorial, feel free to skip ahead.

What is MLOps?

In the last decade or so, the 'DevOps' movement has gained a significant professional following within the world of software engineering, with a large number of dedicated DevOps roles springing up across development teams around the world. The motivation for this movement is to combine aspects of software development (Dev) with elements of Operational (Ops) software activities with the aim of accelerating the delivery of reliable, working software on an ongoing basis.

A major focus of the DevOps movement is on establishing and maintaining Continuous Integration and Continuous Delivery (CI/CD) pipelines. In practice, well designed and cleanly implemented CI/CD pipelines offer teams utilising them the ability to continuously modify their software system to (in principle) dramatically reduce the time-to-value for new software patches and features, while simultaneously minimising the risk of downside from bugs and outages related to releasing these patches and features. Teams operating mature implementations of this delivery mechanism often release updates on an hourly basis (or faster!) with the ability to quickly and cleanly rollback changes if they introduce a bug (though most of these *should* be caught somewhere in the pipeline).

In contrast, 'traditional' approaches to releasing software essentially stockpile fixes and features for predefined release windows (perhaps on a weekly, monthly or quarterly basis). While this sort of approach is not *uniformly* a poor approach, it does introduce a lot of pressure around the release window, can create a lot of complexity around the product integration and release process, and ultimately heighten the risk of serious service outages and by extension brand damage.

Open in app Get started

service trust the *validity* and *stability* of the service they consume. Importantly, *this must be on an ongoing basis* — potentially for years at a time.

Consequently, *ad hoc* model training and evaluation activities that require significant manual intervention akin to 'traditional' software delivery processes on the part of Data Science teams — an all too common sight in the world of Data Science — can introduce serious technical and business risk into a service's lifecycle, and should be thought of as particularly pernicious form of technical debt.

> Much like successful commercial software products, successful commercial Machine Learning (ML) projects require that the users of the ML service trust the validity and stability of the service they consume.

That is where MLOps comes in. This nascent movement can be regarded as a superset of the DevOps movement: it aims to accelerate the delivery of reliable, working *ML* software on an ongoing basis. It therefore concerns itself with CI/CD pipelines in much the same way as DevOps, but also adds specific variations on these CI/CD problems. Most notably, the concept of Continuous Training (CT) is added to the mix. What is CT, you ask?

- **Continuous Training** — Many ML services require their underlying ML models to be retrained on some fixed basis, or on the occurence of specific events (e.g. data change, upstream model update). Additionally, it is important that these newly retrained models conform to th 68 umptions and behaviours that defined the previous version of the model in question. Continuous Training, then, is about establishing *robust, automated* processes for training and deploying models as a specialised variation of conventional Continuous Delivery pipelines, including the pre-processing, evaluation, selection and serving of ML models as part of broader software services.

The aim here is to allow ML practitioners to quickly and confidently deploy their latest and greatest models into production, while preserving the stability of the services that rely upon them.

Thinking bigger

risks of having organisationally isolated ML/technical capabilities. In many ways, this is convergent with the learnings of the broader software engineering community in recognising similar needs (and risks) for 'conventional' software applications.

> As with the broader software engineering ecosystem before it, the ML toolchain is becoming commoditised: the barriers to entry (cost, expertise) are gradually being lowered and the market is becoming more competitive.

This recognition has resulted in a burgeoning ecosystem around many of the problems outlined above. Each passing month sees the introduction of new tools and platforms explicitly targeted at ML/MLOps challenges. As with the broader software engineering ecosystem before it, the ML toolchain is becoming commoditised: the barriers to entry (cost, expertise) are gradually being lowered and the market is becoming more competitive. For those of you looking to get started with MLOps (or ML in general), this makes it a *very* exciting time to get involved: you can go a *long* way towards having 'production ready' ML pipelines for very little cost. Here's how.

Creating an ML pipeline

Right, it is about time for the tutorial!

The ML pipeline presented here is built using GitHub Actions — GitHub's Workflow automation tool. If you sign up for a GitHub account, you get access to this feature *for free*. There are usage and resource limits (as you might expect), but these are surprisingly generous as a free offering. If you can keep your models relatively lightweight, you could build a small product on top of the free offering alone, so for those of you with some cool MVP ideas, it could be a great place to start. Plus, if the free resources aren't cutting it for you, you can also create 'self hosted' actions too.

So what will the pipeline example presented here do? The pipeline is going to be a basic CT/CD pipeline built on top of the Serverless ML example discussed in a previous post. If you haven't read it yet, it's worth checking out:

Here's what your nice new pipeline will do:

1. Setup your environment and install dependencies.
2. Download the latest available version of a dataset.
3. Train and evaluate a new version of a model on the latest dataset.
4. Upload the new model and evaluation results.
5. Trigger the redeployment of your new model as an API if the previous steps succeed.

Additionally, you're going to see how you can *schedule* this workflow to run on a regular basis as a `cron` job (again, using GitHub Actions).

Before you begin

If you'd like to run the demo project, you'll need to register for a Google Cloud account. At the time of writing, you'll get $300 of credit added to your account. That'll *more* than cover the costs of running the code in this tutorial (which will be essentially free, anyway!). You can find out more here:

GCP Free Tier - Free Extended Trials and Always Free | Google Cloud
20+ free products Get free hands-on experience with popular products, including Compute Engine and Cloud Storage, up to…
cloud.google.com

As always, if you do sign up but don't wish to continue using Google Cloud, *make sure to cancel your account*!

Here's the template repository you'll be needing:

Open in app Get started

Next, you should either create a fork of the template repository into your personal GitHub account, or create a new repository from the template. With this done, you'll need to set up a couple of GitHub Secrets to allow your pipeline to access your Google Cloud services. You can find out how to set up secrets in the GitHub docs. Specifically, you'll need to add:

- `GCP_PROJECT_ID` - A unique Google Cloud project ID. You can find your project ID in the Google Cloud Console. Google provides a doc for finding this value if you're unsure.
- `GCP_SA_KEY` - A Service Account (SA) key. This is a secure key that gives access to specific cloud services in your project. In this example, you'll be using a default SA with elevated privileges for simplicity. In practice, you should look at creating an SA with reduced privileges. *You'll need to create a new key for your SA and copy it as a secret to GitHub.* Again, Google have a detailed doc on how to generate this key.

Note: As always with confidential/security info, don't share your SA key with anyone! Ideally, delete it when you're done with this example (unless you want to continue using it for your own projects of course).

With those set up, you'll need to enable your Google Cloud Build, Cloud Functions and Cloud Storage APIs. To do this, simply look through the left-hand navigation bar in the Google Cloud Console and select the relevant cloud services. If the relevant API is not activated, you'll be given the option to activate it when you've clicked on the service.

Finally, in the pipeline definition below, you'll need to define a *unique* bucket name in your Google Cloud Storage. The name used in the example pipeline definition (below) is `pipeline-example`. You should replace this with the name of your own bucket after forking but *before* trying to run the example. Additionally, you'll want to upload the `datasets/default.csv` dataset in the repository to `{your-bucket-name}/heart-disease/dataset.csv`.

Now, for good or ill, it is time for some YAML — The Language of the Cloud.

The pipeline is defined as a YAML file. You should check out GitHub's introduction to the workflow file format too, if you feel like being particularly thorough. If you're comfortable with YAML, this is probably easy enough to follow, but either way, here's how it breaks down:

0. Job setup & scheduling

This section of the file names the workflow `Train and Deploy` (this will be how GitHub displays your workflow in the GitHub UI), and then provides *event triggers* that will trigger your pipeline to run. In this case, you'll see that there are two triggers: `push` and `schedule`.

In the case of `push`, the pipeline will run every time you update your repository's `master` branch (the specific branches can be listed under the `branches` field). Practically, this means every time you merge a change to the code in the repository into `master` the pipeline will retrain the model and redeploy it. This can be useful for immediately propagating code changes to a live ML service.

For the `schedule` trigger, you simply set a `cron` schedule for your pipeline to run on. In other words: a fixed schedule on which your pipeline will run. The example value provided below will run the pipeline every weekday morning at 08:00. If you're not sure how to configure a `cron` schedule, here's a great interactive editor for you to play with.

```
name: Train and Deploy

on:
  push:
    branches:
      - master
  schedule:
    - cron:  '0 8 * * MON-FRI'
```

Feel free to play with other triggers. Additionally, you can write custom triggers too, if you're feeling fancy. These custom triggers could help you setup a 'true' event-driven architecture. Just a thought.

Now for the boring, but very important step. The key `train` defines the name of the step. Here you have a single job. Within this, you must define the virtual machine you're going to run your job on (`runs-on`) . In this case, the example is using `ubuntu-latest` . Next you define each of the `steps` within the job.

First, the `actions/checkout@v2` is run. This runs the GitHub-provided `checkout` action. This'll clone and checkout the default (typically `master`) branch of your repository.

Next, the job sets up the `gcloud` command line tool. This step uses an action provided by Google Cloud to make your life that little bit easier. This allows subsequent steps to access the `gcloud` and `gsutils` command line tools. You'll use these later too download/upload data from/to Google Cloud, and to redeploy your model API.

After this, you have two Python related steps. The first sets up a basic Python 3.7 environment. The second installs any dependencies you have in the toplevel `requirements.txt` file. And with that, your job is configured to run your pipeline proper. Now for the fun bits.

```
train:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2

    - name: Setup GCP client
      uses: GoogleCloudPlatform/github-actions/setup-gcloud@master
      with:
        version: '290.0.1'
        project_id: ${{ secrets.GCP_PROJECT_ID }}
        service_account_key: ${{ secrets.GCP_SA_KEY }}
        export_default_credentials: true

    - name: Set up Python
      uses: actions/setup-python@v2
      with:
        python-version: 3.7

    - name: Install Python dependencies
      run: |
        python -m pip install --upgrade pip
        pip install -r requirements.txt
```

Open in app Get started

2. Download dataset

First up, the job downloads the latest available dataset from the `pipeline-example` bucket. In practice, the pipeline would pick up the latest version of the dataset available on the provided path. This allows you to create an *independent* data pipeline to load and transform the dataset for the ML pipeline to pick up when it next runs.

```
- name: Download the latest dataset
      run: |
        gsutil cp gs://pipeline-example/heart-disease/dataset.csv
datasets/default.csv
```

You'll see that the step uses `gsutil` , Google's Cloud Storage command line utility. This lets you copy files to and from Google Cloud. Simple!

3. Train and evaluate model

Now the job has loaded the latest dataset, it is time to run the 'core' training task. In this case, this is identical to the example given in the previous Serverless ML tutorial.

```
- name: Run training task
      run: |
        python steps/train.py --path=datasets/default.csv
```

As a bonus, the `steps/train.py` script writes the following metadata and metrics to the `artifacts/metrics.json` path. As you'll see, this gets uploaded to Google Cloud too, so you can review how model performance (and training duration) changes over time. This can come in very handy!

```
metrics = dict(
    elapsed = end - start,
    acc = acc,
    val_acc = val_acc,
    roc_auc = roc_auc,
    timestamp = datetime.now().strftime(TIMESTAMP_FMT)
)
```

4. Upload model and metrics

The next step is to push your new model and metrics to Google Cloud Storage. You'll see that the pipeline uploads three files:

- `latest.joblib` - The 'latest' version of the model. This will be the most current 'valid' model the pipeline has produced.
- `${{ env.GITHUB_RUN_ID }}.joblib` - An archived version of the above model (identified by the unique GitHub run ID that produced it).
- `metrics/${{ env.GITHUB_RUN_ID }}.json` - An archived version of the above model's metrics (identified by the unique GitHub run ID that produced it). These can be ordered by the date they were created to produce a time series showing model performance over time.

```
- name: Upload new model and associated metrics
      run: |
        gsutil cp artifacts/pipeline.joblib gs://pipeline-
example/heart-disease/models/latest.joblib
        gsutil cp artifacts/pipeline.joblib gs://pipeline-
example/heart-disease/models/${{ env.GITHUB_RUN_ID }}.joblib
        gsutil cp artifacts/metrics.json gs://pipeline-
example/heart-disease/models/metrics/${{ env.GITHUB_RUN_ID }}.joblib
```

Now you have a new model, some metrics and all of it neatly archived in Google Cloud Storage.

5. Deploy Model as Cloud Function

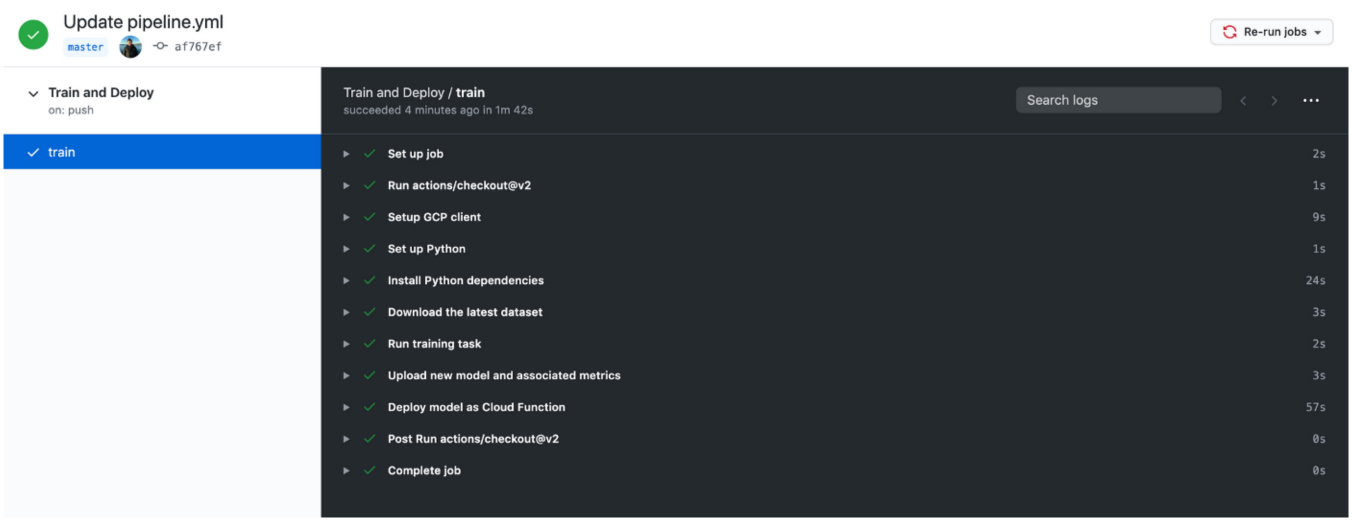
Finally, with all that done, it is time to redeploy your latest model.

```
- name: Deploy model as Cloud Function
      run: |
        gcloud functions deploy heart-disease --entry-
point=predict_handler --runtime=python37 --project=${{
secrets.GCP_PROJECT_ID }} --allow-unauthenticated --trigger-http
```

check out the previous post in this series. After a few moments, your new model will be rotated into service.

Finishing up

And that's it. If your pipeline completes successfully, you'll be able to see a nice long list of green ticks for each successful step.



```
Train and Deploy / train
succeeded 4 minutes ago in 1m 42s

▶ ✓ Set up job                                  2s
▶ ✓ Run actions/checkout@v2                     1s
▶ ✓ Setup GCP client                            9s
▶ ✓ Set up Python                               1s
▶ ✓ Install Python dependencies                24s
▶ ✓ Download the latest dataset                 3s
▶ ✓ Run training task                           2s
▶ ✓ Upload new model and associated metrics     3s
▶ ✓ Deploy model as Cloud Function             57s
▶ ✓ Post Run actions/checkout@v2                0s
▶ ✓ Complete job                                0s
```

You've now got an automated pipeline for loading new data, retraining a model, archiving the new model and its metrics and then redeploying your model API on a fixed schedule. Pretty cool, eh?

Next steps

This is only a baby ML pipeline. There's a lot things you could do to make it more sophisticated. Some initial ideas could be:

- Create a Cloud Function that is triggered when model metrics are uploaded to notify you (maybe via Slack or email) if model performance drops below a given value.
- Add a step to abort model redeployment if evaluation results are worse than previous model (or some test cases).
- Add a custom Cloud Function trigger to run the workflow when your dataset updates (rather than on a fixed basis, which may be unnecessary).

Open in app

Get started

Sign up for The Variable

By Towards Data Science

Every Thursday, the Variable delivers the very best of Towards Data Science: from hands-on tutorials and cutting-edge research to original features you don't want to miss. Take a look.

Get this newsletter

About Help Terms Privacy

Get the Medium app

Download on the App Store

GET IT ON Google Play