

Continuous Delivery for Machine Learning Models

By Alfredo Deza

Is it really the sad truth that natural philosophy (what we now call science) has so far separated off from its origins that it has left behind only papyrologists—people who take paper in, put paper out, and while reading and writing assiduously, earnestly avoid the tangible? Do they consider direct contact with data to be of negative value? Are they, like some redneck in the novel *Tobacco Road*, actually proud of their ignorance?

—Dr. Joseph Bogen

As a professional athlete, I was often dealing with injuries. Injuries have all kinds of severity levels. Sometimes it would be something minor, like a mild contracture on my left hamstring after intense hurdle workouts. Other times it would be more serious, like insufferable lower back pain. High-performance athletes cannot afford to have days off in the middle of the season. If the plan is to work out seven days a week, it is critical to go through those seven days. Missing out a day has serious repercussions that can diminish (or entirely wipe out) the workouts until that point. Workouts are like pushing a wheelbarrow uphill, and missing a workout means stepping to the side letting the wheelbarrow ride downhill. The repercussion of that action is that you will have to go back and pick up that wheelbarrow to push it up again. You cannot miss out on workouts.

If you are injured and you cannot work out, getting back up in full shape as soon as possible is a priority *as important as finding alternative workouts*. That means that if your hamstring hurts and you can't run, see if you can go to the pool and keep up the cardio plan. Hill repeats are not possible tomorrow because you broke a toe? Then try hopping on the bike to tackle those same hills. Injuries require a war strategy; giving up and quitting is not an option, but if you must retreat, then retreating *the least*

possible is considered first. If we cannot fire cannons, let's bring the cavalry. There is always an option, and creativity is as important as trying to recover fully.

Recovery requires strategy as well, but more than strategy, it requires constant evaluation. Since you keep working out as much as possible with an injury, it is essential to evaluate if the injury is getting worse. If you get on the bike to compensate because you can't run, you must be hyper-aware if the bike is making the injury worse. The constant evaluation for injuries is a rather simplistic algorithm:

1. First thing every day, assess if the injury is the same, worse, or better than the day before.
2. If it is worse, then make changes to avoid the previous workouts or alter them. Those may be harming recovery.
3. If it is the same, compare the injury against last week or last month even. Ask the question *"Am I feeling the same, worse, or better than last week?"*
4. Finally, if you feel better, that strongly reinforces that the current strategy is working, and you should continue until fully recovered.

With some injuries, I had to evaluate at a higher frequency (rather than waiting until the next morning). The result of constant evaluation was the key to recovery. In some cases, I had to evaluate if a specific action was hurting me. One time I broke a toe (slammed it into the corner of a bookshelf), and I immediately strategized: can I walk? Do I feel pain if I run? The answer to all of these was a resounding yes. I tried going swimming that afternoon. For the next few weeks, I would constantly check if walking was possible without pain. Pain is not a foe. It is the indicator that helps you decide to keep doing what you are doing or stop and rethink the current strategy.

Constant evaluation, making changes and adapting to the feedback, and applying new strategies to achieve success is exactly what continuous integration (CI) and continuous delivery (CD) are about. Even today, where information about robust deployment strategies is easily available, you often encounter businesses without tests or a poor testing strategy to ensure a product is ready for a new release or even releases that take weeks (and months!). I recall trying to cut a new release of a major open source project, and there were times it would take close to a week. Even worse, the Quality Assurance (QA) lead would send emails to every team lead and ask them if they felt ready for a release or wanted more changes.

Sending emails around and waiting for different replies is not a straightforward way to release software. It is prone to error and highly inconsistent. The feedback loop that CI/CD platforms and steps grant you and your team is invaluable. If you find a problem, you must automate it away and make it not-a-problem for the next release. Constant evaluation, just like injuries with high-performing athletes, is a core pillar of DevOps and absolutely critical for successful machine learning operationalization.

I like the description of continuous as persistence or recurrence of a process. CI/CD are usually mentioned together when talking about the system that builds, verifies, and deploys artifacts. In this chapter, I will detail what a robust process looks like and how you can enable various strategies to implement (or improve) a pipeline to ship models into production.

Packaging for ML Models

It wasn't that long ago I heard about packaging ML models for the first time. If you've never heard about packaging models before, it's OK—this is all fairly recent, and *packaging* here doesn't mean some special type of operating system package like an RPM (Red Hat Package Manager) or DEB (Debian Package) file with special directives for bundling and distribution. This all means getting a model into a container to take advantage of containerized processes to help sharing, distributing, and easy deployment. I've already described containerization in detail in “Containers” on page 68 and why it makes sense to use them for operationalizing machine learning versus using other strategies like virtual machines, but it is worth reiterating that the ability to quickly try out a model from a container regardless of the operating system is a dream scenario come true.

There are three characteristics of packaging ML models into containers that are significant to go over:

- As long as a container runtime is installed, it is effortless to run a container locally.
- There are plenty of options to deploy a container in the cloud, with the ability to scale up or down as needed.
- Others can quickly try it out with ease and interact with the container.

The benefits of these characteristics are that maintainability becomes less complicated, and debugging a nonperformant model locally (or in a cloud offering even) can be as simple as a few commands in a terminal. The more complicated the deployment strategy is, the more difficult it will be to troubleshoot and investigate potential issues.

For this section, I will use an ONNX model and package it within a container that serves a Flask app that performs the prediction. I will use the **RoBERTa-SequenceClassification** ONNX model, which is very well documented. After creating a new Git repository, the first step is to figure out the dependencies needed. After creating the Git repository, start by adding the following *requirements.txt* file:

```
simpletransformers==0.4.0
tensorboardX==1.9
transformers==2.1.0
```

```
flask==1.1.2
torch==1.7.1
onnxruntime==1.6.0
```

Next, create a Dockerfile that installs everything in the container:

```
FROM python:3.8

COPY ./requirements.txt /webapp/requirements.txt

WORKDIR /webapp

RUN pip install -r requirements.txt

COPY webapp/* /webapp

ENTRYPOINT [ "python" ]

CMD [ "app.py" ]
```

The Dockerfile copies the requirements file, creates a *webapp* directory, and copies the application code into a single *app.py* file. Create the *webapp/app.py* file to perform the sentiment analysis. Start by adding the imports and everything needed to create an ONNX runtime session:

```
from flask import Flask, request, jsonify
import torch
import numpy as np
from transformers import RobertaTokenizer
import onnxruntime

app = Flask(__name__)
tokenizer = RobertaTokenizer.from_pretrained("roberta-base")
session = onnxruntime.InferenceSession(
    "roberta-sequence-classification-9.onnx")
```

This first part of the file creates the Flask application, defines the tokenizer to use with the model, and finally, it initializes an ONNX runtime session that requires passing a path to the model. There are quite a few imports that aren't used yet. You will make use of those next when adding the Flask route to enable the live inferencing:

```
@app.route("/predict", methods=["POST"])
def predict():
    input_ids = torch.tensor(
        tokenizer.encode(request.json[0], add_special_tokens=True)
    ).unsqueeze(0)

    if input_ids.requires_grad:
        numpy_func = input_ids.detach().cpu().numpy()
    else:
        numpy_func = input_ids.cpu().numpy()
```

```

inputs = {session.get_inputs()[0].name: numpy_func(input_ids)}
out = session.run(None, inputs)

result = np.argmax(out)

return jsonify({"positive": bool(result)})

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000, debug=True)

```

The `predict()` function is a Flask route that enables the `/predict` URL when the application is running. The function only allows POST HTTP methods. There is no description of the sample inputs and outputs yet because one critical part of the application is missing: the ONNX model does not exist yet. Download the **RoBERTa-SequenceClassification** ONNX model locally, and place it at the root of the project. This is how the final project structure should look:

```

.
├── Dockerfile
├── requirements.txt
├── roberta-sequence-classification-9.onnx
└── webapp
    └── app.py

```

1 directory, 4 files

One last thing missing before building the container is that there is no instruction to copy the model into the container. The `app.py` file requires the model `roberta-sequence-classification-9.onnx` to exist in the `/webapp` directory. Update the `Dockerfile` to reflect that:

```
COPY roberta-sequence-classification-9.onnx /webapp
```

Now the project has everything needed so you can build the container and run the application. Before building the container, let's double-check everything works. Create a new virtual environment, activate it, and install all the dependencies:

```

$ python3 -m venv venv
$ source venv/bin/activate
$ pip install -r requirements.txt

```

The ONNX model exists at the root of the project, but the application wants it in the `/webapp` directory, so move it inside that directory so that the Flask app doesn't complain (this extra step is not needed when the container runs):

```
$ mv roberta-sequence-classification-9.onnx webapp/
```

Now run the application locally by invoking the `app.py` file with Python:

```

$ cd webapp
$ python app.py
* Serving Flask app "app" (lazy loading)

```

```

* Environment: production
  WARNING: This is a development server.
  Use a production WSGI server instead.
* Debug mode: on
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)

```

Next, the application is ready to consume HTTP requests. So far, I've not shown what the expected inputs are. These are going to be JSON-formatted requests with JSON responses. Use the *curl* program to send a sample payload to detect sentiment:

```

$ curl -X POST -H "Content-Type: application/JSON" \
  --data '["Containers are more or less interesting"]' \
  http://0.0.0.0:5000/predict

{
  "positive": false
}

$ curl -X POST -H "Content-Type: application/json" \
  --data '["MLOps is critical for robustness"]' \
  http://0.0.0.0:5000/predict

{
  "positive": true
}

```

The JSON request is an array with a single string, and the response is a JSON object with a “positive” key that indicates the sentiment of the sentence. Now that you’ve verified that the application runs and that the live prediction is functioning properly, it is time to create the container locally to verify all works there. Create the container, and tag it with something meaningful:

```

$ docker build -t alfredodeza/roberta .
[+] Building 185.3s (11/11) FINISHED
=> [internal] load metadata for docker.io/library/python:3.8
=> CACHED [1/6] FROM docker.io/library/python:3.8
=> [2/6] COPY ./requirements.txt /webapp/requirements.txt
=> [3/6] WORKDIR /webapp
=> [4/6] RUN pip install -r requirements.txt
=> [5/6] COPY webapp/* /webapp
=> [6/6] COPY roberta-sequence-classification-9.onnx /webapp
=> exporting to image
=> => naming to docker.io/alfredodeza/roberta

```

Now run the container locally to interact with it in the same way as when running the application directly with Python. Remember to map the ports of the container to the localhost:

```

$ docker run -it -p 5000:5000 --rm alfredodeza/roberta
* Serving Flask app "app" (lazy loading)
* Environment: production
  WARNING: This is a development server.

```

```
    Use a production WSGI server instead.
* Debug mode: on
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Send an HTTP request in the same way as before. You can use the *curl* program again:

```
$ curl -X POST -H "Content-Type: application/json" \
  --data '["espresso is too strong"]' \
  http://0.0.0.0:5000/predict

{
  "positive": false
}
```

We've gone through many steps to package a model and get it inside a container. Some of these steps might seem overwhelming, but challenging processes are a perfect opportunity to automate and leverage continuous delivery patterns. In the next section, I'll automate all of this using continuous delivery and publishing this container to a container registry that anyone can consume.

Infrastructure as Code for Continuous Delivery of ML Models

Recently at work, I saw that a few test container images existed in a public repository, which were widely used by the test infrastructure. Having images hosted in a container registry (like Docker Hub) is already a great step in the right direction for repeatable builds and reliable tests. I encountered a problem with one of the libraries used in a container that needed an update, so I searched for the files used to create these test containers. They were nowhere to be found. At some point, an engineer built these locally and uploaded the images to the registry. This presented a big problem because I couldn't make a simple change to the image since the files needed to build the image were lost.

Experienced container developers can find a way to get most (if not all) files to rebuild the container, but that is beside the point. A step forward in this problematic situation is to create automation that can automatically build these containers from known source files, including the *Dockerfile*. Rebuilding or solving the problem to update the container and re-upload to the registry is like finding candles and flashlights in a blackout, instead of having a generator that starts automatically as soon as the power goes away. Be highly analytical when situations like the one I just described happens. Rather than pointing fingers and blaming others, use these as an opportunity to enhance the process with automation.

The same problem happens in machine learning. We tend to grow easily accustomed to things being manual (and complex!), but there is always an opportunity to automate. This section will not go over all the steps needed in containerization again (already covered in “Containers” on page 68), but I will go into the details needed to automate everything. Let’s assume we’re in a similar situation to the one I just described and that someone has created a container with a model that lives in Docker Hub. Nobody knows how the trained model got into the container; there are no docs, and updates are needed. Let’s add a slight complexity: the model is not in any repository to be found, but it lives in Azure as a registered model. Let’s get some automation going to solve this problem.



It might be tempting to add models into a GitHub repository. Although this is certainly possible, GitHub has (at the time of this writing) a hard file limit of 100 MB. If the model you are trying to package is close to that size, you might not be able to add it to the repository. Further, Git (the version control system) is not meant to handle versioning of binary files and has the side-effect of creating huge repositories because of this.

In the current problem scenario, the model is available on the Azure ML platform and previously registered. I didn’t have one already, so I quickly registered **RoBERTa-SequenceClassification** using Azure ML Studio. Click the Models section and then “Register model” as shown in **Figure 4-1**.

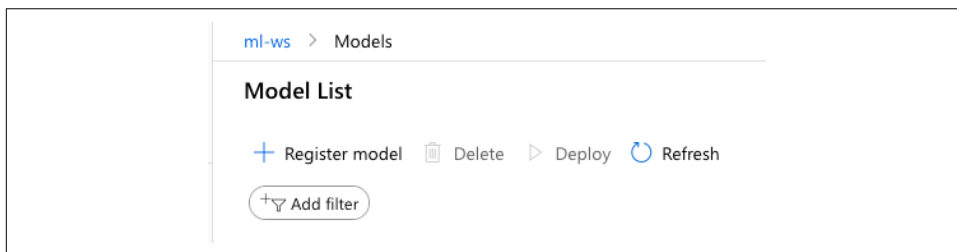


Figure 4-1. Azure model registering menu

Fill out the form shown in **Figure 4-2** with the necessary details. In my case, I downloaded the model locally and need to upload it using the “Upload file” field.

Register a model

×

Name *

roberta-sequence

👁

Description

Sentiment prediction using RoBERTa-sequencing ONNX model

Model framework *

ONNX

⌵

*

Framework version *

1.6

Model file or folder *

☒ Upload file
☐ Upload folder

roberta-sequence-classification-9.onnx

*

Browse

Add tags

Name

:

Value

Add tag

Add properties

Name

:

Value

Add property

Figure 4-2. Azure model registering form



If you want to know more about registering a model in Azure, I cover how to do that with the Python SDK in [“Registering Models” on page 243](#).

Now that the pretrained model is in Azure let’s reuse the same project from [“Packaging for ML Models” on page 95](#). All the heavy lifting to perform the (local) live inferencing is done, so create a new GitHub repository and add the project contents *except* for the ONNX model. Remember, there is a size limit for files in GitHub, so it isn’t possible to add the ONNX model into the GitHub repo. Create a `.gitignore` file to ignore the model and prevent adding it by mistake:

```
*.onnx
```

After pushing the contents of the Git repository without the ONNX model, we are ready to start automating the model creation and delivery. To do this, we will use GitHub Actions, which allows us to create a continuous delivery workflow in a YAML file that gets triggered when configurable conditions are met. The idea is that whenever the repository has a change in the main branch, the platform will pull the registered model from Azure, create the container, and lastly, it will push it to a container registry. Start by creating a `.github/workflows/` directory at the root of your project, and then add a `main.yml` that looks like this:

```
name: Build and package RoBERTa-sequencing to Dockerhub

on:
  # Triggers the workflow on push or pull request events for the main branch
  push:
    branches: [ main ]

  # Allows you to run this workflow manually from the Actions tab
  workflow_dispatch:
```

The configuration so far doesn't do anything other than defining the action. You can define any number of jobs, and in this case, we define a `build` job that will put everything together. Append the following to the `main.yml` file you previously created:

```
jobs:
  build:
    runs-on: ubuntu-latest
    steps:

    - uses: actions/checkout@v2

    - name: Authenticate with Azure
      uses: azure/login@v1
      with:
        creds: ${secrets.AZURE_CREDENTIALS}

    - name: set auto-install of extensions
      run: az config set extension.use_dynamic_install=yes_without_prompt

    - name: attach workspace
      run: az ml folder attach -w "ml-ws" -g "practical-mlops"

    - name: retrieve the model
      run: az ml model download -t "." --model-id "roberta-sequence:1"

    - name: build flask-app container
      uses: docker/build-push-action@v2
      with:
        context: ./
        file: ./Dockerfile
        push: false
        tags: alfredodeza/flask-roberta:latest
```

The build job has many steps. In this case, each step has a distinct task, which is an excellent way to separate failure domains. If everything were in a single script, it would be more difficult to grasp potential issues. The first step is to check out the repository when the action triggers. Next, since the ONNX model doesn't exist locally, we need to retrieve it from Azure, so we must authenticate using the Azure action. After authentication, the *az* tool is made available, and you must attach the folder for your workspace and group. Finally, the job can retrieve the model by its ID.



Some steps in the YAML file have a `uses` directive, which identifies what external action (for example `actions/checkout`) and at what version. Versions can be branches or published tags of a repository. In the case of `checkout` it is the `v2` tag.

Once all those steps complete, the RoBERTa-Sequence model should be at the root of the project, enabling the next steps to build the container properly.

The workflow file is using `AZURE_CREDENTIALS`. These are used with a special syntax that allows the workflow to retrieve secrets configured for the repository. These credentials are the service principal information. If you aren't familiar with a service principal, this is covered in the [“Authentication” on page 238](#). You will need the service principal's configuration that has access to the resources in the workspace and group where the model lives. Add the secret on your GitHub repository by going to Settings, then Secrets, and finally clicking the “New repository secret” link. [Figure 4-3](#) shows the form you will be presented when adding a new secret.

A screenshot of the GitHub 'Actions secrets / New secret' form. The form has a title bar 'Actions secrets / New secret'. Below it, there is a 'Name' field with the text 'AZURE_CREDENTIALS'. Below that is a 'Value' field containing a JSON object. The JSON object has four keys: 'clientId', 'sqlManagementEndpointUrl', 'galleryEndpointUrl', and 'managementEndpointUrl'. The values are: 'xxxxxxxx-3af0-4065-8e14-xxxxxxxxxxxxx', 'https://management.core.windows.net:8443/', 'https://gallery.azure.com/', and 'https://management.core.windows.net/' respectively. At the bottom of the form is a green button labeled 'Add secret'.

Figure 4-3. Add secret

Commit and push your changes to your repository and then head to the Actions tab. A new run is immediately scheduled and should start running in a few seconds. After a few minutes, everything should've completed. In my case, [Figure 4-4](#) shows it takes close to four minutes.

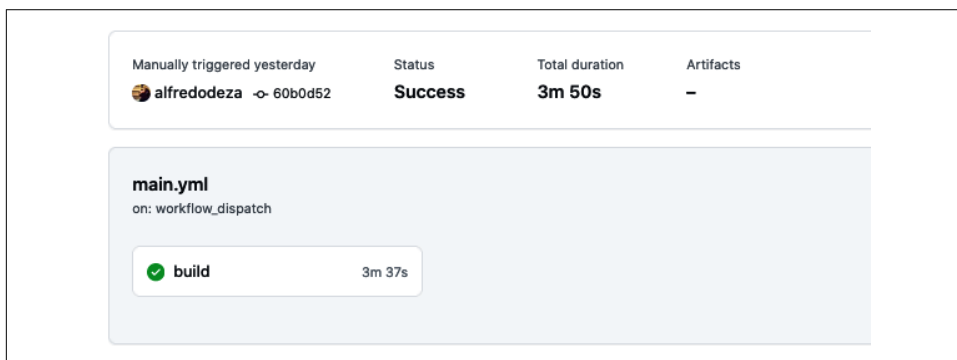


Figure 4-4. *GitHub action success*

There are now quite a few moving parts to accomplish a successful job run. When designing a new set of steps (or pipelines, as I'll cover in the next section), a good idea is to enumerate the steps and identify greedy steps. These *greedy steps* are steps that are trying to do too much and have lots of responsibility. At first glance, it is hard to identify any step that might be problematic. The process of maintaining a CI/CD job includes refining responsibilities of steps and adapting them accordingly.

Once the steps are identified, you can break them down into smaller steps, which will help you understand the responsibility of each part faster. A faster understanding means easier debugging, and although it's not immediately apparent, you will benefit from making this a habit.

These are the steps we have for packaging the RoBERTa-Sequence model:

1. Check out the current branch of the repository.
2. Authenticate to Azure Cloud.
3. Configure auto-install of Azure CLI extensions.
4. Attach the folder to interact with the workspace.
5. Download the ONNX model.
6. Build the container for the current repo.

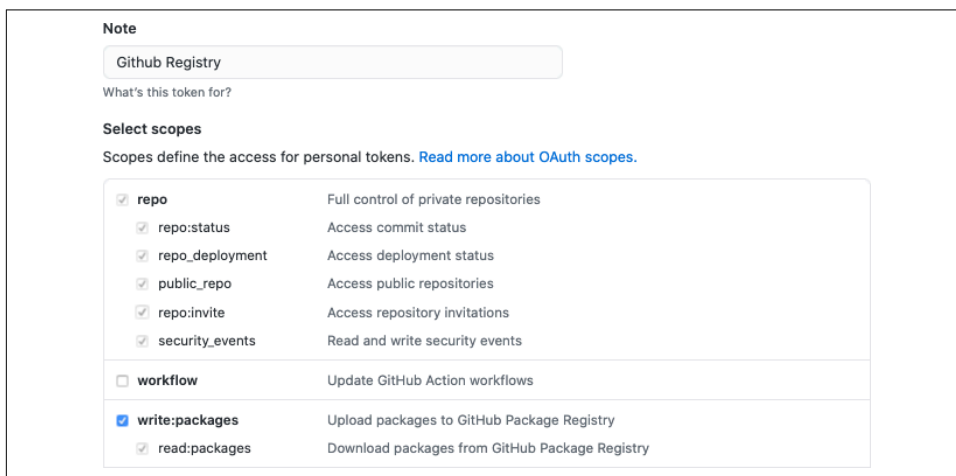
There is one final item missing, though, and that is to publish the container after it builds. Different container registries will require different options here, but most do support GitHub Actions, which is refreshing. Docker Hub is straightforward, and all it requires is to create a token and then save it as a GitHub project secret, along with

your Docker Hub username. Once that is in place, adapt the workflow file to include the authentication step before building:

```
- name: Authenticate to Docker hub
  uses: docker/login-action@v1
  with:
    username: ${ secrets.DOCKER_HUB_USERNAME }
    password: ${ secrets.DOCKER_HUB_ACCESS_TOKEN }
```

Lastly, update the build step to use `push: true`.

Recently, GitHub has released a container registry offering as well, and its integration with GitHub Actions is straightforward. The same Docker steps can be used with minor changes and creating a PAT (Personal Access Token). Start by creating a PAT by going to your GitHub account settings, clicking Developer Settings, and finally “Personal access” tokens. Once that page loads, click “Generate new token.” Give it a meaningful description in the Note section, and ensure that the token has permissions to write packages as I do in [Figure 4-5](#).



Note

Github Registry

What's this token for?

Select scopes

Scopes define the access for personal tokens. [Read more about OAuth scopes.](#)

<input checked="" type="checkbox"/> repo	Full control of private repositories
<input checked="" type="checkbox"/> repo:status	Access commit status
<input checked="" type="checkbox"/> repo_deployment	Access deployment status
<input checked="" type="checkbox"/> public_repo	Access public repositories
<input checked="" type="checkbox"/> repo:invite	Access repository invitations
<input checked="" type="checkbox"/> security_events	Read and write security events
<input type="checkbox"/> workflow	Update GitHub Action workflows
<input checked="" type="checkbox"/> write:packages	Upload packages to GitHub Package Registry
<input checked="" type="checkbox"/> read:packages	Download packages from GitHub Package Registry

Figure 4-5. GitHub Personal Access Token

Once you are done, a new page is presented with the actual token. This is the only time you will see the token in plain text, so make sure you copy it now. Next, go to the repository where the container code lives, and create a new repository secret, just like you did with the Azure service principal credentials. Name the new secret `GH_REGISTRY` and paste the contents of the PAT created in the previous step. Now you are ready to update the Docker steps to publish the package using the new token and GitHub’s container registry:

```
- name: Login to GitHub Container Registry
  uses: docker/login-action@v1
  with:
```

```

registry: ghcr.io
username: ${ github.repository_owner }}
password: ${ secrets.GH_REGISTRY }}

- name: build flask-app and push to registry
  uses: docker/build-push-action@v2
  with:
    context: ./
    tags: ghcr.io/alfredodeza/flask-roberta:latest
    push: true

```

In my case, *alfredodeza* is my GitHub account, so I can tag with it along with the *flask-roberta* name of the repository. These will need to match according to your account and repository. After pushing the changes to the main branch (or after merging if you made a pull request), the job will trigger. The model should get pulled in from Azure, packaged within the container, and finally published as a GitHub Package in its container registry offering, looking similar to [Figure 4-6](#).

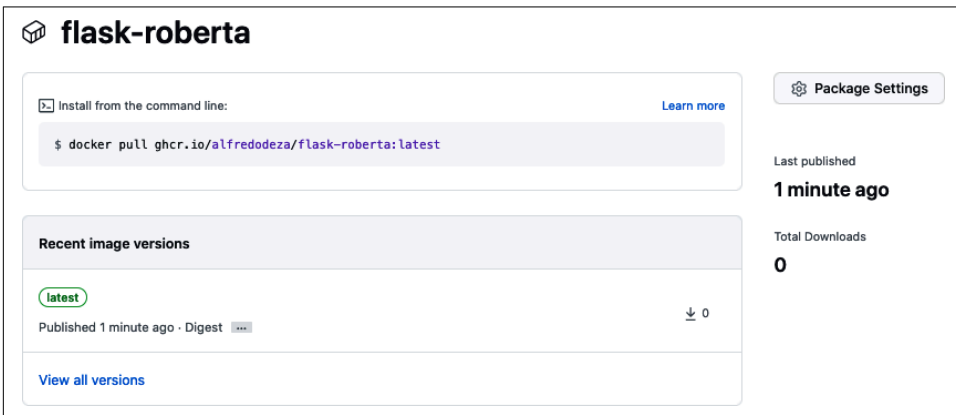


Figure 4-6. GitHub Package container

Now that the container is packaging and distributing the ONNX model in a fully automated fashion by leveraging GitHub's CI/CD offering and container registry, we have solved the problematic scenario I assumed at the beginning of the chapter: a model needs to get packaged in a container, but the container files are not available. In this way, you are providing clarity to others and to the process itself. It is segmented into small steps, and it allows any updates to be done to the container. Finally, the steps publish the container to a selected registry.

You can accomplish quite a few other things with CI/CD environments besides packaging and publishing a container. CI/CD platforms are the foundation for automation and reliable results. In the next section, I go into other ideas that work well regardless of the platform. By being aware of general patterns available in other platforms, you can take advantage of those features without worrying about the implementations.

Using Cloud Pipelines

The first time I heard about pipelines, I thought of them as more advanced than the typical scripting pattern (a procedural set of instructions representing a build). But pipelines aren't advanced concepts at all. If you've dealt with shell scripts in any continuous integration platform, then a pipeline will seem straightforward to use. A pipeline is nothing more than a set of steps (or instructions) that can achieve a specific objective like publishing a model into a production environment when run. For example, a pipeline with three steps to train a model can be as simple as [Figure 4-7](#).

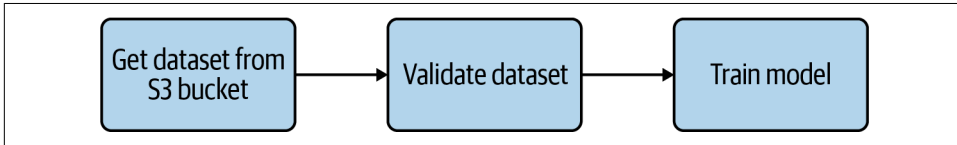


Figure 4-7. Simple pipeline

You could represent the same pipeline as a shell script that does all three things at once. There are multiple benefits with a pipeline that separates concerns. When each step has a specific responsibility (or concern), it is easier to grasp. If a single-step pipeline that retrieves the data, validates it, and trains the model is failing, it isn't immediately clear why that might fail. Indeed you can dive into the details, look at logs, and check the actual error. If you separate the pipeline into three steps and the *train model* step is failing, you can narrow the failure's scope and get to a possible resolution faster.



One general recommendation that you can apply to the many aspects of operationalizing machine learning is to consider making any operation more straightforward for a future failure situation. Avoid being tempted to go fast and get a pipeline (like in this case) deployed and running in a single step because it is easier. Take the time to reason about what would make it easier for you (and others) to build ML infrastructure. When a failure does happen, and you identify problematic aspects, go back to the implementation and improve it. You can apply the concepts of CI/CD to improvement: continuous evaluation and improvement of processes is a sound strategy for robust environments.

Cloud pipelines are no different from any continuous integration platform out there except that they are hosted or managed by a cloud provider.

Some definitions of CI/CD pipelines you can encounter try to define elements or parts of a pipeline rigidly. In reality, I think that the parts of the pipeline should be loosely defined and not constrained by definitions. RedHat [has a nice explanation of](#)

pipelines that describes five common elements: build, test, release, deploy, and validate. These elements are mostly for mix-and-match, not to strictly include them in the pipeline. For example, if the model you are building doesn't need to get deployed, then there is no need to pursue a deploy step at all. Similarly, if your workflow requires extracting and preprocessing data, you need to implement it as another step.

Now that you are aware that a pipeline is basically the same as a CI/CD platform with several steps, it should be straightforward to apply machine learning operations to an actionable pipeline. **Figure 4-8** shows a rather simplistic assumed pipeline, but this can involve several other steps as well, and like I've mentioned, these elements can be mixed and matched together to any number of operations and steps.

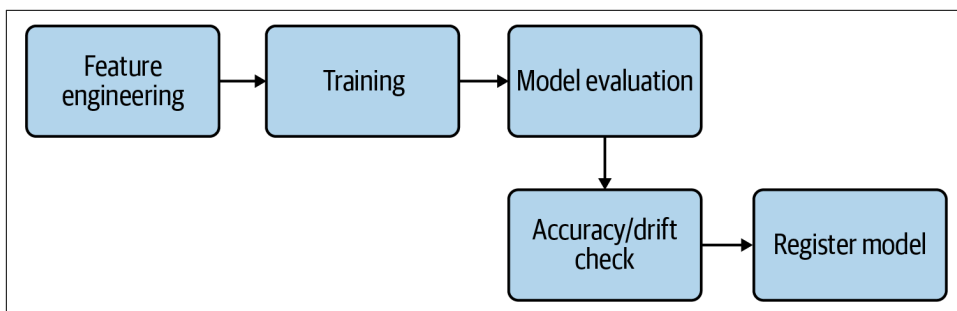


Figure 4-8. Involved pipeline

AWS SageMaker does an outstanding job of providing examples that are ready to use out of the box for crafting involved pipelines that include everything you need to run several steps. SageMaker is a specialized machine learning platform that goes beyond offering steps in a pipeline to accomplish a goal like publishing a model. Since it is specialized for machine learning, you are exposed to features that are particularly important for getting models into production. Those features don't exist in other common platforms like GitHub Actions, or if they do, they aren't as well thought out because the primary goal of platforms like GitHub Actions or Jenkins isn't to train machine learning models but rather be as generic as possible to accommodate for most common use cases.

Another crucial problem that is somewhat hard to solve is that specialized machines for training (for example, GPU-intensive tasks) are just not available or hard to configure in a generic pipeline offering.

Open SageMaker Studio and head over to the Components and Registries section on the left sidebar and select Projects. Several SageMaker project templates show up to choose from, as shown in **Figure 4-9**.

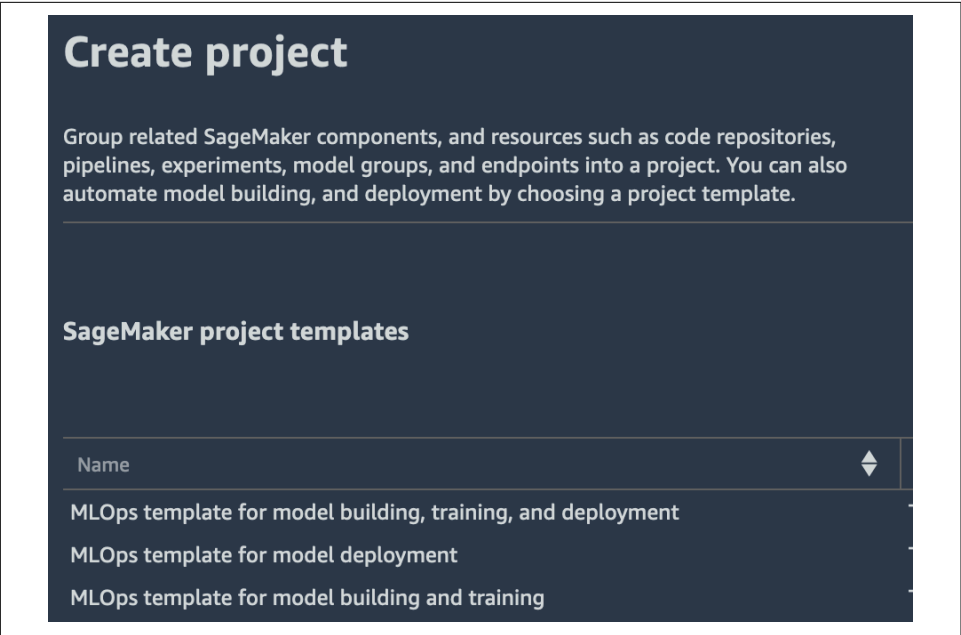


Figure 4-9. SageMaker templates



Although the examples are meant to get you started, and Jupyter Notebooks are provided, they are great for learning more about the steps involved and how to change and adapt them to your specific needs. After creating a pipeline instance in SageMaker, training, and finally registering the model, you can browse through the parameters for the pipeline, like in [Figure 4-10](#).

Executions			Graph	Parameters	Settings
Parameters		Type	Value		
ProcessingInstanceType		String	mLm5.xlarge		
ProcessingInstanceCount		Integer	1		
TrainingInstanceType		String	mLm5.xlarge		
ModelApprovalStatus		String	PendingManualApproval		
InputDataUrl		String	s3://sagemaker-servicecatalog-seedcode-us-east-2/d...		

Figure 4-10. Pipeline parameters

Another crucial part of the pipeline that shows all the steps involved is also available, as shown in [Figure 4-11](#).

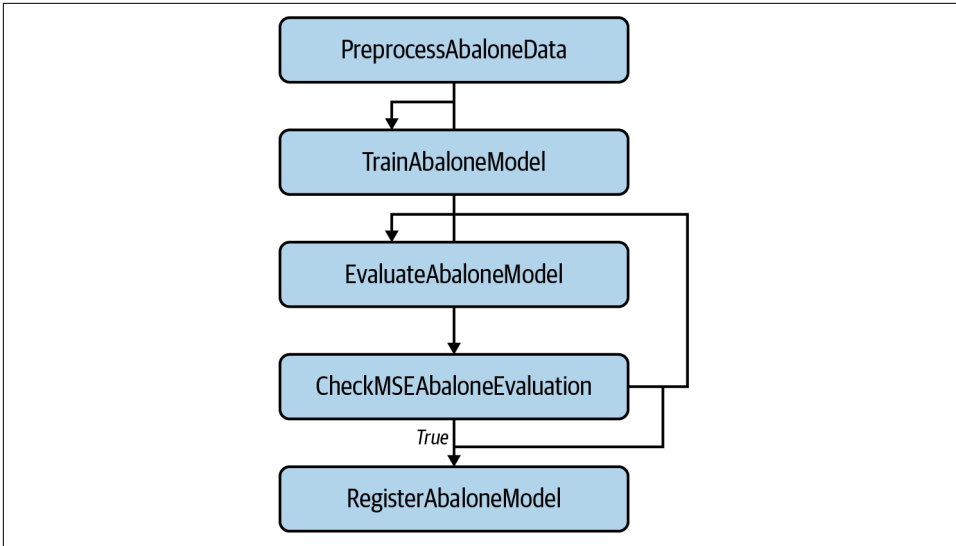


Figure 4-11. SageMaker pipeline

As you can see, preparing data, training, evaluating, and registering a model are all part of the pipeline. The main objective is to register the model to deploy it later for live inferencing after packaging. Not all the steps need to be captured in this particular pipeline, either. You can craft other pipelines that can run whenever there is a newly registered model available. That way, that pipeline is not tied to a particular model, but rather, you can reuse it for any other model that gets trained successfully and registered. Reusability of components and automation is another critical component of DevOps that works well when applied to MLOps.

Now that pipelines are demystified, we can see certain enhancements that can make them more robust by manually controlling rolling out models or even switching inferencing from one model to another.

Controlled Rollout of Models

There are a few concepts from web service deployments that map nicely into strategies for deploying models into production environments, like creating several instances of a live inferencing application for scalability and progressively switching from an older to a newer model. Before going into some of the details that encompass the control part of rolling out models into production, it is worth describing the strategies where these concepts come into play.

I'll discuss two of these strategies in detail in this section. Although these strategies are similar, they have particular behavior that you can take advantage of when deploying:

- Blue-green deployment
- Canary deployment

A blue-green deployment is a strategy that gets a new version into the staging environment identical to production. Sometimes this staging environment is the same as production, but traffic is routed differently (or separately). Without going into details, Kubernetes is a platform that allows for this type of deployment with ease, since you can have the two versions in the same Kubernetes cluster, but routing the traffic to a separate address for the newer (“blue”) version while production traffic is still going into the older (“green”). The reason for this separation is that it allows further testing and assurance that the new model is working as expected. Once this verification is complete and certain conditions are satisfactory, you modify the configuration to switch traffic from the current model to the new one.

There are some issues with blue-green deployments, primarily associated with how complicated it can be to replicate production environments. Again, this is one of those situations where Kubernetes is a perfect fit since the cluster can accommodate the same application with different versions with ease.

A canary deployment strategy is a bit more involved and somewhat riskier. Depending on your level of confidence and the ability to progressively change configuration based on constraints, it is a sound way to send models into production. In this case, traffic is routed progressively to the newer model *at the same time the previous model is serving predictions*. So the two versions are live and processing requests simultaneously, but doing them in different ratios. The reason for this percentage-based rollout is that you can enable metrics and other checks to capture problems in real time, allowing you to roll back immediately if conditions are unfavorable.

For example, assume that a new model with better accuracy and no noted drift is ready to get into production. After several instances of this new version are available to start receiving traffic, make a configuration change to send 10% of all traffic to the new version. While traffic starts to get routed, you notice a dismal amount of errors from responses. The HTTP 500 errors indicate that the application has an internal error. After some investigation, it shows that one of the Python dependencies that do the inferencing is trying to import a module that has been moved, causing an exception. If the application receives one hundred requests per minute, only ten of those would’ve experienced the error condition. After noticing the errors, you quickly change the configuration to send all traffic to the older version currently deployed. This operation is also referred to as a *rollback*.

Most cloud providers have the ability to do a controlled rollout of models for these strategies. Although this is not a fully functional example, the Azure Python SDK can define the percentage of traffic for a newer version when deploying:

```
from azureml.core.webservice import AksEndpoint

endpoint.create_version(version_name = "2",
                        inference_config=inference_config,
                        models=[model],
                        traffic_percentile = 10)
endpoint.wait_for_deployment(True)
```

The tricky part is that a canary deployment's objective is to progressively increase until the `traffic_percentile` is at 100%. The increase has to happen alongside meeting constraints about application healthiness and minimal (or zero) error rates.

Monitoring, logging, and detailed metrics of production models (aside from model performance) are absolutely critical for a robust deployment strategy. I consider them crucial for deployment, but they are a core pillar of the robust DevOps practices covered in [Chapter 6](#). Besides monitoring, logging, and metrics that have their own chapter, there are other interesting things to check for continuous delivery. In the next section, we will see a few that make sense and increase the confidence of deploying a model into production.

Testing Techniques for Model Deployment

So far, the container built in this chapter works great and does exactly what we need: from some HTTP requests with a carefully crafted message in a JSON body, a JSON response predicts the sentiment. A seasoned machine learning engineer might have put accuracy and drift detection (covered in detail in [Chapter 6](#)) in place before getting to the model packaging stage. Let's assume that's already the case and concentrate on other helpful tests you can perform before deploying a model into production.

When you send an HTTP request to the container to produce a prediction, several software layers need to go through from start to end. At a high level, these are critical:

1. Client sends an HTTP request, with a JSON body, in the form of an array with a single string.
2. A specific HTTP PORT (5000) and endpoint (*predict*) have to exist and get routed to.
3. The Python Flask application has to receive the JSON payload and load it into native Python.
4. The ONNX runtime needs to consume the string and produce a prediction.
5. A JSON response with an HTTP 200 response needs to contain the boolean value of the prediction.

Every single one of these high-level steps can (and should) be tested.

Automated checks

While putting together the container for this chapter, I got into some problems with the `onnxruntime` Python module: the documentation doesn't pin (an exact version number) the version, which caused the latest version to get installed, which needed different arguments as input. The accuracy of the model was good, and I was not able to detect a significant drift. And yet, I deployed the model only to find it fully broken once requests were consumed.

With time, applications become better and more resilient. Another engineer might add error handling to respond with an error message when invalid inputs get detected, and perhaps with an HTTP response with an appropriate HTTP error code along with a nice error message that the client can understand. You must test out these types of additions and behaviors before allowing a model to ship into production.

Sometimes there will be no HTTP error condition and no Python tracebacks either. What would happen if I made a change like the following to the JSON response:

```
{
  "positive": "false"
}
```

Without looking back at the previous sections, can you tell the difference? The change would go unnoticed. The canary deployment strategy would go to 100% without any errors detected. The machine learning engineer would be happy with high accuracy and no drift. And yet, this change has completely broken the effectiveness of the model. If you haven't caught the difference, that is OK. I encounter these types of problems all the time, and they can take me hours sometimes to detect the problem: instead of `false` (a boolean value), it is using `"false"` (a string).

None of these checks should ever be manual; manual verification should be kept to a minimum. Automation should be a high priority, and the suggestions I've so far made can all be added as part of the pipeline. These checks can be generalized to other models for reuse, but at a high level, they can run in parallel as shown in [Figure 4-12](#).

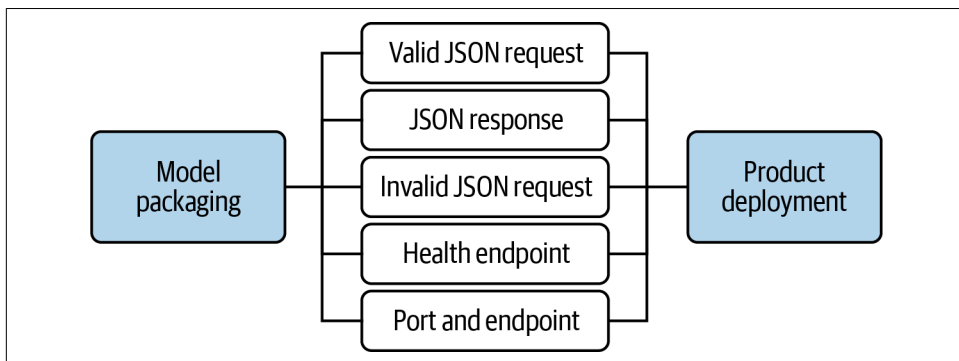


Figure 4-12. Automated checks

Linting

Beyond some of the functional checks I mention, like sending HTTP requests, there are other checks closer to the code in the Flask app that are far simpler to implement, like using a linter (**I recommend Flake8 for Python**). It would be best to automate all these checks to prevent getting into trouble when the production release needs to happen. Regardless of the development environment you are in, I *strongly* recommend enabling a linter for writing code. While creating the Flask application, I found errors as I adapted the code to work with HTTP requests. Here is a short example of the linter's output:

```
$ flake8 webapp/app.py
webapp/app.py:9:13: F821 undefined name 'RobertaTokenizer'
```

Undefined names break applications. In this case, I forgot to import the `RobertaTokenizer` from the `transformers` module. As soon as I realized this, I added the import and fixed it. This didn't take me more than a few seconds.

In fact, the earlier you can detect these problems, the better. When talking about security in software, it is typical to hear “software supply chain,” where the *chain* is all the steps from development to shipping code into production. And in this chain of events, there is a constant push to *shift left*. If you see these steps as one big chain, the leftmost link is the developer creating and updating the software, and the end of the chain (the farthest to the right) is the released product, where the end-consumer can interact with it.

The earlier you can shift left the error detection, the better. This is because it is cheaper and faster than waiting all the way until it is in production when a rollback needs to happen.

Continuous improvement

A couple of years ago, I was the release manager of a large open source software. The software was so complicated to release that it would take me anywhere from two days up to a whole week. It was tough to make improvements as I was responsible for other systems as well. One time, while trying to get a release out, following the many different steps to publish the packages, a core developer asked me to get in one last change. Instead of saying “No” right away, I asked: “*Has this change been tested already?*”

The response was completely unexpected: “*Don't be ridiculous, Alfredo, this is a one-line change, and it is a documentation comment in a function. We really need this change to be part of the release.*” The push to get the change in came all the way from the top, and I had to budge. I added the last-minute change and cut the release.

The very first thing the next morning, we came back to users (and most importantly, customers) all complaining that the latest release was completely broken. It would install, but it would not run at all. The culprit was the one-line change that, although it was a comment within a function, was being parsed by other code. There was an unexpected syntax in that comment, so it prevented the application from starting up. The story is not meant to chastise the developer. He didn't know better. The whole process was a learning moment for everyone involved, and it was now clear how expensive this one-line change was.

There was a set of disruptive events that followed. Aside from restarting the release process, the testing phase for the one change took another (extra) day. Lastly, I had to *retire* the released packages and redo the repositories so new users would get the previous version.

It was beyond costly. The number of people involved and the high impact made this an excellent opportunity to assert that this should not be allowed again—even if it is a one-line change. The earlier the detection, the least impact it will have, and the cheaper it is to fix.

Conclusion

Continuous delivery and the practice of constant feedback is crucial for a robust workflow. As this chapter proves, there is a lot of value in automation and continuous improvement of the feedback loop. Packaging containers, along with pipelines and CI/CD platforms in general, are meant to make it easier to add more checks and verifications, which are intended to increase the confidence of shipping models into production.

Shipping models into production is the number one objective, but doing so with very high confidence, in a resilient set of steps, is what you should strive for. Your task does not end once the processes are in place. You must keep finding ways to thank yourself later by asking the question: what can I add today to make my life easier if this process fails? Finally, I would strongly recommend creating these workflows in a way that makes it easy to add more checks and verifications. If it is hard, no one will want to touch it, defeating the purpose of a robust pipeline to ship models into production.

Now that you have a good grasp of delivering models and what the automation looks like, we will dive into AutoML and Kaizen in the next chapter.

Exercises

- Create your own Flask application in a container, publish it to a GitHub repository, document it thoroughly, and add GitHub Actions to ensure it builds correctly.
- Make changes to the ONNX container so that it pushes to Docker Hub instead of GitHub Packages.
- Modify a SageMaker pipeline, so it prompts you before registering the model after training it.
- Using the Azure SDK, create a Jupyter notebook that will increase the percentile of traffic going to a container.

Critical Thinking Discussion Questions

- Name at least four critical checks you can add to verify a packaged model in a container is built correctly.
- What are the differences between canary and blue-green deployments? Which one do you prefer? Why?
- Why are cloud pipelines useful versus using GitHub Actions? Name at least three differences.
- What does *packaging a container* mean? Why is it useful?
- What are three characteristics of package machine learning models?