We'll Do It Live: Updating Machine Learning Models on Flask/uWSGI with No Downtime

Kevin Zecchini
Oct 21, 2019 · 9 min read

How can you make live updates to a Flask API under the hood when it is already serving requests? And how can you do it without any downtime?

The answer appears obvious: Just load a new model from a POST request, right? This may seem like a simple, appropriate approach, but the devil is always in the details.

This blog post will walk through the steps of getting a Flask application with an endpoint to update a model into production. First we will consider this approach for a bare-bones Flask application. Next, we will deploy it behind a Web Server Gateway Interface (WSGI) and find there are still shortcomings. Then, we will add a process lock for model updates to ensure we continue to serve responses while our app updates. Finally, we will discuss scaling out our WSGI process lock solution to multiple servers via Kubernetes.

Why we need more than just a Flask App

Flask is one of the most popular REST API frameworks used for hosting machine learning (ML) models. The choice is heavily influenced by a data science team's expertise in Python and the reusability of training assets built in Python. At WW, Flask is used extensively by the data science team to serve predictions from various ML models. However, there are a few considerations that need to be made before a Flask application is production-ready.

If Flask code isn't modified to run asynchronously, it only can run one request per process at a time. When you scale from your local development up to production loads

of hundreds — or thousands — of requests per second (rps), this can turn into a problem. The first step to productionalizing your Flask app is to place it behind a WSGI which can spawn and manage threads/processes. This post will detail the configuration of uWSGI, but other frameworks are available such as Gunicorn and Gevent.

A primer on the GIL, threads, and processes

Python uses a Global Interpreter Lock (GIL) that prevents multiple threads from executing Python bytecode at once. The GIL is released on I/O operations including waiting on a socket and file system reads/writes. Thus, in some circumstances, threading can lead to performance improvements. If an operation is not I/O heavy, too many threads can cause a GIL bottleneck. In this case, higher concurrency in Python is achieved by using multiple parallel processes.

Placing a Flask application behind a WSGI will allow you to increase concurrency through processes and/or threads. For uWSGI, a Flask application will be deployed according to the number of `threads` and `processes` declared in the configuration. Because ML predictions will usually not benefit from threading, we set `threads=1` and gain our concurrency via multiple processes. Running multiple processes does come with side effects: we need to hold the application memory for each process, and processes cannot explicitly communicate with each other.

Single-threaded Flask to uWSGI

First, we will start with an example application built in Flask and walk through deploying it behind uWSGI, allowing us to scale to multiple processes. Let's say you have a simple application `myapp.py` that globally loads a model, has a `predict` endpoint, and has an `update-model` endpoint:

```
from flask import Flask, request, jsonify
import dill
import sys

app = Flask(__name__)

# global model
model = None

@app.route('/predict', methods=['GET'])
def predict():
    features = request.args.getlist('features')
```

```
    pred = model.predict(features)
    return jsonify({'prediction': pred})

@app.route('/update-model', methods=['POST'])
def update_model():
    new_path = request.args.get('path')
    load_model(new_path)
    return jsonify({'status': 'update complete!'})

def load_model(model_path):
    global model
    with open(model_path, 'rb') as f:
        model = dill.load(f)

if __name__ == '__main__':
    # get path to initialize model
    path = sys.argv[1]
    load_model(path)
    app.run()
```

To deploy via uWSGI, first install via pip:

```
pip install uwsgi
```

Configuration for uWSGI can be passed in the command line via a uwsgi.ini file. An example configuration file to host a multiprocess uWSGI server at port 5000 could look like:

```
[uwsgi]
socket = 0.0.0.0:5000
protocol = http
module = myapp:app
threads = 1
processes = 4
```

We run the server with the command:

```
uwsgi uwsgi.ini
```

Keep in mind that the uWSGI module does not run as the main python application, so some refactoring may be needed to pull the initial model load out of the main block.

For example:

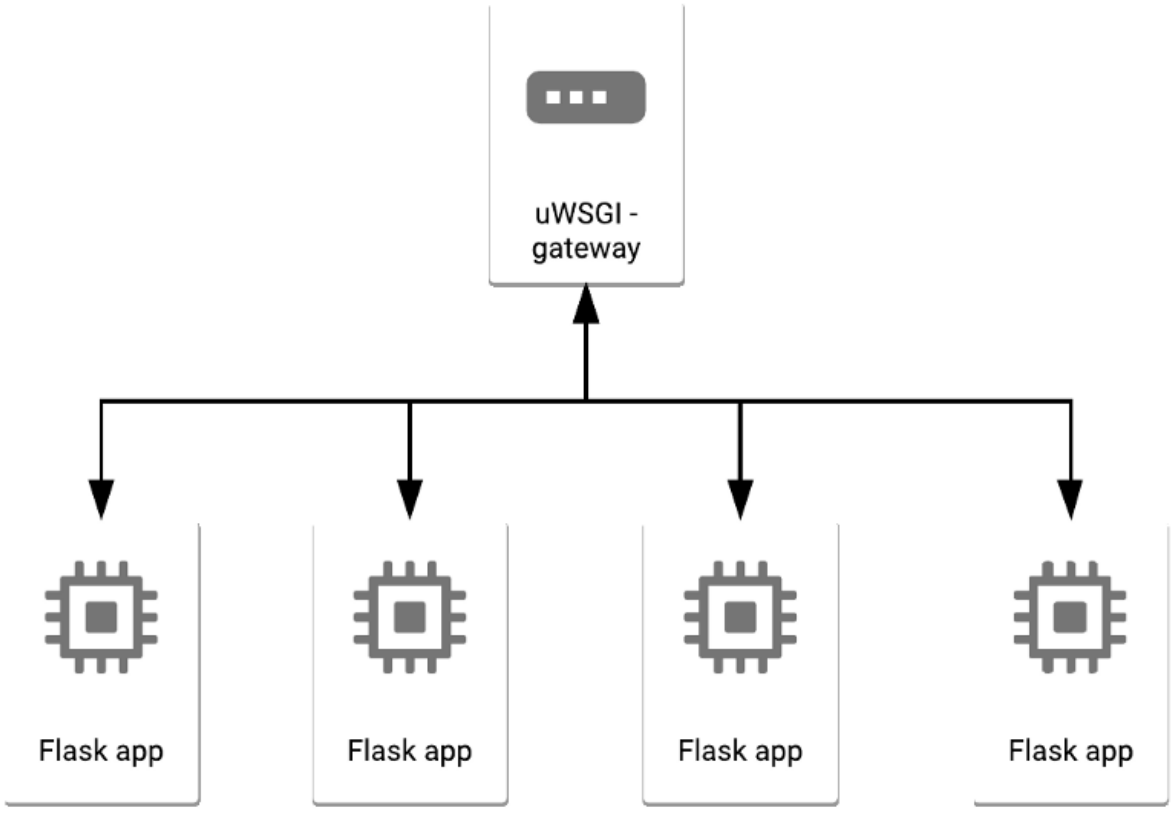
```
# prefork load model so each uWSGI process has a copy
# will not run if this is in __main__ block

model_path = os.environ.get('MODEL_PATH')
load_model(model_path)

if __name__ == '__main__':
    app.run()
```

This should get a simple uWSGI deployment up and running. For more detailed information, see the uWSGI quickstart docs.

Now your uWSGI deployment looks something like:



Simple uWSGI deployment with 4 processes

We are ready to serve! But are we ready to update our model?

Now that Flask is behind a WSGI, throughput is increased linearly by the number of processes running — great news! Everything is going smoothly in production.

But suppose that we've trained a new model and want to update our service. We would hit our POST endpoint to load the new model into memory. Remember, Flask requests are blocking, which means that a POST request to load a new model into memory (which might take a few seconds/minutes) will block the execution of other GET requests. Having your app behind a WSGI framework partially solves this issue. Let's say we deploy with *p* processes. We have one process updating our model via a POST request, while the rest of our processes are serving GET requests. We only drop throughput by *p-1* processes!

However, that's not the full story. Because state is not shared between Python processes, we now have one process with an updated model, and the rest still serving the old model. The question now becomes how can we update all processes effectively? Can we update all processes while still retaining the minimum throughput of *p-1* processes?

Updating models on the Fly: uWSGI middleware, caching, and process locking

The first step to ensure all processes are updated to the latest model is to introduce a cache (i.e., a redis cache or uwsgi cache) to keep track of the most recent model. We update the cached model hash and model location during the `/model-update` POST request. For each incoming request, a diff will be performed between the processes' model hash and the cached hash. If there is a difference, we need to update this process. An implementation using a redis cache (running on localhost) could look like this:

```
import redis

@app.route('/update-model', methods=['POST'])
def update_model():
    r = get_cache()
    new_path = request.args.get('path')
    load_model(new_path)
    r.set('model_hash', model.hash)
    r.set('model_location', path)

return jsonify({'status': 'update complete!'})

@app.teardown_request
def check_cache(ctx):
    r = get_cache()
    global model
    cached_hash = r.get('model_hash')
```

```
        if model.hash != cached_hash:
            model_location = r.get('model_location')
            load_model(model_location)

    def get_cache():
        if 'cache' not in g:
            g.cache = redis.Redis()
        return g.cache
```

Our application will now update all of the processes to the new model. But since there is no coordination between our processes, we don't know how many processes are updating at a given time. Worst case scenario, we could be updating *p-1* processes at the same time. This means that throughput is reduced to only 1 process, which is the same as if our application was just a single Flask app.

In order to manage these blocking update processes, we introduce a locking mechanism via our caching framework. If a process detects that it needs to update to the most recent model, it will also check to see if a lock is available to obtain from the cache. This lock acts as a busy signal. If our `busy_signal` is 1, then we know some other process is running an update and we should not block the current process. If the `busy_signal` is 0, then we know that no other processes are currently updating, and we can continue our update safely.

Let's add this flag into the previous functions:

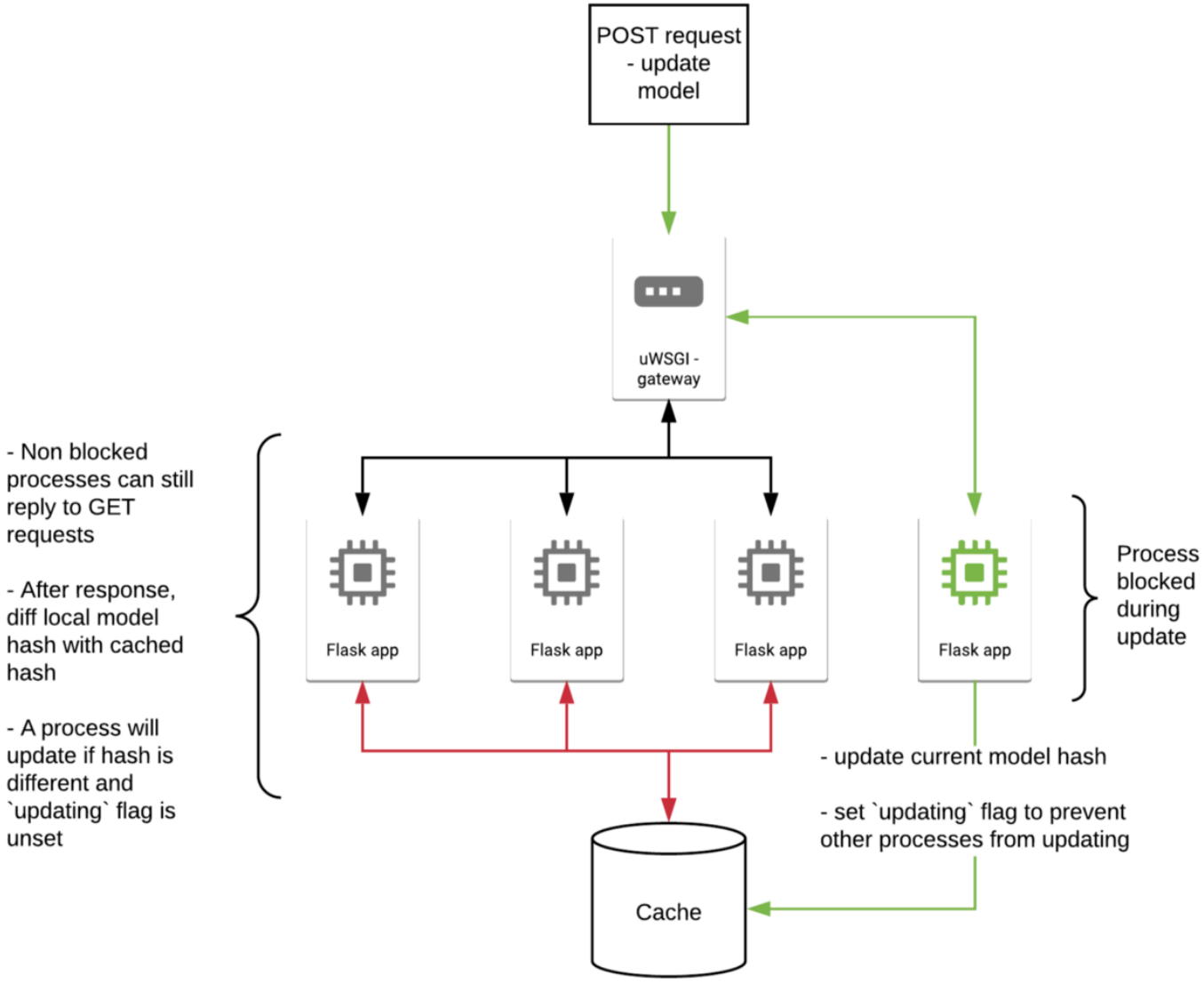
```
    @app.route('/update-model', methods=['POST'])
    def update_model():
        r = get_cache()
        new_path = request.args.get('path')
        busy_signal = int(r.get('busy_signal'))
        if not busy_signal:
            r.set('busy_signal', 1)
            load_model(new_path, r)
            r.set('busy_signal', 0)
        return jsonify({'status': 'update complete!'})

    @app.teardown_request
    def check_cache(ctx):
        r = get_cache()
        global model
        cached_hash = r.get('model_hash')
        if model.hash != cached_hash:
            busy_signal = int(r.get('busy_signal'))
            if not busy_signal:
                # if not busy, we set signal to busy, then update
                r.set('busy_signal', 1)
```

```
            model_location = r.get('model_location')
            load_model(model_location, r)
            r.set('busy_signal', 0)

def get_cache():
    if 'cache' not in g:
        g.cache = redis.Redis(decode_responses='utf-8')
    return g.cache
```

With this change, we have a way for each process to check a busy signal to make sure no updates occur concurrently. This means that the minimum number of nonblocking processes running is always guaranteed to be *p-1*. The following diagram illustrates how a POST request would update the model:



Example POST request to update a model without blocking GET requests

However, there is yet another issue. Even though we are checking the cache for model

updates in a `teardown_request`, Flask will execute this function *before* returning a response. This means we are performing the `busy_signal` check and potentially a full model update in a blocking manner. Although this would only affect *p-1* total requests per model update, we don't want any of our requests to have to wait for a blocking process to complete. Since Flask is a WSGI application, it cannot fundamentally handle anything outside of returning the response object. In order to modify behavior beyond the construction of the response, we can add Middleware layers to the WSGI application.

To perform a callback action after a response is returned, we can create a middleware layer and return a ClosingIterator which executes our function. Our middleware code looks like this:

```
class AfterResponse:
    '''App extension which wraps the middleware
    '''
    def __init__(self, app):
        self.callbacks = []

        # install extension
        app.after_response = self

        # install middleware
        app.wsgi_app = AfterResponseMiddleware(app.wsgi_app, self)

    def __call__(self, callback):
        self.callbacks.append(callback)
        return callback

    def flush(self):
        for fn in self.callbacks:
            fn()

class AfterResponseMiddleware:
    '''WSGI middleware to return `ClosingIterator`
       with callback functions
    '''
    def __init__(self, application, after_response_ext):
        self.application = application
        self.after_response_ext = after_response_ext

    def __call__(self, environ, after_response):
        iterator = self.application(environ, after_response)
        try:
            return ClosingIterator(iterator,
                                   [self.after_response_ext.flush])
        except:
            return iterator
```

To incorporate this in our code, all we need to do is register the middleware with the Flask application and add a decorator to any function that needs to execute after the response is sent back to the requestor. Because this happens outside of the Flask context, we use a `werkzeug` local context to hold our redis connection instead of using the Flask global context `g`.

```
from werkzeug.local import Local

app = Flask(__name__)
AfterResponse(app)

local = Local()

@app.after_response
def check_cache():
    r = get_cache()
    global model
    cached_hash = r.get('model_hash')
    if model.hash != cached_hash:
        busy_signal = int(r.get('busy_signal'))
        if not busy_signal:
            # if not busy, we update and first set signal to busy
            r.set('busy_signal', 1)
            model_location = r.get('model_location')
            load_model(model_location, r)
            r.set('busy_signal', 0)

def get_cache():
    cache = getattr(local, 'cache', None)
    if cache is None:
        local.cache = redis.Redis(decode_responses='utf-8')
    return local.cache
```

Finally, this ensures that no requests will be blocked during our service updates. For each request, the cache check and model update happens after a response is already given.

Scaling up: Kubernetes deployments

In this section, we will detail the small modifications that must be made when deploying this system in a Kubernetes cluster.

So far we've created a single server system that can perform non-blocking model updates through process management and WSGI middleware. After getting a single WSGI server working seamlessly, scaling this service to multiple machines is a piece of

cake. We can create a Docker container that will start a uWSGI server and scale up our replicas through Kubernetes.

Our Dockerfile will look like this:

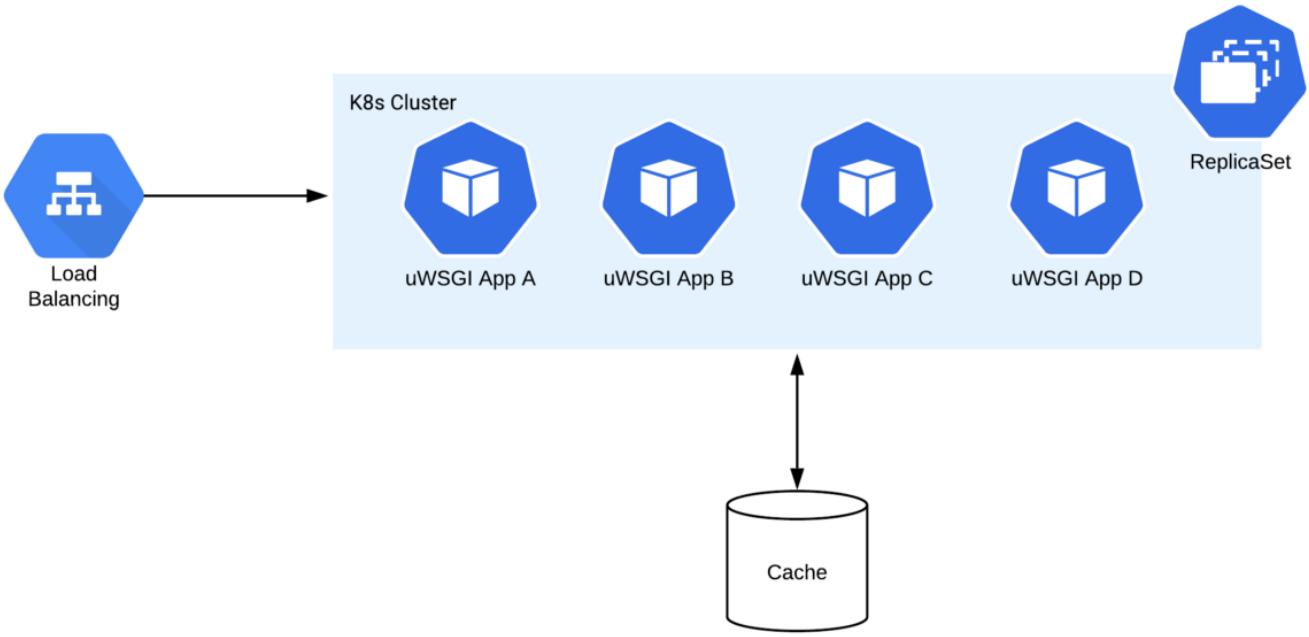
```
FROM python:3.6-stretch

COPY myapp.py ./myapp.py
COPY uwsgi.ini ./uwsgi.ini

RUN pip install uwsgi dill redis Flask

ENTRYPOINT ["uwsgi", "uwsgi.ini"]
```

After deploying through Kubernetes, we will have an endpoint that will route traffic to each one of our uWSGI servers.



uWSGI Kubernetes deployment with shared cache

The most important part to note here is how the cache manages locks. For an incoming POST request, we want a global key in the cache to update with the new model. For any Pod that did not receive the request, it can still access the global `model_hash` and perform a diff.

However for our `busy_signal`, we probably don't want one global key shared by all Pods. If this were the case, we would be limited to only one process update at a time, which could propagate very slowly through our replica set depending on the size of our deployment.

Instead, we may want to allow each Pod to update one process at a time. This would mean that we need a signal for *each individual* uWSGI Pod. The value of the key can be determined at deploy time by an environment variable, but for now we can call them `busy_signal_a`, `busy_signal_b`, etc.

Managing our locking in this manner will ensure that we have at least *p-1* processes active within each uWSGI Pod, for a minimum *num_replicas*(p-1)* active processes throughout the entire Kubernetes deployment. This leaves the maximum processes that can update at once equal to the number of replicas in our set.

In summary, we started with a single threaded Flask application, deployed behind uWSGI, and developed a method using process locks to handle live updates in production. I hope this pattern is helpful in updating your ML models without any downtime. All code from this post is available here.

> *— Kevin Zecchini, Senior Data Scientist, WW*

Interested in joining the WW team? Check out the careers page to view technology job listings as well as open positions on other teams.

Python Machine Learning Data Science Flask API