Article
Zhong Li · Sep 6, 2020 18m read

Open Exchange

Deploy ML/DL models into a consolidated AI demo service stack

Keywords: IRIS, IntegratedML, Flask, FastAPI, Tensorflow Serving, HAProxy, Docker, Covid-19

Purpose:

We touched on some quick demos of deep learning and machine learning over the past few months, including a simple Covid-19 X-Ray image classifier and a Covid-19 lab result classifier for possible ICU admissions. We also touched on an IntegratedML demo implementation of the ICU classifier. While the "data science" hiking still goes on, it might also be a good time to try some AI service deployment from the "data engineering" perspective - could we wrap up everything we touched on so far into a set of service APIs? What are the common tools, components, and infrastructure that we could leverage to achieve such a service stack in its simplest possible approach?

Scope

In scope:

As a jump start, we can simply use docker-compose to deploy the following dockerised components into an AWS Ubuntu server

- **HAProxy** - load balancer
- **Gunicorn** vs. **Univorn** - web gateway servers
- **Flask** vs. **FastAPI** - application servers for web app UI , service API definitions and Heatmap generations etc
- **Tensorflow Model Serving** vs. **Tensorflow-GPU Model Serving** - application backend servers for image etc classifications etc
- IRIS **IntegratedML** - consolidated App+DB AutoML with SQL interface
- Python3 in **Jupyter Notebook** to emulate a client for **benchmarking**
- Docker and **docker-compose**
- **AWS Ubuntu** 16.04 with a Tesla T4 GPU

Note: Tensorflow Serving with GPU is for demo purpose only - you can simply switch off the gpu related image (in a dockerfile) and the config (in the docker-compose.yml).

Out of scope or on next wish list:

- **Nginx** or **Apache** etc web servers are omitted in demo for now
- **RabbitMQ** and Redis - queue broker for reliable messaging that can be replaced by IRIS or Ensemble.
- **IAM** (Intersystems API Manger) or **Kong** is on wish list
- **SAM** (Intersystems System Alert & Monitoring)
- **ICM** (Intersystems Cloud Manager) with **Kubernetes** Operator - always one of my favorites since its birth
- **FHIR** (Intesystems IRIS based FHIR R4 server and FHIR Sandbox for SMART on FHIR apps)
- **CI/CD** devop tools or **Github Actions**

A "Machine Learning Engineer" would inevitably put hands all over these components to provision some production environments along service life-cycles anyway. We can scope more in over the time.

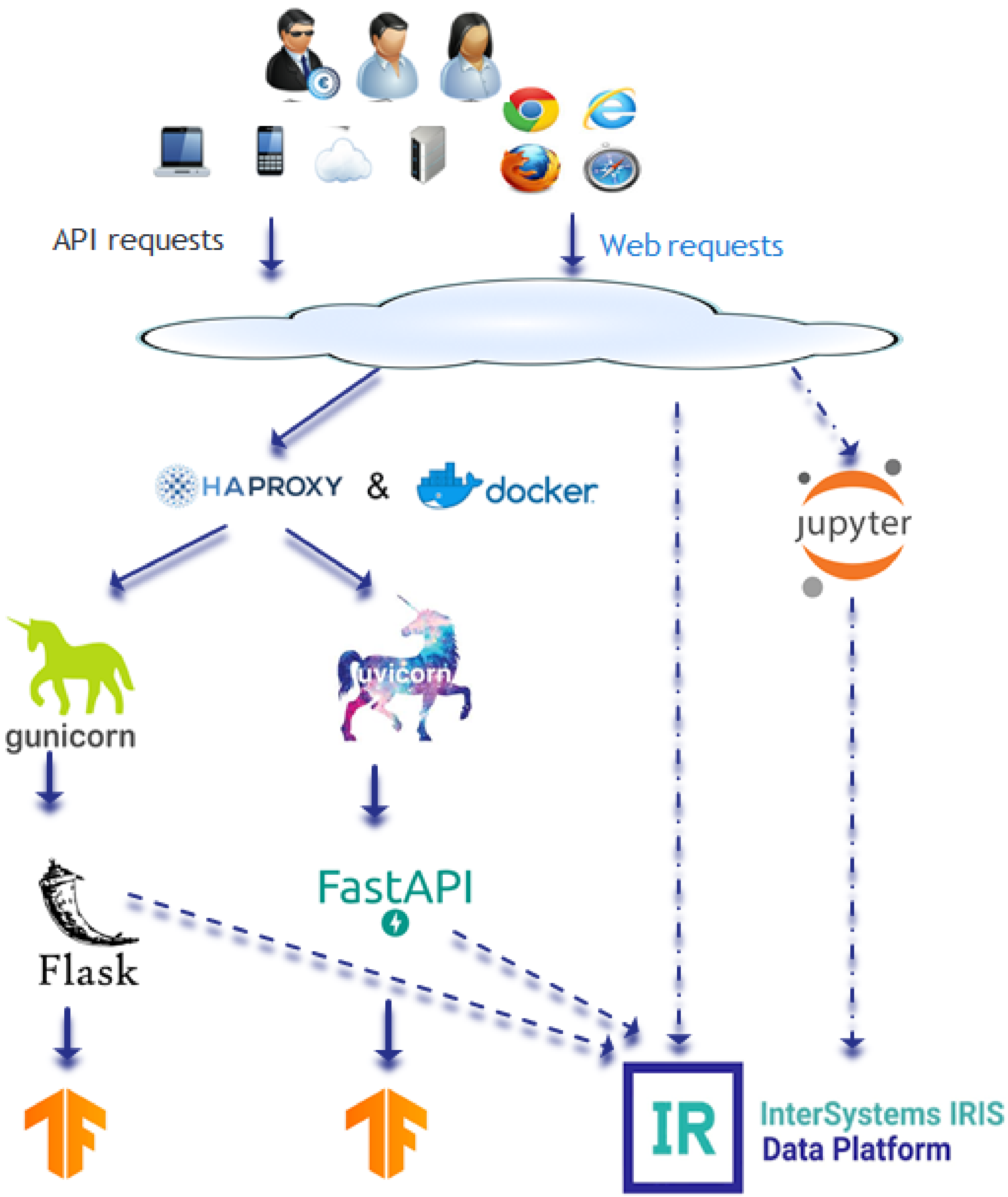
Github repository

The full source code is at: https://github.com/zhongli1990/covid-ai-demo-deployment

Also the integratedML-demo-template repository is reused together with the new repository.

Deployment Pattern

Below shows the logical deployment pattern for this "AI demo in Dockers" testing framework.



For demo purpose I deliberately created 2 separate stacks for deep learning classification as well as web rendering, then used a HAProxy as a soft load balancer to distribute the incoming API requests across these 2 stacks in a stateless way.

- Guniorn + Flask + Tensorflow Serving
- Univcorn + FaskAPI + Tensorflow Serving GPU

IRIS with IntegratedML is used for machine learning demo samples as i.e. in the previous post of ICU prediction.

I omitted some common components in current demo that would be needed or considered for production services:

- Web servers: Nginx or Apache etc. They would be needed between HAProxy and Gunicorn/Uvicorn, for proper HTTP session handling i.e. avoid DoS attacks etc.
- Queue manager and DBs: RabbitMQ and/or Redis etc, between Flask/FastAPI and backend serving, for reliable Async serving and data/config persistence etc.
- API Gateway: IAM or Kong clusters, between HAProxy load-balancer and web server for API management without creating a sing-point-of-failure.
- Monitoring & Alert: SAM would be nice.
- Provisioning for CI/CD devops: ICM with K8s would be needed for cloud neutral deployment & management, and for CI/CD with other common devops tools.

Actually, IRIS itself can certainly be used as enterprise grade queue manager as well as a high-performing database for reliable messaging. In the pattern analysis it becomes apparent IRIS can be in place of RabbitMQ/Redis/MongoDBs etc queue brokers and databases, so it would be better consolidated with much less latency and better throughout performance. And even more, IRIS Web Gateway (previously CSP Gateway) can certainly be positioned in place of Gunicorn or Unicorn etc, right?

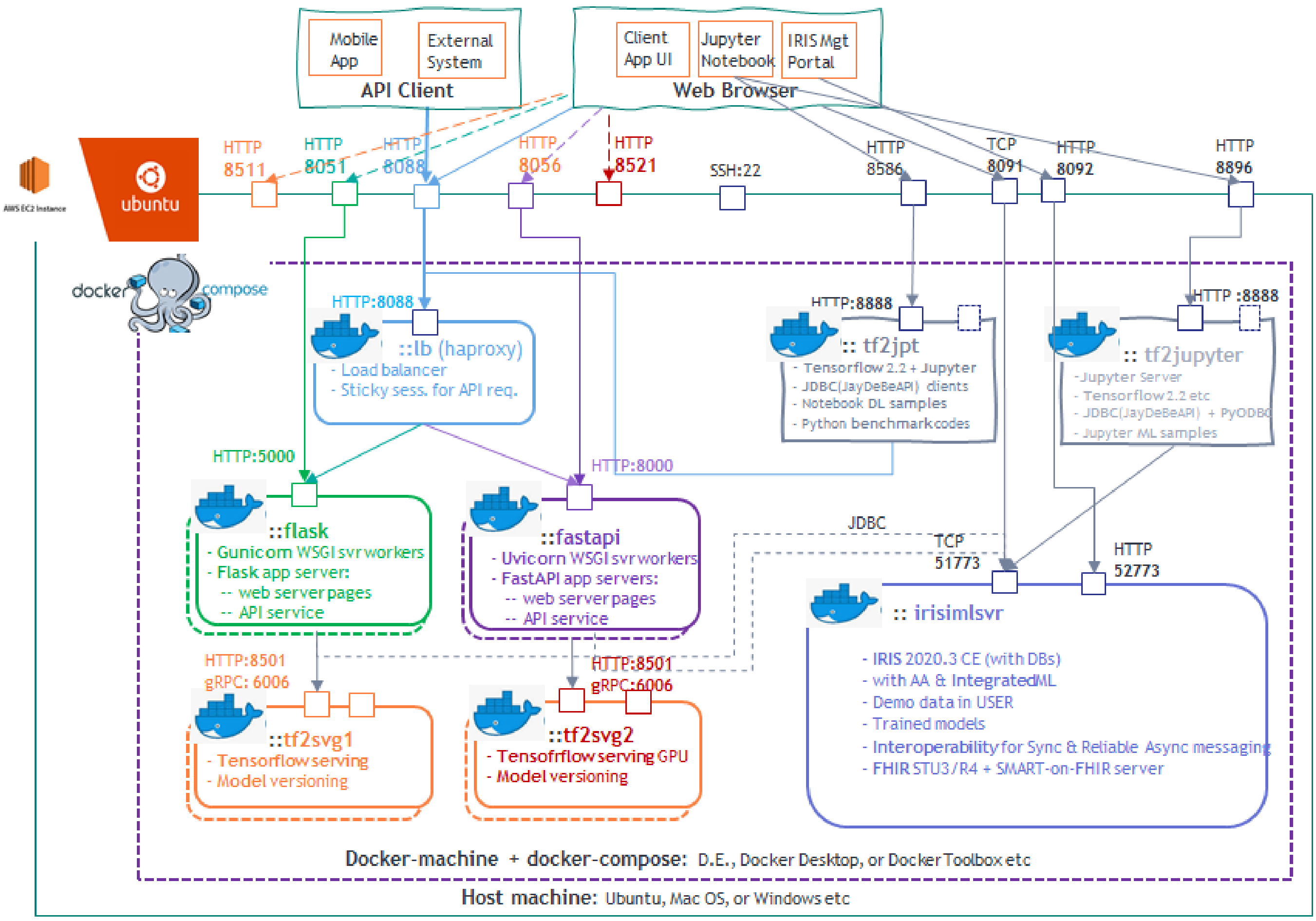
Environment Topology

There are a few common options to implement the above logical pattern in all-Docker components. On top of our heads would be:

- docker-compose
- docker swarm etc
- Kubernetes etc
- ICM with K8s Operation

This demo starts with "docker-compose" for functional PoC and some benchmarking. Certainly we'd love to use K8s and possibly with ICM too over the time.

As described in its docker-compose.yml file, a physical implementation of its environment topology on an AWS Ubuntu server would end up something like this:



The above diagram shows how those **service ports** of all Docker instances are mapped and exposed directly on the Ubuntu server for demo purpose. In production it should be all security hardened. And for pure demo purpose, all containers are connected into the same Docker network; while in production it would be separated as external routable vs internal non-routable.

Dockerised Components

Below shows how those **storage volumes** in host machine are mounted to each container instance as specified in this docker-compose.ymlfile:

ubuntu@ip-172-31-35-104:/zhong/flask-xray$ tree ./ -L 2

```
./
??? covid19                                (Flask+Gunicorn container and
Tensorflow Serving container will mount here)
?   ???
 app.py                                   (Flask main app
:  Both web application and API service interfaces are defined and implemented here)
?   ??? covid19_models                  (Tensorflow models
 are published and version
ed here for image classification Tensorflow Serving container with CPU)
?   ??? Dockerfile                         (Flask server with Gunicorn:
CMD ["gunicorn", "app:app", "--bind", "0.0.0.0:5000", "--workers", "4", "--threads",
"2"])
?   ???
```

```
 models                               (Models in .h5 format for Flask app and API dem
o of heatmap generation by grad-cam on X-Rays)
?   ??? __pycache__
?   ??? README.md
?   ???
 requirements.txt             (Python packages needed for the full Flask+Gunicorn app
s)
?   ??? scripts
?   ??? static                                    (Web static files)
?   ??? templates                           (Web rendering templates)
?   ??? tensorflow_serving          (Config file for tensorflow serving service)
?   ??? test_images
??? covid-fastapi                      (FastAPI+Uvicorn container and
Tensorflow Serving with GPU container will mount here)
?   ??? covid19_models              (
Tensorflow serving GPU models
 are published and versioned here for image classification)
?   ??? Dockerfile                          (Uvicorn+FastAPI
 server are started here:



)
?   ??? main.py                               (FastAPI app
: both web application and API service interfaces are defined and implemented here)
?   ???
 models                            (Models in .h5 format for FastAPI app and API demo
 of heatmap generation by grad-cam on X-Rays)
?   ??? __pycache__
?   ??? README.md
?   ??? requirements.txt
?   ??? scripts
?   ??? static
?   ??? templates
?   ??? tensorflow_serving
?   ??? test_images
??? docker-
compose.yml      (Full stack Do
cker definition file.  Version 2.3
 is used to accommodate Docker GPU "nvidia runtime", otherwise can be version 3.x)
??? haproxy                              (HAProxy
docker service is defined here.  Note: sticky session can be defined for backend LB.
)
?   ??? Dockerfile
?   ??? haproxy.cfg
??? notebooks                          (Jupyter Notebook
container service with Tensorflow 2.2 and Tensorboard etc)
    ??? Dockerfile
    ???
 notebooks                  (Sa
mple notebook files to
emulate external API Client apps for functional tests and
API benchmark tests in Python on the load balancer etc)
??? requirements.txt
```

Note: the above docker-compose.yml is for deep learning demo of Convid X-Rays. It is used together with another integratedML-demo-template's docker-compose.yml to form the full service stack as displayed in the environment topology.

Service Start-ups

A simple **docker-compose up -d** would start up all container services:

ubuntu@ip-172-31-35-104:$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS
PORTS NAMES
31b682b6961d iris-aa-server:2020.3AA "/iris-main" 7 weeks ago Up 2 days (healthy)
2188/tcp, 53773/tcp, 54773/tcp, 0.0.0.0:**8091**->51773/tcp, 0.0.0.0:**8092**->52773/tcp **iml-template-master_irisimlsvr_1**
6a0f22ad3ffc haproxy:0.0.1 "/docker-entrypoint.…" 8 weeks ago Up 2 days
0.0.0.0:**8088**->8088/tcp **flask-xray_lb_1**
71b5163d8960 ai-service-fastapi:0.2.0 "uvicorn main:app --…" 8 weeks ago Up 2 days
0.0.0.0:**8056**->8000/tcp **flask-xray_fastapi_1**
400e1d6c0f69 tensorflow/serving:latest-gpu "/usr/bin/tf_serving…" 8 weeks ago Up 2 days
0.0.0.0:8520->8500/tcp, 0.0.0.0:**8521**->8501/tcp **flask-xray_tf2svg2_1**
eaac88e9b1a7 ai-service-flask:0.1.0 "gunicorn app:app --…" 8 weeks ago Up 2 days
0.0.0.0:**8051**->5000/tcp **flask-xray_flask_1**
e07ccd30a32b tensorflow/serving "/usr/bin/tf_serving…" 8 weeks ago Up 2 days
0.0.0.0:8510->8500/tcp, 0.0.0.0:**8511**->8501/tcp **flask-xray_tf2svg1_1**
390dc13023f2 tf2-jupyter:0.1.0 "/bin/sh -c '/bin/ba…" 8 weeks ago Up 2 days
0.0.0.0:**8506**->6006/tcp, 0.0.0.0:**8586**->8888/tcp **flask-xray_tf2jpt_1**
88e8709404ac tf2-jupyter-jdbc:1.0.0-iml-template "/bin/sh -c '/bin/ba…" 2 months ago Up 2 days
0.0.0.0:6026->6006/tcp, 0.0.0.0:**8896**->8888/tcp **iml-template-master_tf2jupyter_1**

docker-compose up --scale fastapi=2 --scale flask=2 -d for example will horizontally scale up to 2x Gunicorn+Flask containers and 2x Univcorn+FastAPI containers:

ubuntu@ip-172-31-35-104:/zhong/flask-xray$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
dbee3c20ea95 ai-service-fastapi:0.2.0 "uvicorn main:app --…" 4 minutes ago Up 4 minutes 0.0.0.0:8057->8000/tcp **flask-xray_fastapi_2**
95bcd8535aa6 ai-service-flask:0.1.0 "gunicorn app:app --…" 4 minutes ago Up 4 minutes 0.0.0.0:8052->5000/tcp **flask-xray_flask_2**

... ...

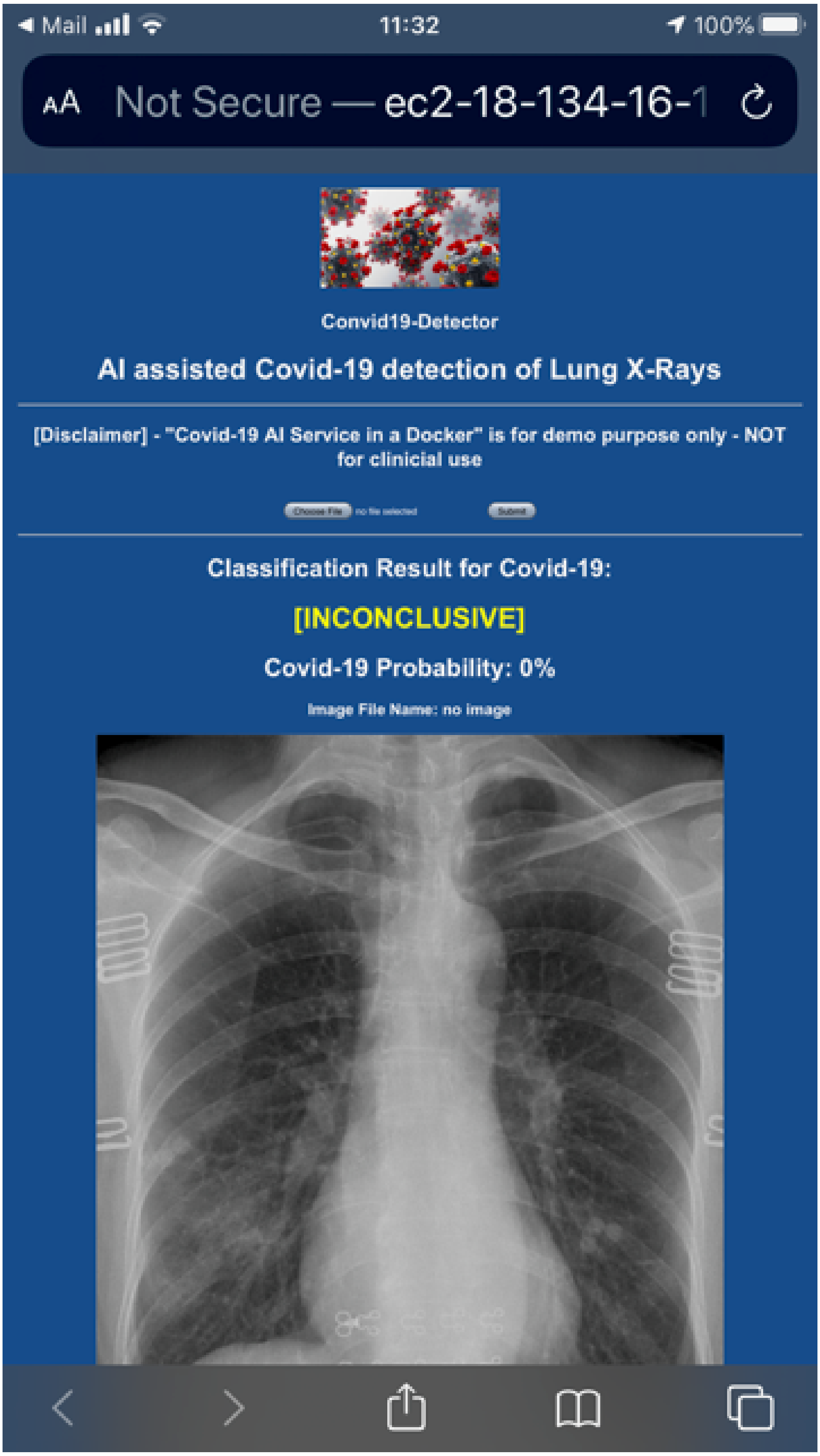
Running another "docker-compose up -d" in the "integrtedML-demo-template"'s working directory has brought up the irisimlsvr and tf2jupyter container in the above list.

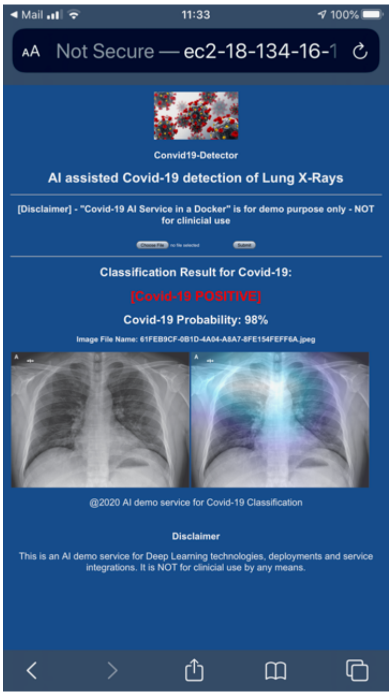
Tests

1. AI demo web app with a simple UI

After starting up the above docker services, we can visit a demo web application for X-Ray Covid-19 lung detection hosted in an AWS EC2 instance at a temp address at http://ec2-18-134-16-118.eu-west-2.compute.amazonaws.com:8056/

Here below is a couple of screens captured from my mobile. It has a very simple demo UI: basically I just click "Choose File" then "Submit" button to upload an X-Ray image, then the app will show a classification report. If it is classified as Covid-19 X-Ray, an heatmap will be shown to emulate the "detected" lesion area via DL; and if not, the classification report will only show the upload X-Ray image.


Mail
11:32
100%
AA Not Secure — ec2-18-134-16-1
Convid19-Detector
AI assisted Covid-19 detection of Lung X-Rays
[Disclaimer] - "Covid-19 AI Service in a Docker" is for demo purpose only - NOT for clinicial use
Classification Result for Covid-19:
[INCONCLUSIVE]
Covid-19 Probability: 0%
Image File Name: no image

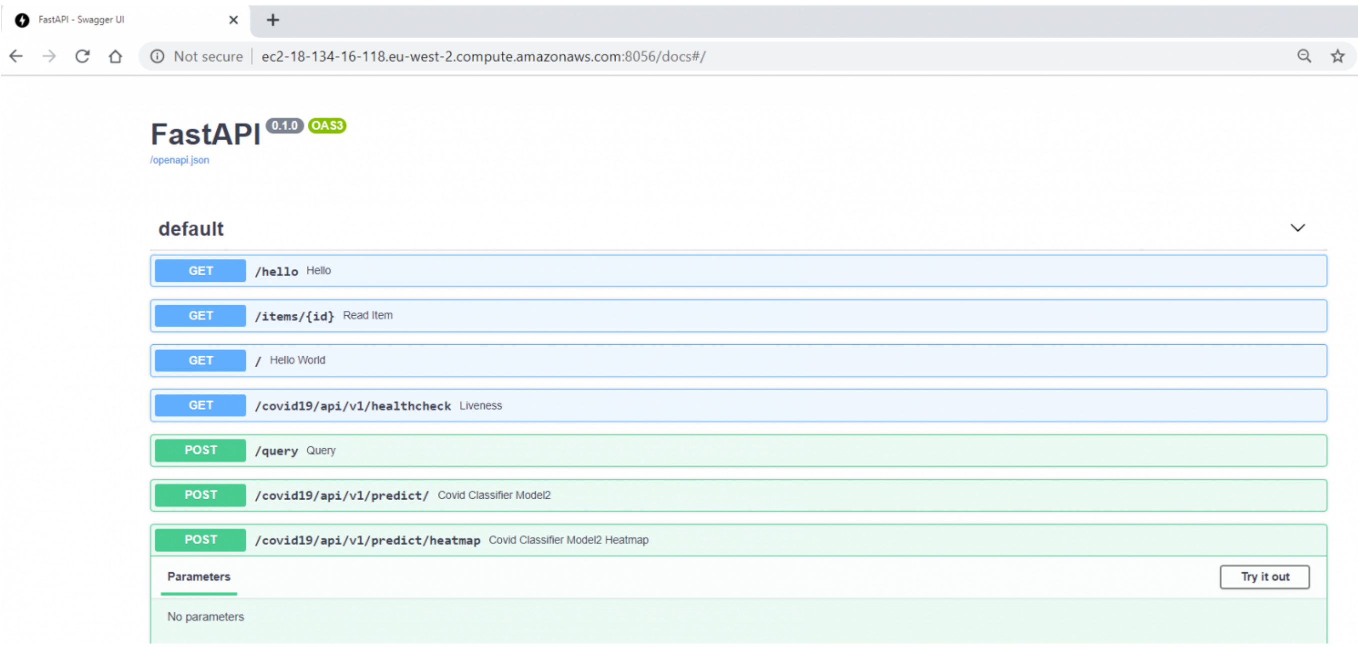


The web app is a Python server page whose logic are mainly coded in [FastAPI's main.py] file, as well as in [Flask's app.py] file.

When having a bit more spare time I may document in detail the coding & convention differences between Flask and FastAPI. Actually I hope I could do a Flask vs. FastAPI vs. IRIS for AI demo hosting.

2. Test demo APIs

FastAPI (expose at port 8056) has built in Swagger API docs, as shown below. This is very handy. All I need to do is to use "/docs" in its URL, for example:



I built in a few place holders (such as /hello and /items) and some real demo API interfaces (such as /healthcheck, /predict, and predict/heatmap).

Let's have a quick test on these APIs, by running a few Python lines (as an API Client App emulator) in one of the Jupyter Notebook samples files I scratched up for this AI demo service.

Below I am running this file for example: https://github.com/zhongli1990/covid-ai-demo-deployment/blob/master/notebooks/notebooks/Covid19-3class-Heatmap-Flask-FastAPI-TF-serving-all-in-one-HAProxy2.ipynb

First to test backend TF-Serving (port 8511) and TF-Serving-GPU (port 8521) are up and functioning:

```
!curl http://172.17.0.1:8511/v1/models/covid19  # tensorflow serving
!curl http://172.17.0.1:8521/v1/models/covid19  # tensorflow-gpu serving
```


```
{
 "model_version_status": [
  {
   "version": "2",
   "state": "AVAILABLE",
   "status": {
    "error_code": "OK",
    "error_message": ""
   }
  }
 ]
}
{
 "model_version_status": [
  {
   "version": "2",
   "state": "AVAILABLE",
   "status": {
    "error_code": "OK",
    "error_message": ""
   }
  }
 ]
}
```


Then test the following service APIs are up & running:

- Gunicorn+Flask+TF-Serving
- Unicorn+FastAPI+TF-Serving-GPU
- Load balancer HAProxy in front of bother services above

```
r = requests.get('http://172.17.0.1:8051/covid19/api/v1/healthcheck')  # tf srving do
cker with cpu
print(r.status_code, r.text)
r = requests.get('http://172.17.0.1:8056/covid19/api/v1/healthcheck')  # tf-
serving docker with gpu
print(r.status_code, r.text)
r = requests.get('http://172.17.0.1:8088/covid19/api/v1/healthcheck')  # tf-
serving docker with HAproxy
```

```
print(r.status_code, r.text)
```

And expected results would be:

```
200 Covid19 detector API is live!
200 "Covid19 detector API is live!\n\n"
200 "Covid19 detector API is live!\n\n"
```

Test some functional API interface, such as /**predict/heatmap** to return the classification and heatmap result of an input X-Ray image. The inbound image is based64 encoded before sending in via HTTP POST per API definitions:

```
%%time



# importing the requests library
import argparse
import base64



import requests



# defining the api-endpoint
API_ENDPOINT = "http://172.17.0.1:8051/covid19/api/v1/predict/heatmap"



image_path = './Covid_M/all/test/covid/nejmoa2001191_f3-PA.jpeg'
#image_path = './Covid_M/all/test/normal/NORMAL2-IM-1400-0001.jpeg'
#image_path = './Covid_M/all/test/pneumonia_bac/person1940_bacteria_4859.jpeg'
b64_image = ""
# Encoding the JPG,PNG,etc. image to base64 format
with open(image_path, "rb") as imageFile:
    b64_image = base64.b64encode(imageFile.read())



# data to be sent to api
data = {'b64': b64_image}



# sending post request and saving response as response object
r = requests.post(url=API_ENDPOINT, data=data)



print(r.status_code, r.text)



# extracting the response
```

```
print("{}".format(r.text))
```

All such test images had also been uploaded into GitHub. The result of above code will be:

200 {"Input_Image":"http://localhost:8051/static/source/0198f0ae-85a0-470b-bc31-dc1918c15b9620200906-170443.png","Output_Heatmap":"http://localhost:8051/static/result/Covid19_98_0198f0ae-85a0-470b-bc31-dc1918c15b9620200906-170443.png.png","X-Ray_Classfication_Raw_Result":[[0.805902302,0.15601939,0.038078323]],"X-Ray_Classification_Covid19_Probability":0.98,"X-Ray_Classification_Result":"Covid-19 POSITIVE","model_name":"Customised Incpetion V3"}

{"Input_Image":"http://localhost:8051/static/source/0198f0ae-85a0-470b-bc31-dc1918c15b9620200906-170443.png","Output_Heatmap":"http://localhost:8051/static/result/Covid19_98_0198f0ae-85a0-470b-bc31-dc1918c15b9620200906-170443.png.png","X-Ray_Classfication_Raw_Result":[[0.805902302,0.15601939,0.038078323]],"X-Ray_Classification_Covid19_Probability":0.98,"X-Ray_Classification_Result":"Covid-19 POSITIVE","model_name":"Customised Incpetion V3"}

```
CPU times: user 16 ms, sys: 0 ns, total: 16 ms
Wall time: 946 ms
```

3. Benchmark-test demo service APIs

We set up a HAProxy load balancer instance. We also started a Flask service with 4x workers, and a FastAPI service with 4x workers too.

Why don't we create i.e. 8x Pyhon processes directly in the Notebook file, to emulate 8x concurrent API clients sending requests into the demo service APIs, to see what happens

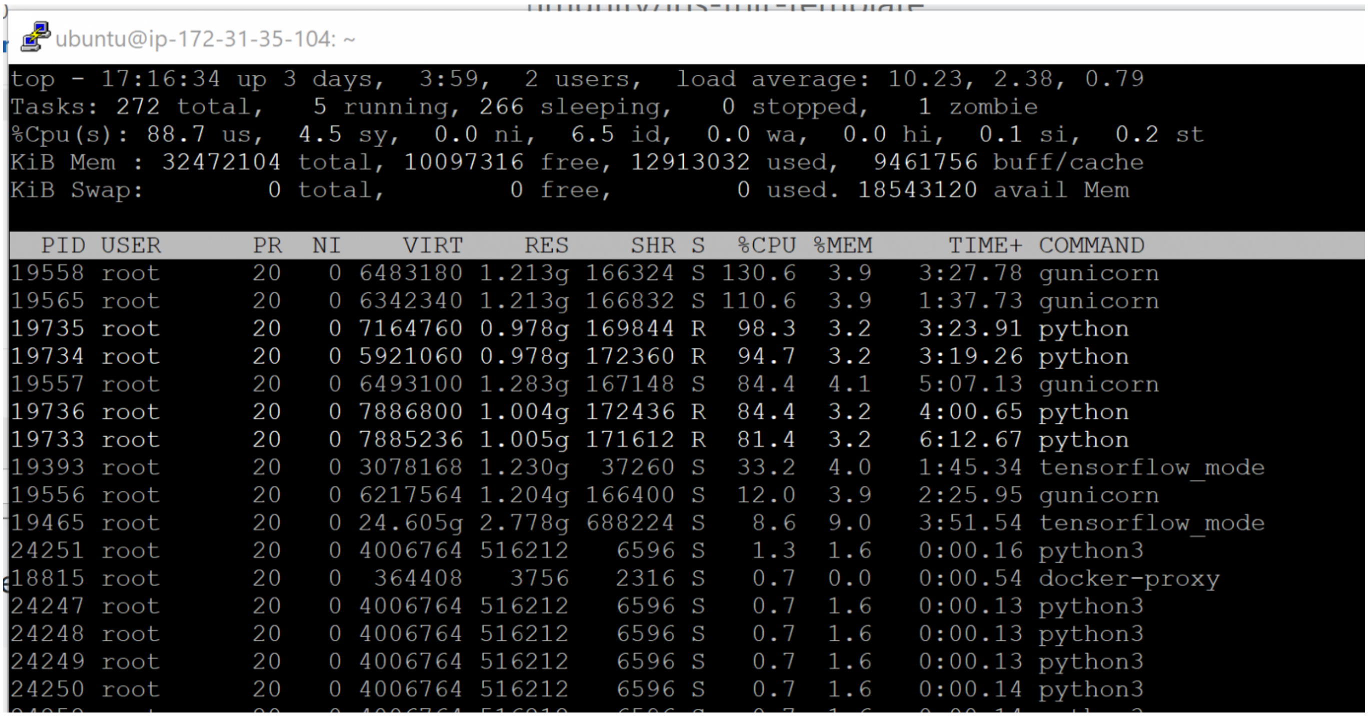
```
#from concurrent.futures import ThreadPoolExecutor as PoolExecutor
from concurrent.futures import ProcessPoolExecutor as PoolExecutor
import http.client
import socket
import time



start = time.time()



#laodbalancer:
API_ENDPOINT_LB = "http://172.17.0.1:8088/covid19/api/v1/predict/heatmap"
API_ENDPOINT_FLASK = "http://172.17.0.1:8052/covid19/api/v1/predict/heatmap"
API_ENDPOINT_FastAPI = "http://172.17.0.1:8057/covid19/api/v1/predict/heatmap"
def get_it(url):
    try:
        # loop over the images
        for imagePathTest in imagePathsTest:
            b64_image = ""
            with open(imagePathTest, "rb") as imageFile:
                b64_image = base64.b64encode(imageFile.read())
```

```
            data = {'b64': b64_image}
            r = requests.post(url, data=data)
            #print(imagePathTest, r.status_code, r.text)
        return r
    except socket.timeout:
        # in a real world scenario you would probably do stuff if the
        # socket goes into timeout
        pass



urls = [API_ENDPOINT_LB, API_ENDPOINT_LB,
        API_ENDPOINT_LB, API_ENDPOINT_LB,
        API_ENDPOINT_LB, API_ENDPOINT_LB,
        API_ENDPOINT_LB, API_ENDPOINT_LB]



with PoolExecutor(max_workers=16) as executor:
    for _ in executor.map(get_it, urls):
        pass

print("--- %s seconds ---" % (time.time() - start))
```

So it took 74s to process 8x27 = 216 test images. This load balanced demo stack was able to process 3 images per second(by returning classification and heatmap results to clients):

```
--- 74.37691688537598 seconds ---
```

From the Putty session's Top command, we can see 8x server processes (4x gunicorn + 4 unicorn/python) started to ramp up as soon as the above benchmark scripts started running



```
ubuntu@ip-172-31-35-104: ~
top - 17:16:34 up 3 days,  3:59,  2 users,  load average: 10.23, 2.38, 0.79
Tasks: 272 total,   5 running, 266 sleeping,   0 stopped,   1 zombie
%Cpu(s): 88.7 us,  4.5 sy,  0.0 ni,  6.5 id,  0.0 wa,  0.0 hi,  0.1 si,  0.2 st
KiB Mem : 32472104 total, 10097316 free, 12913032 used,  9461756 buff/cache
KiB Swap:        0 total,        0 free,        0 used. 18543120 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
19558 root      20   0 6483180 1.213g 166324 S 130.6  3.9   3:27.78 gunicorn
19565 root      20   0 6342340 1.213g 166832 S 110.6  3.9   1:37.73 gunicorn
19735 root      20   0 7164760 0.978g 169844 R  98.3  3.2   3:23.91 python
19734 root      20   0 5921060 0.978g 172360 R  94.7  3.2   3:19.26 python
19557 root      20   0 6493100 1.283g 167148 S  84.4  4.1   5:07.13 gunicorn
19736 root      20   0 7886800 1.004g 172436 R  84.4  3.2   4:00.65 python
19733 root      20   0 7885236 1.005g 171612 R  81.4  3.2   6:12.67 python
19393 root      20   0 3078168 1.230g  37260 S  33.2  4.0   1:45.34 tensorflow_mode
19556 root      20   0 6217564 1.204g 166400 S  12.0  3.9   2:25.95 gunicorn
19465 root      20   0 24.605g 2.778g 688224 S   8.6  9.0   3:51.54 tensorflow_mode
24251 root      20   0 4006764 516212   6596 S   1.3  1.6   0:00.16 python3
18815 root      20   0  364408   3756   2316 S   0.7  0.0   0:00.54 docker-proxy
24247 root      20   0 4006764 516212   6596 S   0.7  1.6   0:00.13 python3
24248 root      20   0 4006764 516212   6596 S   0.7  1.6   0:00.13 python3
24249 root      20   0 4006764 516212   6596 S   0.7  1.6   0:00.13 python3
24250 root      20   0 4006764 516212   6596 S   0.7  1.6   0:00.14 python3
```

Next

This post is just a starting point to put together an "All-in-Docker AI demo" deployment stack as a testing framework. Next I hope to add in more API demo interfaces such as the Covid-19 ICU prediction interface ideally per FHIR R4 etc, and add in some support DICOM input format. This could also be a test bench to explore more closer integration with IRIS hosted ML capabilities. Over the time it can be used as a testing framework (and a pretty simple one) to intercept more and more ML or DL specialty models as we hike on various AI fronts including medical imaging, population health or personalised prediction, and NLP etc etc. I also listed a wish list at the very end of the previous post (in its "Next" section).

#Artificial Intelligence (AI) #Containerization #Continuous Delivery #Continuous Integration #IntegratedML #Machine Learning (ML) #InterSystems IRIS
Check the related application on InterSystems Open Exchange

Source URL:https://community.intersystems.com/post/deploy-mldl-models-consolidated-ai-demo-service-stack