

記事

[Toshihiko Minamoto](#) · 2022年2月14日 19m read

[Open Exchange](#)

統合AIデモサービススタックにML/DLモデルをデプロイする

キーワード: IRIS、IntegratedML、Flask、FastAPI、Tensorflow Serving、HAProxy、Docker、Covid-19

目的:

過去数か月に渡り、潜在的なICU入室を予測するための単純なCovid-19 X線画像分類器やCovid-19ラボ結果分類器など、ディープラーニングと機械学習の簡単なデモをいくつか見てきました。

また、ICU分類器のIntegratedMLデモ実装についても見てきました。「データサイエンス」の旅路はまだ続いています。しかし、「データエンジニアリング」の観点から、AIサービスデプロイメントを試す時期が来たかもしれません。これまでに見てきたことすべてを、一式のサービスAPIにまとめることはできるでしょうか。このようなサービススタックを最も単純なアプローチで達成するには、こういった一般的なツール、コンポーネント、およびインフラストラクチャを活用できるでしょうか。

対象範囲

対象:

ジャンプスタートとして、docker-composeを使用して、次のDocker化されたコンポーネントをAWS Ubuntuサーバーにデプロイできます。

- **HAProxy** - ロードバランサー
- **Gunicorn** と **Univorn** - Webゲートウェイ****サーバー
- **Flask** と **FastAPI** - WebアプリケーションUI、サービスAPI定義、およびヒートマップ生成などのアプリケーションサーバー
- **Tensorflow Model Serving** と **Tensorflow-GPU Model Serving** - 画像や分類などのアプリケーションバックエンドサーバー
- **IRIS IntegratedML** - SQL インターフェースを備えたアプリとデータベースを統合した AutoML
- **ベンチマーキング**用クライアントをエミュレートする、**Jupyterノートブック**のPython3
- Dockerと**docker-compose**
- Testla T4 GPU搭載の**AWS Ubuntu 16.04**

注意: GPUを使用したTensorflow Servingはデモのみを目的としています。GPU関連の画像（dockerfile）と構成（docker-compose.yml）は、単純にオフにできます。

対象外またはウィッシュリスト:

- **Nginx** または**Apache**などのWebサーバーは、今のところこのデモでは省略されています。
- **RabbitMQ**とRedis - IRISまたはEnsembleと置き換えられる、信頼性の高いメッセージングキューブローカー。
- **IAM** ([InterSystems API Manger](#)) または**Kong**はウィッシュリストに含まれます。
- **SAM** (InterSystemsの[システムアラートと監視](#))
- **Kubernetes** Operator付きの**ICM** ([InterSystems Cloud Manager](#)) - 誕生したときからずっとお気に入りの1つです。
- **FHIR** (Intesystems IRISベースのFHIR R4サーバーとFHIRアプリのSMART用FHIRサンドボックス)
- **CI/CD**開発ツールまたは**Github Actions**

「機械学習エンジニア」は、サービスのライフサイクルに沿って本番環境をプロビジョニングする際に、必然的にこれらのすべてのコンポーネントを操作しなければならないでしょう。
今後徐々に、焦点を当てていきたいと思います。

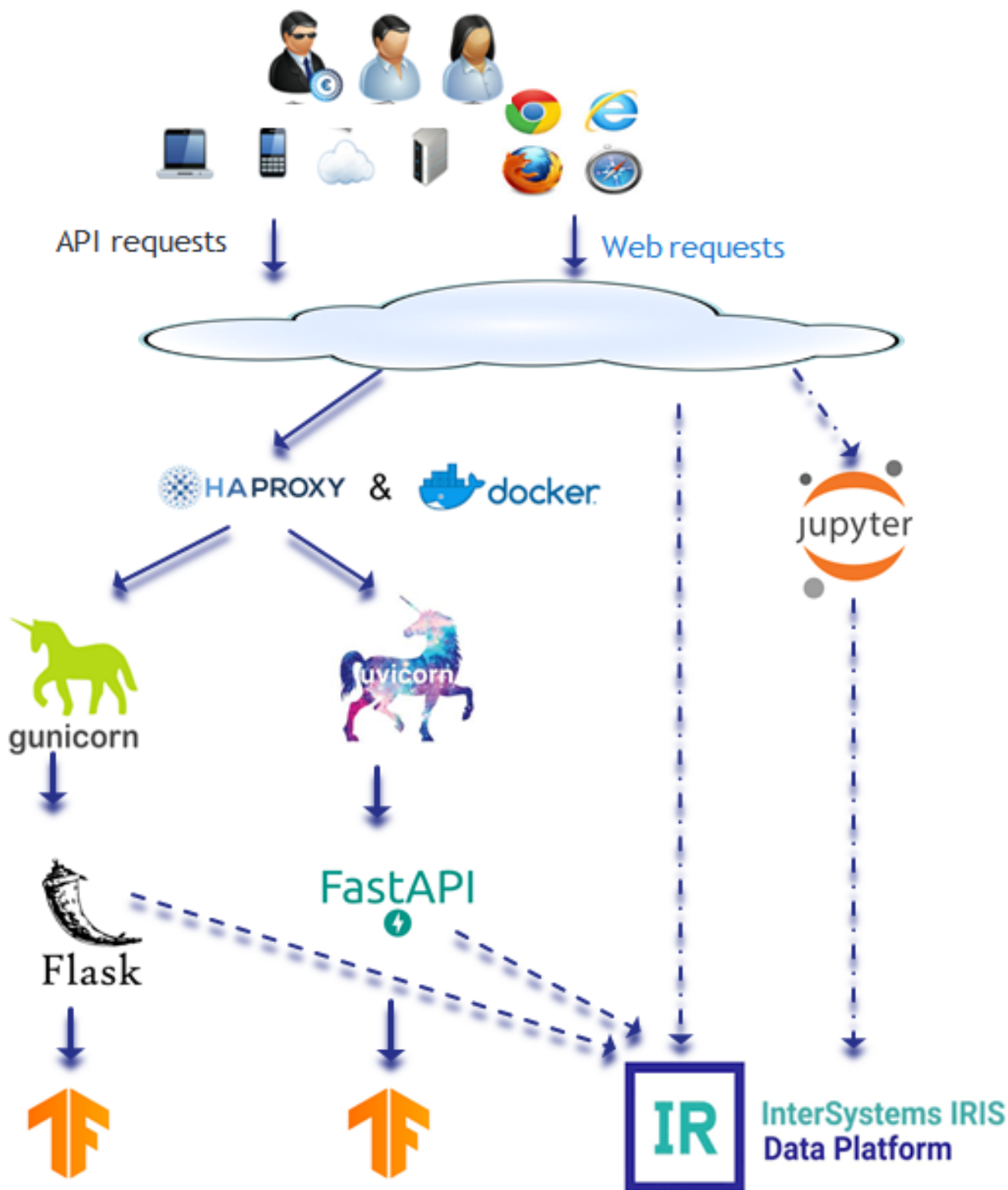
GitHubリポジトリ

全ソースコードの場所: <https://github.com/zhongli1990/covid-ai-demo-deployment>

また、新しいリポジトリとともに、[integratedML-demo-templateリポジトリ](#)も再利用します。

デプロイメントのパターン

以下に、この「DockerでのAIデモ」テストフレームワークの論理的なデプロイパターンを示します。



デモの目的により、意図的にディープラーニング分類とWebレンダリング用に個別のスタックを2つ作成し、HAPoxyをソフトロードバランサーとして使用して、受信するAPIリクエストをこれらの2つのスタックでステートレスに分散できるようにしました。

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

前の記事のICU入室予測と同様に、機械学習デモのサンプルには、IntegratedMLを使用したIRISを使用します。

現在のデモでは、本番サービスでは必要または検討される共通コンポーネントをいくつか省略しました。

- Webサーバー: NginxまたはApacheなど。HAProxyとGunicorn/Uvicornの間で、適切なHTTPセッション処理を行うために必要となります（DoS攻撃を防止するなど）。
- キューマネージャーとDB: RabbitMQやRedisなど。Flask/FastAPIとバックエンドサービングの間で、信頼性のある非同期サービングとデータ/構成の永続性などに使用されます。
- APIゲートウェイ: IAMまたはKongクラスター。単一障害点を作らないように、HAProxyロードバランサーとAPI管理用Webサーバーの間に使用されます。
- 監視とアラート: SAMが適切でしょう。
- CI/CD開発のプロビジョニング: クラウドニューラルデプロイメントと管理、およびその他の一般的な開発ツールによるCI/CDにはK8を使用したICMが必要です。

実際のところ

、IRIS自体は、信頼性の高いメッセージングのためのエンタープライズ級のキューマネージャーとしても、高性能データベースとしても確実に使用することができます。パターン分析では、IRISがRabbitMQ/Redis/MongoDBなどのキューブローカーとデータベースの代わりになることは明らかであるため、レイテンシが大幅に低く、より優れたスループットパフォーマンスとさらに統合する方が良いでしょう。さらに、IRIS Webゲートウェイ（旧CSPゲートウェイ）は、GunicornやUvicornなどの代わりに配置できますよね？

環境トポロジー

上記の論理パターンを全Dockerコンポーネントに実装するには、いくつかの一般的なオプションがあります。頭に思い浮かぶものとしては以下のものがあります。

- docker-compose
- docker swarmなど
- Kubernetesなど
- K8演算を使用したICM

このデモは、機能的なPoCといくつかのベンチマーキングを行うために、「docker-compose」から始めます。もちろん、いつかはK8やICMを使いたいとも考えています。

[docker-compose.yml](#) ファイルに説明されているように、AWS Ubuntuサーバーでの環境トポロジーの物理的な実装は、以下のようになります。

上図は、全Dockerインスタンスのサービスポート

が、デモの目的でどのようにマッピングされており、Ubuntuサーバーで直接公開されているかを示したものです。本番環境では、すべてセキュリティ強化される必要があります。また、純粋なデモの目的により、すべてのコンテナは同じDockerネットワークに接続されています。本番環境では、外部ルーティング可能と内部ルーティング不可能として分離されます。

Docker化コンポーネント

以下は、[docker-compose.yml](#) ファイルに示されるとおり、ホストマシン内でこれらの**ストレージボリューム**が各コンテナインスタンスにどのようにマウントされているかを示します。

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

./

```

??? covid19                                ?Flask+Gunicorn?????
Tensorflow Serving????????????????????
?   ???
    app.py                                ?Flask???????
:   Web?????????API????????????????????????????????????
?   ??? covid19_models                    ?CPU???????Tensorflow Serving???????
Tensorflow????????????????????????????
?   ??? Dockerfile                        ?Flask   ?????Gunicorn:
CMD ["gunicorn", "app:app", "--bind", "0.0.0.0:5000", "--workers", "4", "--threads",
"2"]?
?   ??? models                            ?.h5???Flask?????????????????Grad-
CAM?????????????????API???
?   ??? __pycache__
?   ??? README.md
?   ??? requirements.txt                  ??Flask+Gunicorn???????Python???????
?   ??? scripts
?   ??? static                            ?Web?????????
?   ??? templates                        ?Web?????????????????
?   ??? tensorflow_serving                ?TensorFlow Serving????????????????
?   ??? test_images
??? covid-fastapi                          (FastAPI+Uvicorn?????)
GPU???Tensorflow Serving????????????????????
?   ??? covid19_models                    ????????
Tensorflow Serving GPU????????????????????????????
?   ??? Dockerfile                        ?Uvicorn+FastAPI
????????????????????:

?
?   ??? main.py                          ?FastAPI???
:   Web?????????API????????????????????????????????????
?   ??? models                            ?.h5???FastAPI?????????????????Grad-
CAM?????????????????API???
?   ??? __pycache__
?   ??? README.md
?   ??? requirements.txt
?   ??? scripts
?   ??? static
?   ??? templates
?   ??? tensorflow_serving
?   ??? test_images
??? docker-
compose.yml      ?
???????Docker??????? Docker GPU?nvidia runtime???????2.3?????????????????3.x?????
??? haproxy      (HAProxy
docker????????????????? ??: ??????LB???????????????????????????? )

?   ??? Dockerfile
?   ??? haproxy.cfg
???
    notebooks
        ?TensorFlow 2.2?TensorBoard?????Jupyter????????????????
        ??? Dockerfile
        ??? notebooks                    ??????????API????????????????????
?????????????????????????????????Python???API????????????????????????????
??? requirements.txt

```

注意: 上記の[docker-compose.yml](#)はCovid-19 X線画像のディープラーニングデモ用です。
別の[integratedML-demo-template](#)の[docker-compose.yml](#)

とともに、環境トポロジに表示されるフルサービススタックを形成するために使用されています。

サービスの起動

すべてのコンテナサービスを起動するには、単純な**docker-compose up -d**を使用します。

```
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-masteririsimlsvr1
6a0f22ad3ffc   haproxy:0.0.1                      "/docker-entrypoint.…"  8 weeks ago   Up 2 days    0.0.0.0:8088->8088/tcp   flask-xraylb1
71b5163d8960   ai-service-fastapi:0.2.0           "uvicorn main:app -…"   8 weeks ago   Up 2 days    0.0.0.0:8056->8000/tcp   flask-xrayfastapi1
400e1d6c0f69   tensorflow/serving:latest-gpu      "/usr/bin/tfserving…"  8 weeks ago   Up 2 days    0.0.0.0:8520->8500/tcp, 0.0.0.0:8521->8501/tcp   flask-xraytf2svg11
eaac88e9b1a7   ai-service-flask:0.1.0             "gunicorn app:app -…"   8 weeks ago   Up 2 days    0.0.0.0:8051->5000/tcp   flask-xrayflask1
e07ccd30a32b   tensorflow/serving                 "/usr/bin/tfserving…"  8 weeks ago   Up 2 days    0.0.0.0:8511->8501/tcp   flask-xraytf2svg11
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-xraytf2jpt1
88e8709404ac   tf2-jupyter-jdbc:1.0.0-impl-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-masterf2jupyter1
```

docker-compose up --scale fastapi=2 --scale flask=2 -d for example will horizontally scale up to 2x Unicorn+Flask containers and 2x Unicorn+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-xrayfastapi2
95bcd8535aa6   ai-service-flask:0.1.0             "gunicorn app:app -…"   4 minutes ago   Up 4 minutes  0.0.0.0:8052->5000/tcp   flask-xrayflask2
```

... ..

「IntegratedML-demo-template」の作業ディレクトリで別の「docker-compose up -d」を実行することで、上記リストのirisimlsvrとtf2jupyterコンテナが起動されています。

テスト

1. 単純なUIを備えたAIデモWebアプリ

上記のdockerサービスを起動したら、一時アドレス<http://ec2-18-134-16-118.eu-west-2.compute.amazonaws.com:8056/>のAWS

EC2インスタンスにホストされている[Covid-19に感染した胸部X線画像の検出](#)用デモWebアプリにアクセスできます。

以下は、私の携帯でキャプチャした画面です。デモUIは非常に単純です。基本的に「Choose File」をクリックして「Submit」ボタンを押せば、[X線画像](#)がアップロードされて分類レポートが表示されます。Covid-19

感染のX線画像と

して分類されたら、DLによって「検

出された」病変領域をエミュレートする[ヒートマップが表示](#)

されます。そうでない場合は、分類レポートにはアップロードされたX線画像のみが表示されます。

このWebアプリはPythonサーバーページです。このロジックは主に[FastAPI's main.py](#)ファイルと[Flask's app.py](#)ファイルにコーディングされています。

もう少し時間に余裕がある際には、FlaskとFastAPIのコーディングと規則の違いを詳しく説明かもしれません。実は、AIデモホスティングについて、FlaskとFastAPIとIRISの比較を行いたいと思っています。

2. テストデモAPI

FastAPI（ポート8056で公開）には、以下に示すSwagger APIドキュメントが組み込まれています。これは非常に便利です。以下のようにURLに「/docs」を指定するだけで、利用することができます。

いくつかのプレースホルダー（/helloや/itemsなど）と実際のデモAPIインターフェース（/healthcheck、/predict、predict/heatmapなど）を組み込みました。

これらのAPIに簡単なテストを実行してみましょう。

[私がこのAIデモサービス用に作成したJupyterノートブックサンプル](#)

の1つで、Python行を（APIクライアントアプリエミュレーターとして）いくつか実行します。

以下では、例としてこちらのファイルを実行しています。 <https://github.com/zhongli1990/covid-ai-demo-deployment/blob/master/notebooks/notebooks/Covid19-3class-Heatmap-Flask-FastAPI-TF-serving-all-in-one-HAProxy2.ipynb>

まず、バックエンドのTF-Serving（ポート8511）とTF-Serving-GPU（ポート8521）が稼働していることをテストします。

```
!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": ""
      }
    }
  ]
}
```

次に、以下のサービスAPIが稼働していることをテストします。

Gunicorn+Flask+TF-Serving

Unicorn+FastAPI+TF-Serving-GPU

上記の両サービスの手間にあるHAProxyロードバランサー

```
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)
```

そして、以下のような結果が期待されます。

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

入力X線画像の分類とヒートマップ結果を返す、**/predict/heatmap**

などの機能的なAPIインターフェースをテストします。受信する画像は、API定義に従ってHTTP POST経由で送信される前にbase64にエンコードされています。

%%time

```
# ??????????????????
import argparse
import base64

import requests

# 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 = ""
# JPG?PNG???????base64?????????
with open(image_path, "rb") as imageFile:
    b64_image = base64.b64encode(imageFile.read())

# API?????????
data = {'b64': b64_image}

# POST????????????????????????????????????
r = requests.post(url=API_ENDPOINT, data=data)

print(r.status_code, r.text)

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

このようなすべての[テスト画像もGitHubにアップロード済み](#)です。
上記のコードの結果は以下のようになります。

```
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. デモサービスAPIをベンチマークテストする

HAProxyロードバランサーインスタンスをセットアップします。
また、Flaskサービスを4個のワーカーで開始し、FastAPIサービスも4個のワーカーで開始しました。

ノートブックファイルで直接8個のPythonプロセスを作成し、8個の同時APIクライアントがデモサービスAPIにリクエストを送信する様子をエミュレートしてみてもいいでしょう。

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

start = time.time()

#loadbalancer:
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:
        # ??????
        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:
    # ????????????
    # ??????????????????????????????
    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))
```

したがって、8x27 = 216個のテスト画像を処理するのに74秒かかっています。これは負荷分散されたデモスタックが、（分類とヒートマップの結果をクライアントに返すことで）1秒間に3個の画像を処理できています。

--- 74.37691688537598 seconds ---

PuttyセッションのTopコマンドから、上記のベンチマークスクリプトが実行開始されるとすぐに8個のサーバープロセス（4個のGunicornと4個のUvicorn/Python）がランプアップし始めたことがわかります。

今後の内容

この記事は、テストフレームワークとして「全DockerによるAIデモ」のデプロイメントスタックをまとめるための出発点に過ぎません。次は、Covid-19

ICU入室予測インターフェースなどさらに多くのAPIデモインターフェースを理想としてはFHIR

R4などによって追加し、サポートDICOM入力形式を追加したいと思います。

これは、IRISがホストするML機能とのより緊密な統合を検討するためのテストベンチになる可能性もあります。

医用画像、公衆衛生、またはパーソナリゼされた予測やNLPなど、さまざまなAIフロントを見ていく過程で、徐々に、より多くのMLまたはDL特殊モデルを傍受するためのテストフレームワーク（非常に単純なテストフレーム）として使用していただけるでしょう。

ウィッシュリストは、[前の投稿（「今後の内容」セクション](#)）の最後にも掲載しています。

[#Artificial Intelligence \(AI\)](#) [#IntegratedML](#) [#コンテナ化](#) [#Machine Learning \(ML\)](#) [#継続的インテグレーション](#)

[#継続的デリバリー](#) [#開発者コミュニティ公式](#)

[InterSystems Open Exchange](#)で関連アプリケーションを確認してください

ソースURL:

<https://jp.community.intersystems.com/post/%E7%B5%B1%E5%90%88ai%E3%83%87%E3%83%A2%E3%82%B5%E3%83%BC%E3%83%93%E3%82%B9%E3%82%B9%E3%82%BF%E3%83%83%E3%82%AF%E3%81%ABmldl%E3%83%A2%E3%83%87%E3%83%AB%E3%82%92%E3%83%87%E3%83%97%E3%83%AD%E3%82%A4%E3%81%99%E3%82%8B>