



Data Science Solution Implementation

Dave Ho

dave.ho@nus.edu.sg



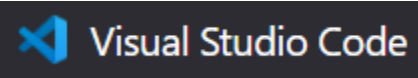
Agenda

1. Deploy Model as an API with Plumber
2. Deploy Model as an API with FastAPI
3. Deploy a Machine Learning application with Streamlit

Deploy Model as an API with Plumber



Ensure the following installations and setup:

- R and RStudio 
- Docker Desktop 
- Visual Studio Code 

Learning Milestones

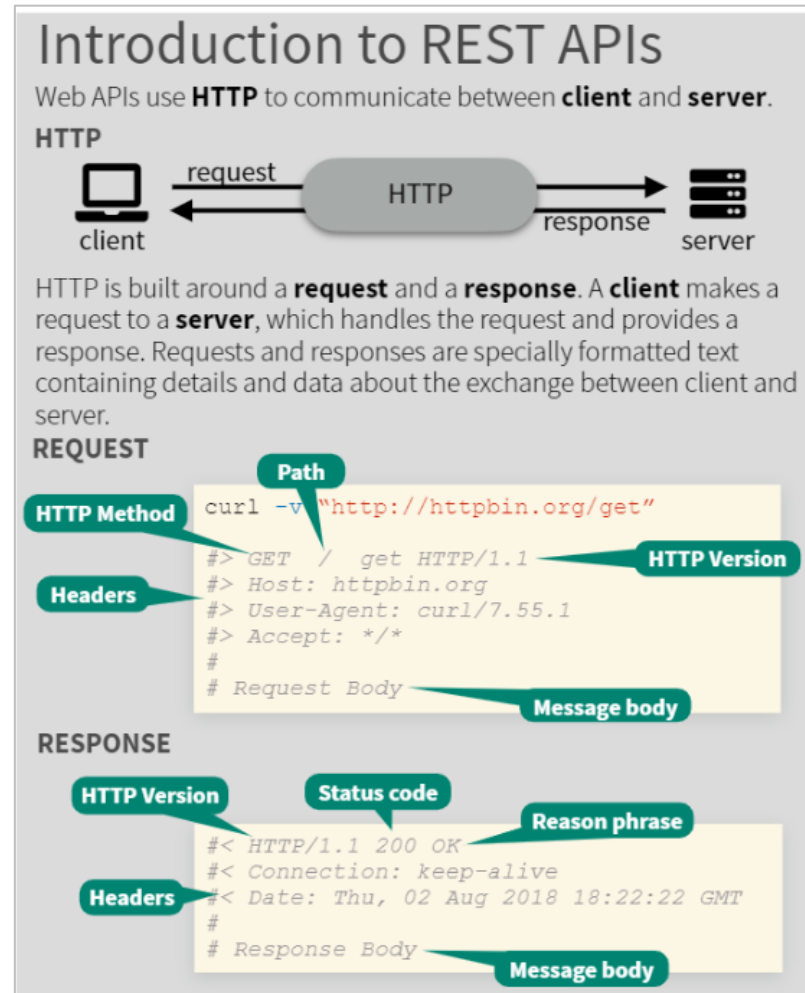
You will learn how to:

1. Persist a ML model.
2. Create an API service for the model.
3. Serve model predictions from a Docker container.

Primer: RESTful APIs

Application programming interfaces that enable applications to exchange information securely over the internet.

Source: Posit. 2023. *REST APIs with plumber::CHEATSHEET*.
<https://github.com/rstudio/cheatsheets/blob/main/plumber.pdf>



Primer: Plumber

Plumber: Build APIs with R

Plumber uses special comments to turn any arbitrary R code into API endpoints. The example below defines a function that takes the `msg` argument and returns it embedded in additional text.

Plumber
comments
begin with `#*`

@ decorators
define API
characteristics

HTTP Method

`/<path>` is used to
define the location
of the endpoint

```
library(plumber)

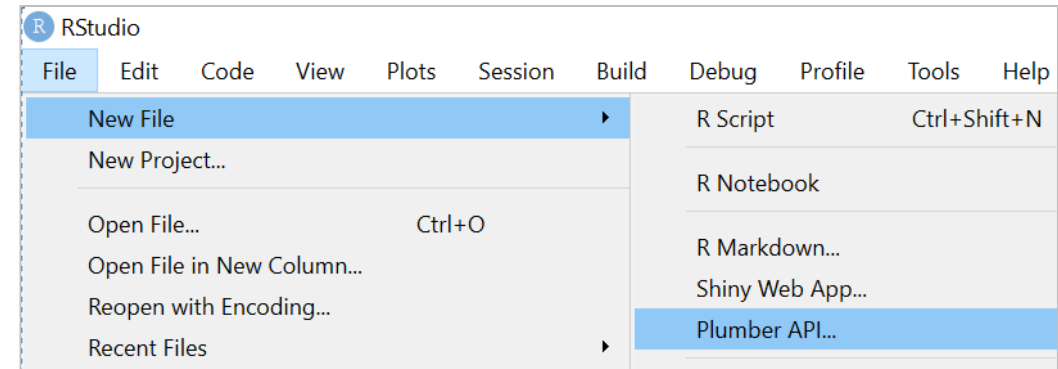
#* @apiTitle Plumber Example API

#* Echo back the input
#* @param msg The message to echo
#* @get /echo
function(msg = "") {
  list(
    msg = paste0(
      "The message is: ", msg, ""
    )
  )
}
```

Source: Posit. 2023. *REST APIs with plumber::CHEATSHEET*.
<https://github.com/rstudio/cheatsheets/blob/main/plumber.pdf>

Try Plumber

1. Launch RStudio.
2. Create a plumber file.
3. Create a function that takes a pair of random number sets (`rnorm(100)`) to generate a scatterplot at API endpoint `/scatter`.
4. Create a function that takes weight and height as inputs to compute bmi at API endpoint `/bmi`.



Solution

GET is used to request data from a specified resource.

Develop API

```
## Plot a scatter plot
## @serializer jpeg
## GET is used to request data from a specified resource.
## @get /scatter
function(){
  rand = rnorm(100)
  rand2 = rnorm(100)
  plot(rand, rand2, main='Scatter Plot')
}

## compute and return BMI given weight and height
## @param ht height of a person in meters
## @param wt weight of a person in kilograms
## @post /bmi
function(ht, wt){
  as.numeric(wt)/(as.numeric(ht)**2)
}
```

POST is used to send (large) data in the request body to a server to create/update a resource.

Develop API

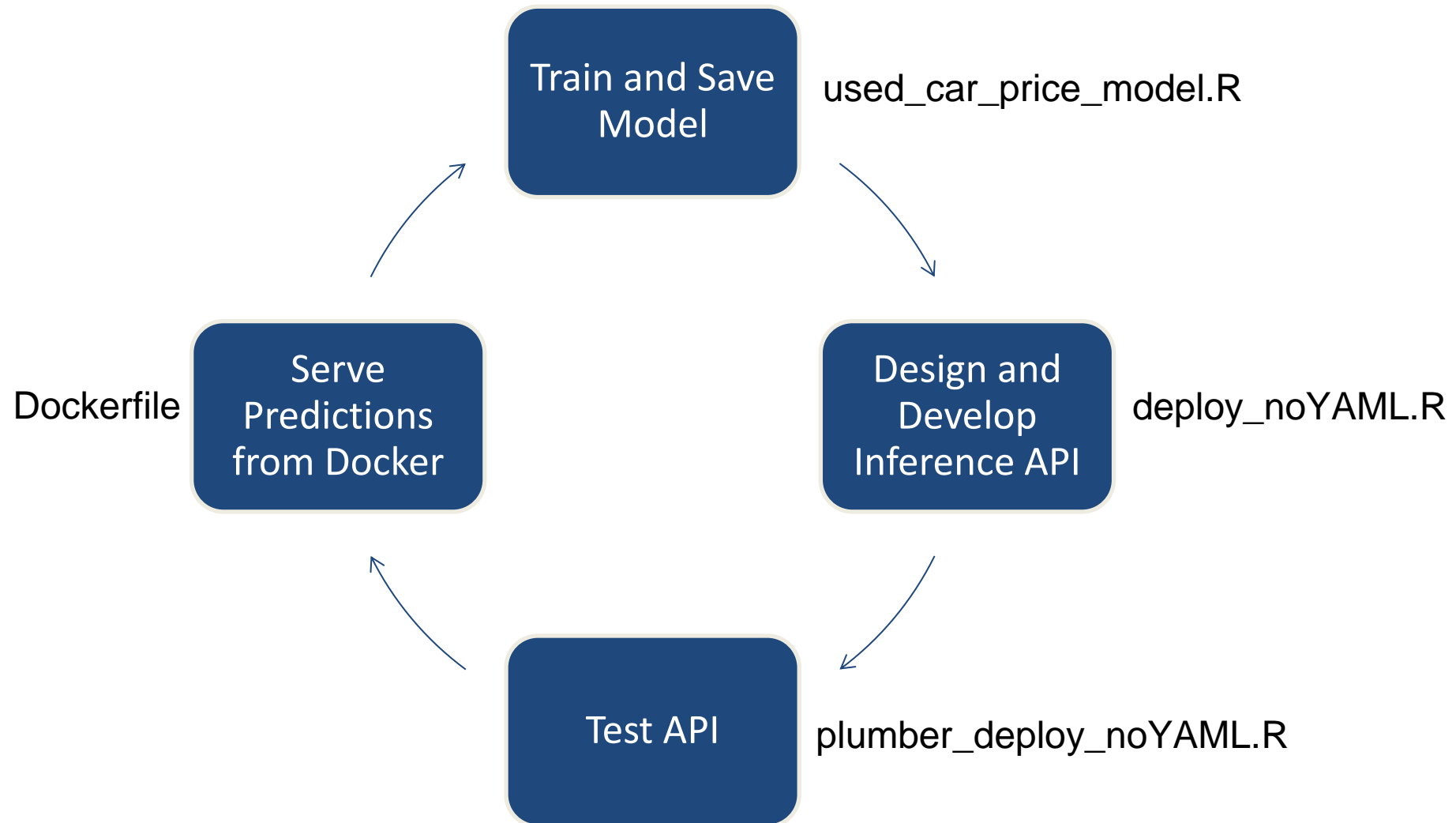
Call API

```
> library(httr)
> headers = c('Content-Type'='application/json')
> data = '{"ht":1.78,"wt":75}'
> url = 'http://127.0.0.1:7922/bmi'
> resp = httr::POST(url=url,httr::add_headers(.headers=headers), body=data)
> content(resp)
[[1]]
[1] 23.6713
```

Call API

```
D:\workspace\R>curl -X POST -H "Content-Type: application/json" -d "{\"ht\":1.78,\"wt\":74}" http://127.0.0.1:4325/bmi
[23.3556]
```

Process Flow



Step 1: Train and Save Model

1. Train a model that predicts the price of a used car.
2. Persist trained model as a RDS (R Data Serialisation) file.

Sample training script: *used_car_price_model.R*

```
# set path to folder containing the source codes
setwd("D:/Workspace/R/dssi-plumber")

# read the data file & inspect it's structure
cars = read.csv('data/cars.csv')
summary(cars)

#set initial seed for reproducibility
set.seed(123)

# train-test split
inds = createDataPartition(cars$Price, p=0.7, list=FALSE, times=1)
train_set = cars[inds,]
test_set = cars[-inds,]
```

```
# train model
model = lm(Price ~. , data = train_set)
# feature select
final_model = stepAIC(model)
summary(final_model)
```

Train model

```
# save final model
saveRDS(final_model, "models/car_price.rds")
```

Persist trained model

Step 2: Design and Develop Inference API

1. Using Plumber, create a function that takes features/covariates as input parameters, loads the trained model and output predicted price at API endpoint '/api/v1/used-car/price'.

Sample Plumber file: *deploy_noYAML.R*

```
model = readRDS("../models/car_price.rds")
```

Load trained model

```
## Model: Price ~ Age + KM + FuelType + HP + Automatic + CC + Weight
```

```
## predict the price of a used car
```

```
##
```

```
## @param Age this is the age of the car as input by the user
```

```
## @param KM distance ran in kms
```

```
## @param FuelType takes one of the 3 values out of Diesel/Petrol/CNG
```

```
## @param HP horsepower
```

```
## @param Automatic 1 or 0
```

```
## @param CC vehicle fuel capacity
```

```
## @param Doors 2/3/4/5 doors
```

```
## @param Weight weight of vehicle
```

API parameters

```
## @post /api/v1/used-car/price
```

API endpoint

```
function(Age, KM, FuelType, HP, Automatic, CC, Doors, Weight){  
  newdata = data.frame(Age = as.numeric(Age), KM = as.numeric(KM),  
                        FuelType = FuelType, HP = as.numeric(HP),  
                        Automatic = as.numeric(Automatic),  
                        CC = as.numeric(CC), Doors = as.numeric(Doors),  
                        Weight = as.numeric(Weight))  
  
  # predict car price using predict function in R  
  return(list(price = jsonlite::unbox(predict(model, newdata))))  
}
```

*API function:
Perform
prediction on
new inputs*

Step 3: Test API

1. Create a wrapper R script that runs the Plumber API with a user-defined port.
2. Run the wrapper script.
3. In Swagger UI, fill in the parameter values and test the '/api/v1/used-car/price' API response.

Sample wrapper script: *plumber_deploy_noYAML.R*

POST /api/v1/used-car/price predict the price of a used car

Parameters Cancel

Name	Description
Age * required string (query)	this is the age of the car as input by the user <input type="text" value="Age - this is the age of the car as input by the"/>
KM * required string (query)	distance ran in kms <input type="text" value="KM - distance ran in kms"/>
FuelType * required string (query)	takes one of the 3 values out of Diesel/Petrol/CNG <input type="text" value="FuelType - takes one of the 3 values out of D"/>
HP * required string (query)	horsepower <input type="text" value="HP - horsepower"/>
Automatic * required string (query)	1 or 0 <input type="text" value="Automatic - 1 or 0"/>
CC * required string (query)	vehicle fuel capacity <input type="text" value="CC - vehicle fuel capacity"/>
Doors * required string (query)	2/3/4/5 doors <input type="text" value="Doors - 2/3/4/5 doors"/>
Weight * required string (query)	weight of vehicle <input type="text" value="Weight - weight of vehicle"/>

Execute

Step 4: Serve Predictions from Docker

1. Start Docker Engine.
2. Create a Dockerfile that uses a Plumber API server image.
3. Specify the instructions to copy source files and running Plumber file.
4. Build the Docker image:

```
> docker build -t dssi-plumber-docker:1.0 .
```

5. Start the application container:

```
> docker run --rm -d -p 80:8000/tcp dssi-plumber-docker:1.0
```

```
FROM trestletech/plumber

# copy necessary files
RUN mkdir /models
RUN mkdir /src
COPY models/car_price.rds /models/car_price.rds
COPY src/deploy_noYAML.R /src/plumber.R
WORKDIR /src

# plumber instructions to run
EXPOSE 8000
ENTRYPOINT ["R", "-e", "pr <- plumber::plumb('/src/plumber.R'); \pr$run(host='0.0.0.0', port=8000)"]
```

Sample Dockerfile: *Dockerfile*

Deploy Model as an API with FastAPI



Ensure the following installations and setup:

- Docker Desktop 
- Anaconda/Python Software 
- Python virtual environment [Recommended]
- Install additional Python packages:

```
> pip install -r requirements.txt
```

Learning Milestones

You will learn how to:

1. Persist a ML model.
2. Create an API service for the model.
3. Serve model predictions from a Docker container.

Primer: FastAPI

FastAPI - Web framework for building APIs with Python

```
from fastapi import FastAPI
```

Import FastAPI

```
app = FastAPI()
```

Create a FastAPI instance

```
@app.get("/")
```

Path operation decorator

```
async def root():
```

Path operation function

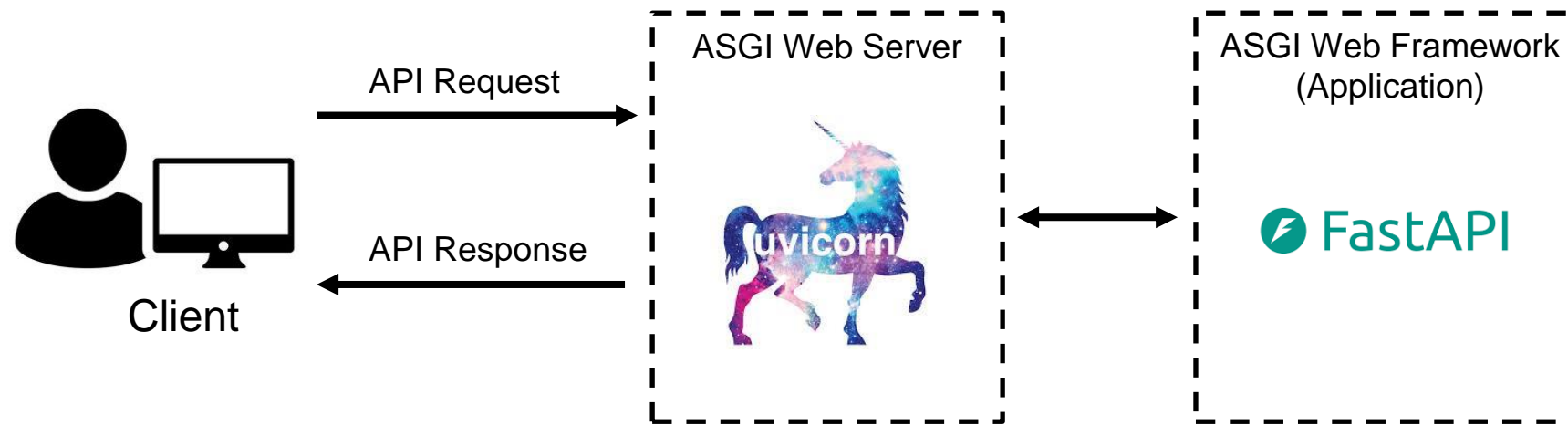
```
    return {"message": "Hello World"}
```

API Response

Source: <https://fastapi.tiangolo.com/tutorial/first-steps/>

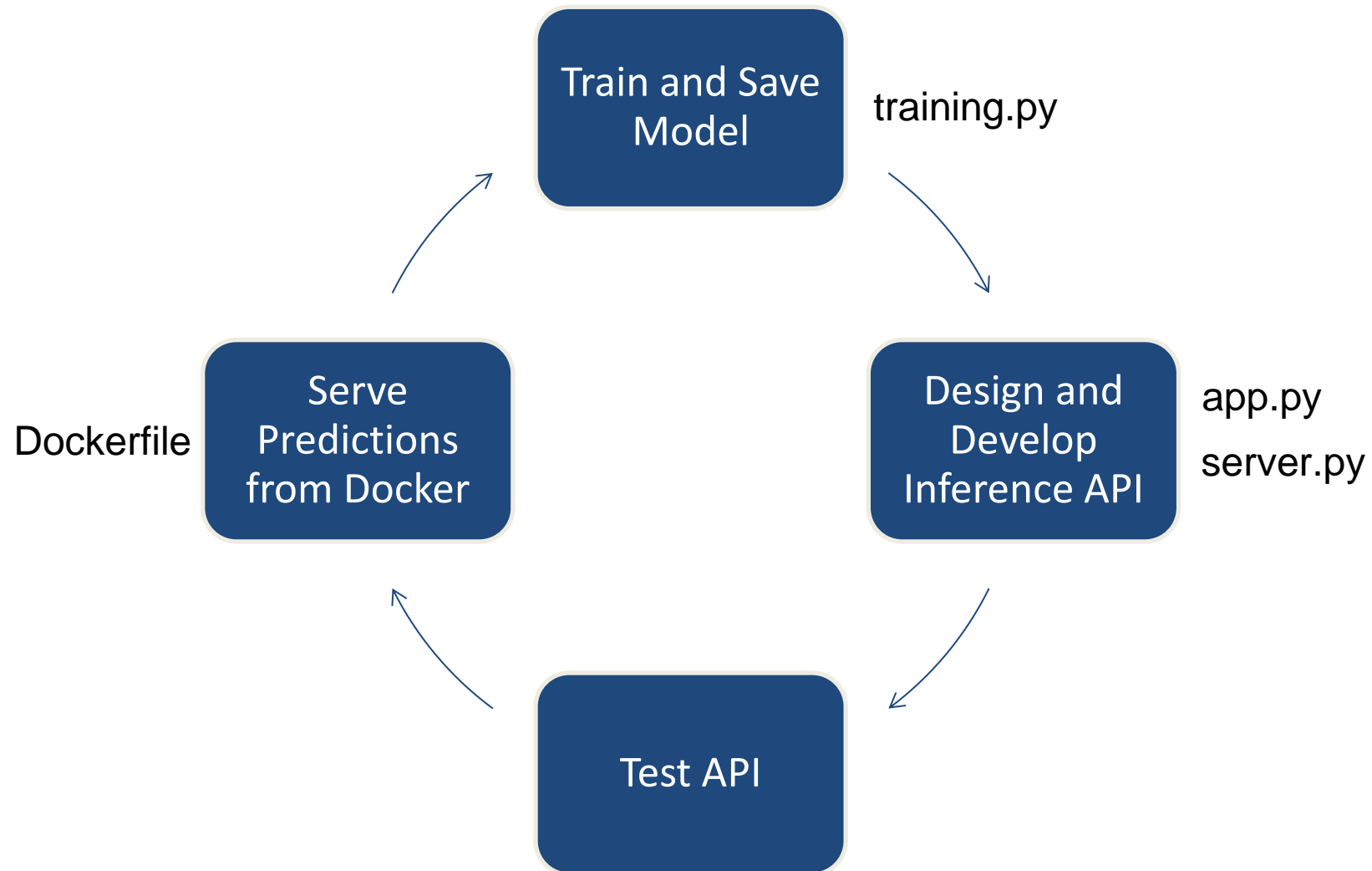
Primer: FastAPI

FastAPI - Web framework for building APIs with Python



Source: <https://www.uvicorn.org/>

Process Flow



Step 1: Train and Save Model

1. Run the training module to train and persist model:

```
> python -m src.training --data_path data/cars.csv --r2_criteria 0.8
```

Sample training script: *training.py*

Step 2: Design and Develop Inference API

1. Using FastAPI, develop an API that takes new data for inference as input and outputs predicted value at endpoint '/api/v1/used-car/price'.
2. Implement ASGI server using Uvicorn for the APIs.

Sample application code: *app.py*

Sample server code: *server.py*

```
from fastapi import FastAPI
from fastapi.responses import JSONResponse
import pandas as pd
from src.model_registry import retrieve
from src.config import appconfig

app = FastAPI()

model, features = retrieve(appconfig['Model']['name'])

@app.get("/")
def home():
    return {"message": "Welcome to DSSI!"}

@app.post(appconfig['API']['used_car_price'])
def predict(data: dict):
    pred_df = pd.DataFrame.from_dict([data])
    pred = model.predict(pred_df[features])
    return {"price": pred[0]}
```

```
import uvicorn
import src

if __name__ == '__main__':
    uvicorn.run("src.app:app", host="0.0.0.0", port=8000)
```

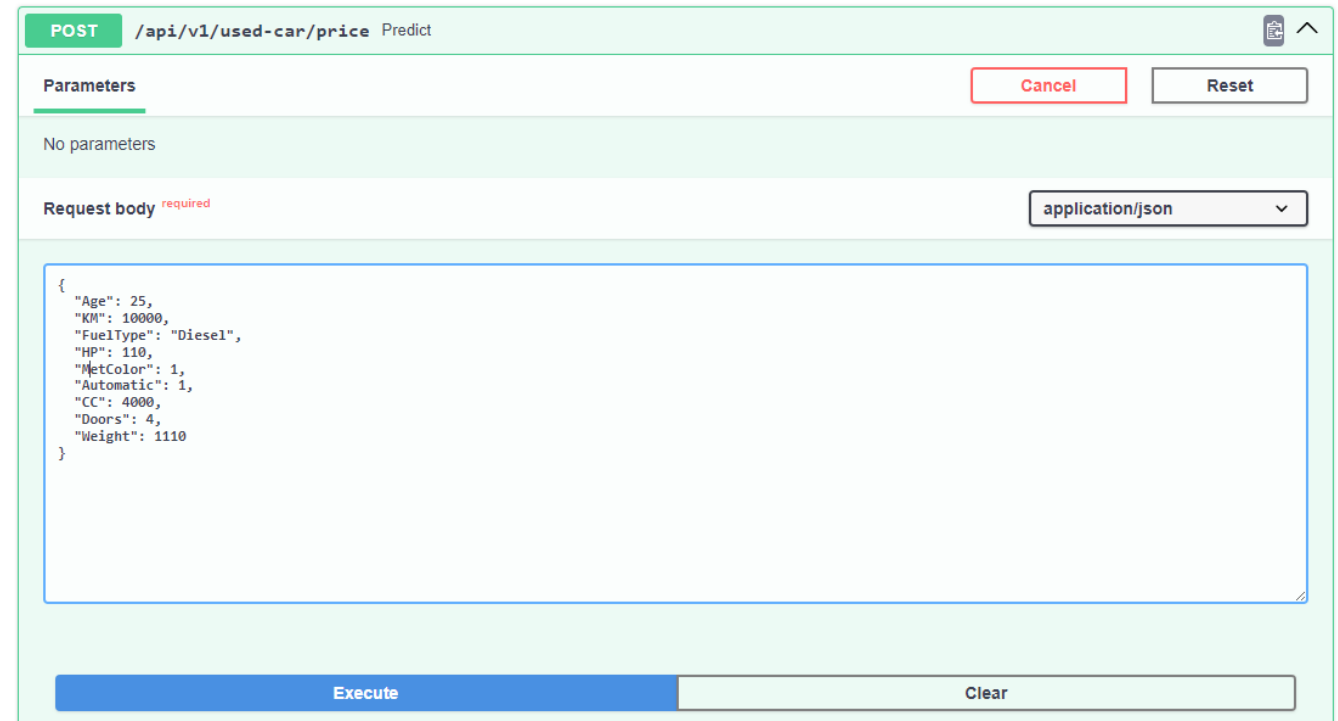
Process Flow

Step 3: Test API

1. Start API service:

```
> python server.py
```

2. Use Swagger UI (<http://127.0.0.1:8000/docs>) to test the response of the API.



POST /api/v1/used-car/price Predict

Parameters Cancel Reset

No parameters

Request body required application/json

```
{
  "Age": 25,
  "KM": 10000,
  "FuelType": "Diesel",
  "HP": 110,
  "MetColor": 1,
  "Automatic": 1,
  "CC": 4000,
  "Doors": 4,
  "Weight": 1110
}
```

Execute Clear

Step 4: Serve Predictions from Docker

1. Start Docker Engine.
2. Create a Dockerfile that specifies the instructions to install required packages, copy source files and run the API service.
3. Build the Docker image:

```
> docker build -t dssi-fastapi-docker:1.0 .
```

4. Start the application container:

```
> docker run --rm -d -p 80:8000 dssi-fastapi-docker:1.0
```

```
FROM python:3.9

WORKDIR /dssi

COPY ./requirements.txt /dssi/requirements.txt

RUN pip install --no-cache-dir --upgrade -r /dssi/requirements.txt

COPY ./server.py ./server.py
COPY ./src ./src
COPY ./models ./models
COPY ./metadata ./metadata

CMD ["python", "server.py"]
```

Sample Dockerfile: *Dockerfile*

Access API

Access the deployed R mode
via API call and perform
exploration of predictions.

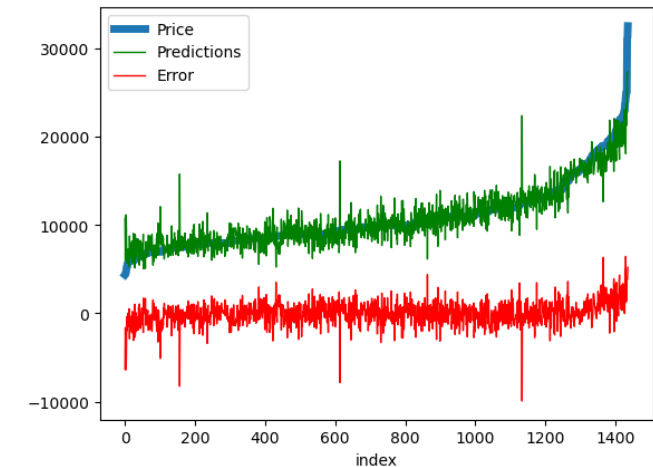
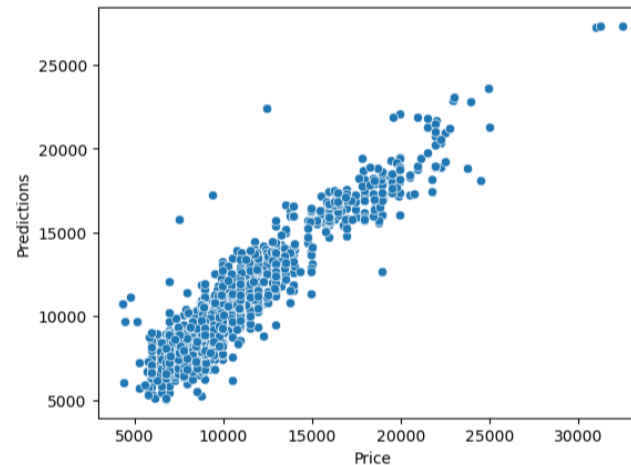
Sample Python notebook:
DSSI_TestAPI.ipynb

```
In [1]: import requests
import pandas as pd

new_data = {
    "Age": 25,
    "KM": 10000,
    "FuelType": "Diesel",
    "HP": 110,
    "MetColor": 1,
    "Automatic": 1,
    "CC": 4000,
    "Doors": 4,
    "Weight": 1110
}

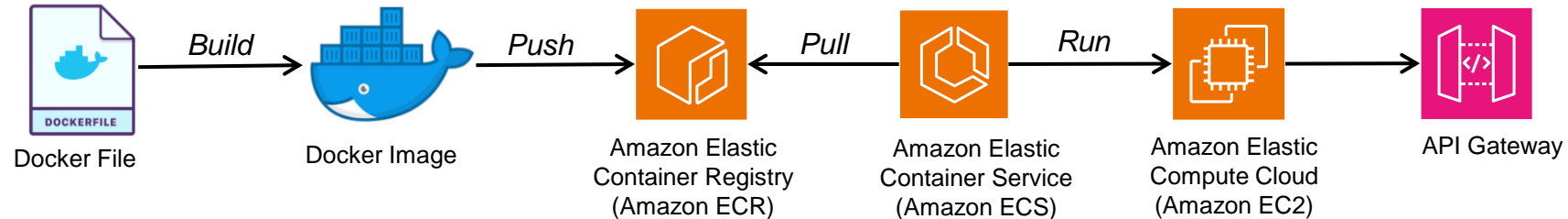
In [2]: resp = requests.post("http://127.0.0.1/api/v1/usedcar/predict", json = new_data)
print("Predicted Price: ", resp.json()["price"])

Predicted Price: 8878.829091562038
```

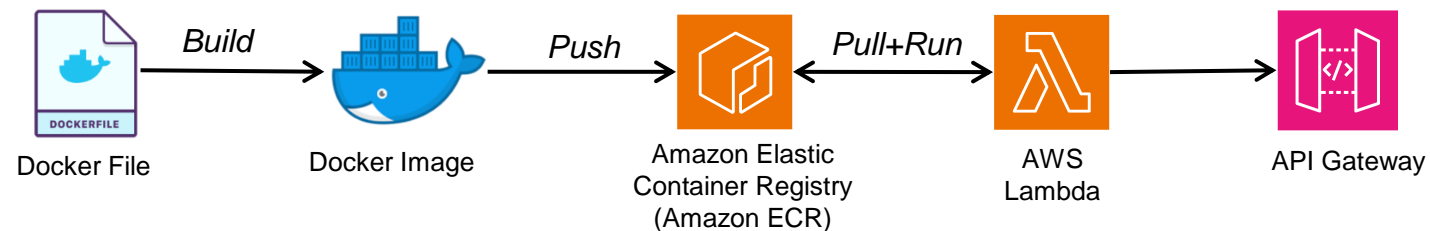
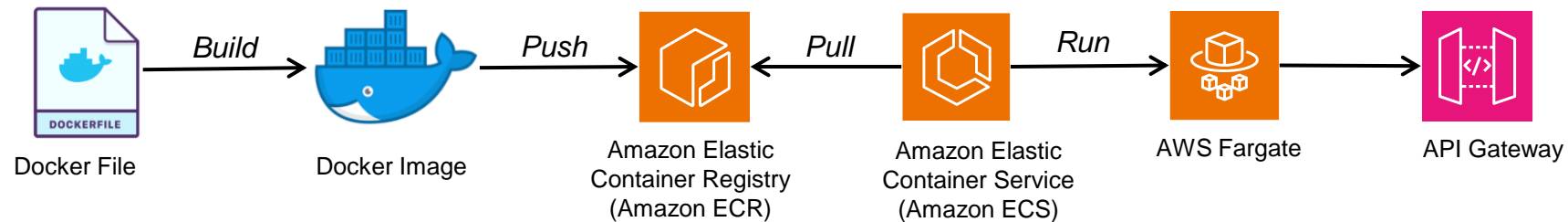


Deploy on AWS

Stateful Deployment (Large consistent workload)





Stateless Deployment (On-demand workload)



Deploy a Machine Learning Application with Streamlit



Ensure the following installations and setup:

- GitHub Account 
- Git and GitHub CLI
- Anaconda/Python Software 
- Python virtual environment [Recommended]
- Install additional Python packages:

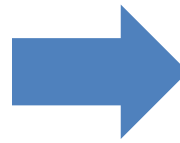
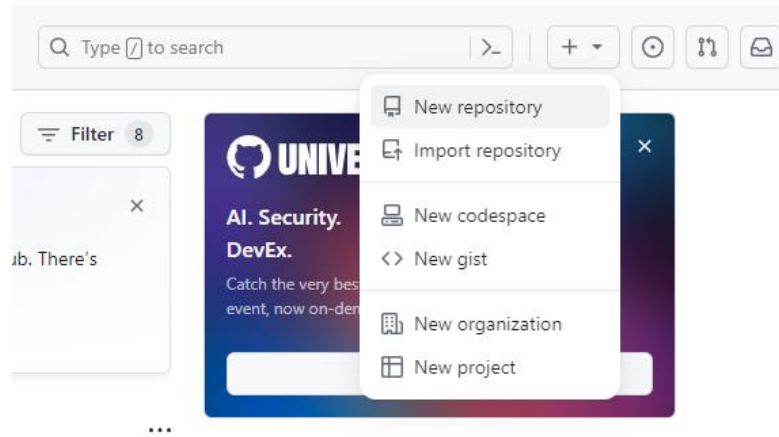
```
> pip install -r requirements.txt
```

Learning Milestones

You will learn how to:

1. Create a web application that uses a trained ML model to automate decisions.
2. Deploy the web application online.

Step 1: Create a public repository on GitHub



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Required fields are marked with an asterisk (*).

Owner * / Repository name *
✓ dssi-py is available.

Great repository names are short and memorable. Need inspiration? How about [ubiquitous-rotary-phone](#) ?

Description (optional)

☒ **Public**
Anyone on the internet can see this repository. You choose who can commit.

☐ **Private**
You choose who can see and commit to this repository.

Initialize this repository with:
☐ **Add a README file**
This is where you can write a long description for your project. [Learn more about READMEs.](#)

Add .gitignore
.gitignore template:

Choose which files not to track from a list of templates. [Learn more about ignoring files.](#)

Choose a license
License:

A license tells others what they can and can't do with your code. [Learn more about licenses.](#)

ⓘ You are creating a public repository in your personal account.

Create repository

Step 2: Push remote repository to GitHub

1. Download and unzip the source code in local directory.
2. Open command prompt (or terminal) and navigate to the directory containing the unzipped source codes.
3. Authentication setup with GitHub (via HTTPS):

```
> gh auth login
```

4. Push repository to GitHub:

```
> git init
> git add Include all the modified contents in the current directory
> git commit -m "first commit" Direct copy-pasting of quotation marks can cause formatting issues in multi-lingual terminals
> git branch -M main
> git remote add origin https://github.com/daveho81/dssi-py.git Change to your own repository URL
> git push -u origin main
```

Try Streamlit

1. Display a dataframe table.

```
# Load diabetes dataset
st.subheader('**Diabetes Data**')
db = datasets.load_diabetes()

df = pd.DataFrame(db.data, columns=db.feature_names)

# Display dataframe as an interactive table
st.dataframe(df, use_container_width=True)
```

2. Run Streamlit application locally.

```
> streamlit run toy-app.py
```

Sample application code: *toy-app.py*

Try Streamlit

3. Add a plot.

```
# Plot histogram for age of patients
fig, ax = plt.subplots(figsize=(6, 3))
if 1==0: # Evaluate True to show plot
    df['age'].hist(bins = 10, ax=ax)
    fig.suptitle("Age Distribution")
    st.pyplot(fig)
```

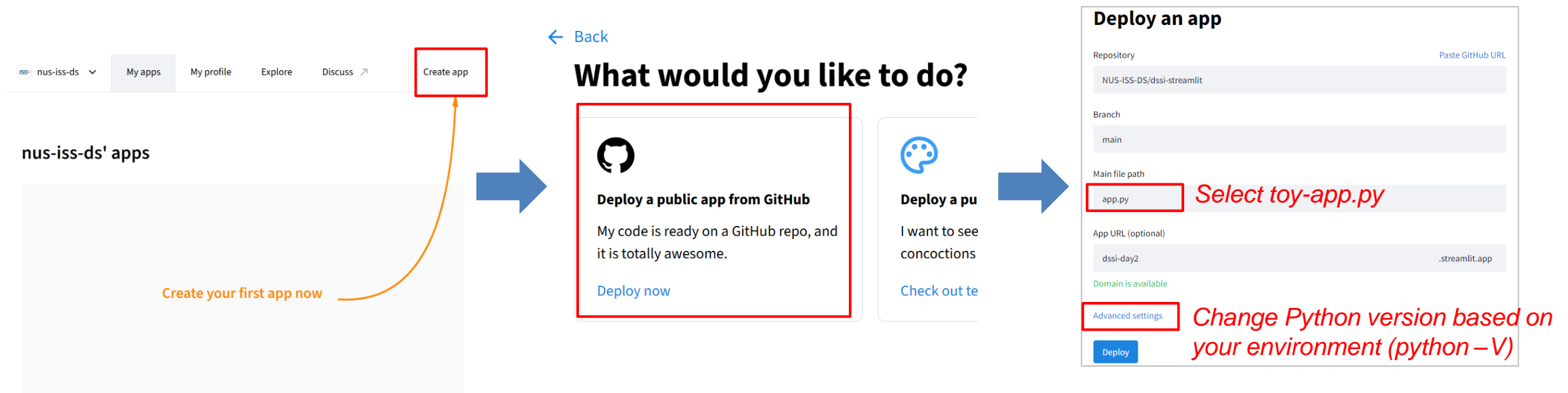
4. Ensure requirements.txt specifies required packages.

5. Commit changes to GitHub repository.

```
> git status
> git add .
> git status
> git commit -m "Enable plot"
> git push -u origin main
```

Try Streamlit

6. Deploy Toy Application on Streamlit Community Cloud.
 - i. Sign up <https://streamlit.io/cloud> using GitHub account.
 - ii. Deploy repository:



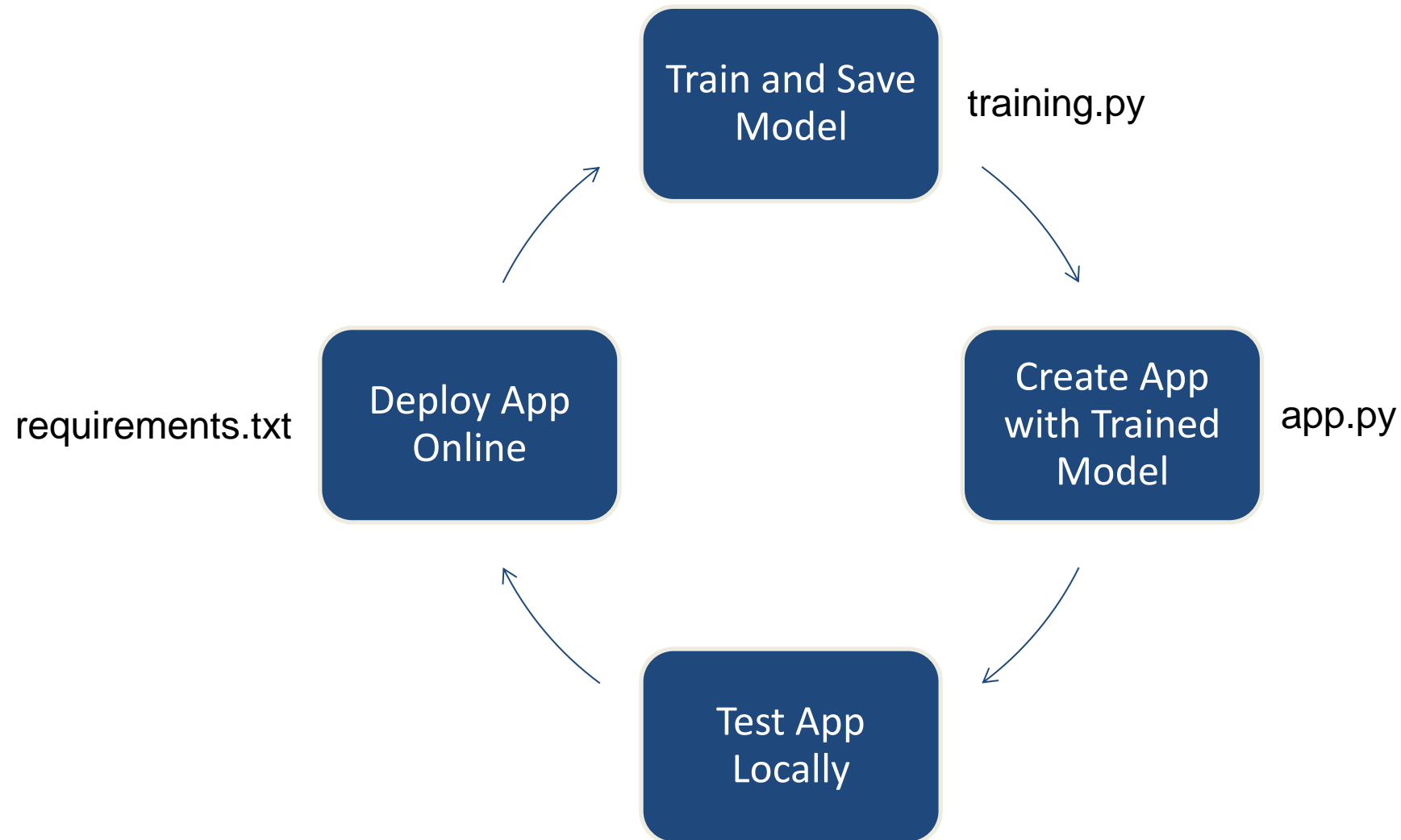
The image shows a sequence of three screenshots from the Streamlit Community Cloud interface, connected by blue arrows indicating the flow of the deployment process.

First Screenshot: The user's profile page on Streamlit. The username is "nus-iss-ds". In the top navigation bar, the "Create app" button is highlighted with a red box. Below the navigation bar, there is a large button that says "Create your first app now". An orange arrow points from this button to the "Create app" button in the navigation bar.

Second Screenshot: The "What would you like to do?" selection screen. It has a "Back" link at the top left. There are two main options: "Deploy a public app from GitHub" (highlighted with a red box) and "Deploy a private app from GitHub". The "Deploy a public app from GitHub" option includes the text "My code is ready on a GitHub repo, and it is totally awesome." and a "Deploy now" button.

Third Screenshot: The "Deploy an app" configuration screen. It contains several input fields: "Repository" (filled with "NUS-ISS-DS/dssi-streamlit"), "Branch" (filled with "main"), "Main file path" (filled with "app.py", highlighted with a red box, and annotated with the red text "Select toy-app.py"), "App URL (optional)" (filled with "dssi-day2"), and "Domain is available" (green text). At the bottom, there is an "Advanced settings" link (highlighted with a red box, with the red text "Change Python version based on your environment (python -V)" next to it) and a "Deploy" button.

Process Flow



Step 1: Train and Save Model

1. Perform EDA and model development on Jupyter notebook.
2. Develop a training script to automate model training and persistence.
3. Run the training module:

```
> python -m src.training --data_path data/loan_dataset.csv --f1_criteria 0.6
```

Sample model training notebook: *DSSI_LoanModel.ipynb*

Sample training script: *training.py*

Step 2: Create App with Trained Model

1. Develop an inference script to load trained model and serve predictions.
2. Build an application with Streamlit that automates decisions with user inputs and trained model.

Sample inference script: *inference.py*

Sample application code: *app.py*

*Retrieve user inputs
and perform
prediction*

```
def app_sidebar():
    st.sidebar.header('Loan Details')
    emp_length_options = ['< 1 year', '1 year', '2 years', '3 years', '4 years', '5 years',
                          '6 years', '7 years', '8 years', '9 years', '10+ years']
    emp_length = st.sidebar.selectbox("Employment Length", emp_length_options)
    int_rate = st.sidebar.slider('Loan Interest Rate', 5, 40, 10, 1)
    annual_inc = st.sidebar.text_input("Annual Income '000s", placeholder="in '000s")
    fico_range_high = st.sidebar.slider('FICO Upper Boundary', 600, 800, 700, 50)
    loan_amnt = st.sidebar.text_input('Loan Amount')

    def get_input_features():
        input_features = {'emp_length': emp_length,
                          'int_rate': int_rate,
                          'annual_inc': int(annual_inc)*1000,
                          'fico_range_high': fico_range_high,
                          'loan_amnt': int(loan_amnt)}

        return input_features

    sdb_coll1, sdb_coll2 = st.sidebar.columns(2)
    with sdb_coll1:
        predict_button = st.sidebar.button("Assess", key="predict")
    with sdb_coll2:
        reset_button = st.sidebar.button("Reset", key="clear")
    if predict_button:
        st.session_state['input_features'] = get_input_features()
    if reset_button:
        st.session_state['input_features'] = {}
    return None
```

Capture user inputs

*Persist user inputs
for the session*

```
assessment = get_prediction(emp_length=st.session_state['input_features']['emp_length'],
                             int_rate=st.session_state['input_features']['int_rate'],
                             annual_inc=st.session_state['input_features']['annual_inc'],
                             fico_range_high=st.session_state['input_features']['fico_range_high'],
                             loan_amnt=st.session_state['input_features']['loan_amnt'])
```

Process Flow

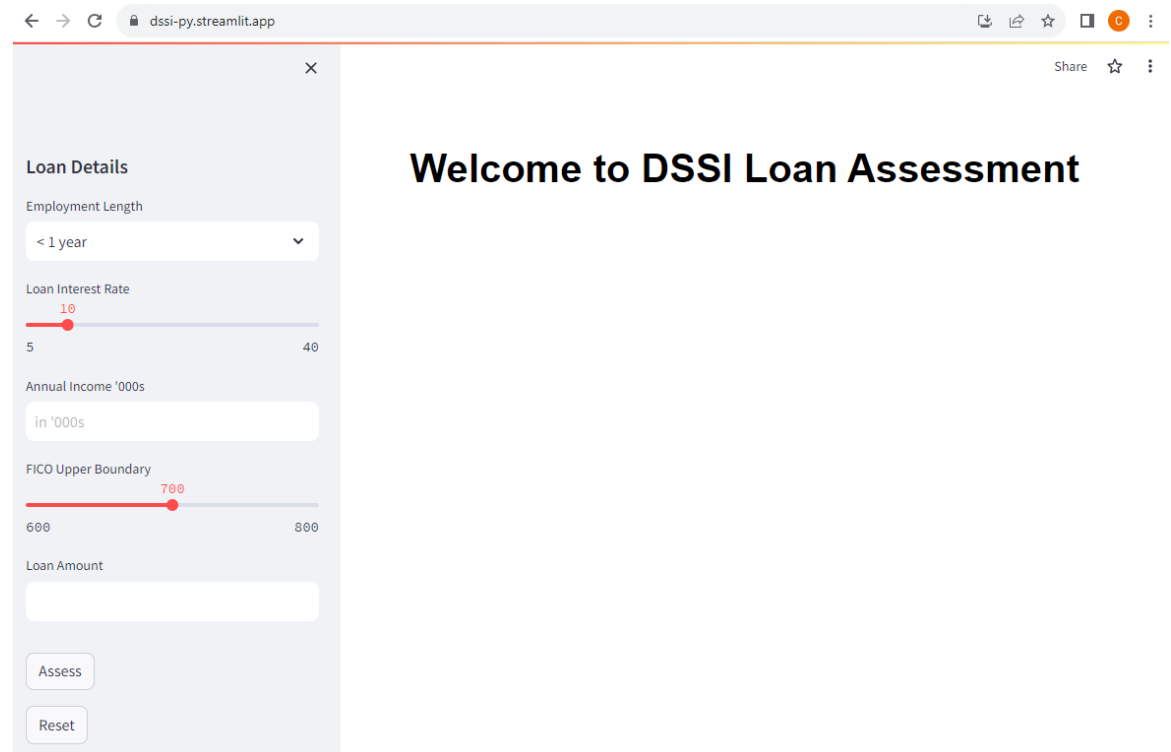
Step 3: Test App Locally

Run Streamlit application:

```
> streamlit run app.py --server.port 8080
```

Step 4: Deploy App Online

1. Commit repository to GitHub.
2. Deploy on Streamlit community cloud.



The screenshot shows a web browser at the URL `dssi-py.streamlit.app`. The application interface is titled "Welcome to DSSI Loan Assessment". On the left, there is a sidebar with the heading "Loan Details" and a close button. The sidebar contains the following inputs: "Employment Length" (a dropdown menu showing "< 1 year"), "Loan Interest Rate" (a slider ranging from 5 to 40 with a red marker at 10), "Annual Income '000s" (a text input field with the placeholder "in '000s"), "FICO Upper Boundary" (a slider ranging from 600 to 800 with a red marker at 700), and "Loan Amount" (a text input field). At the bottom of the sidebar are two buttons: "Assess" and "Reset".

Process Flow

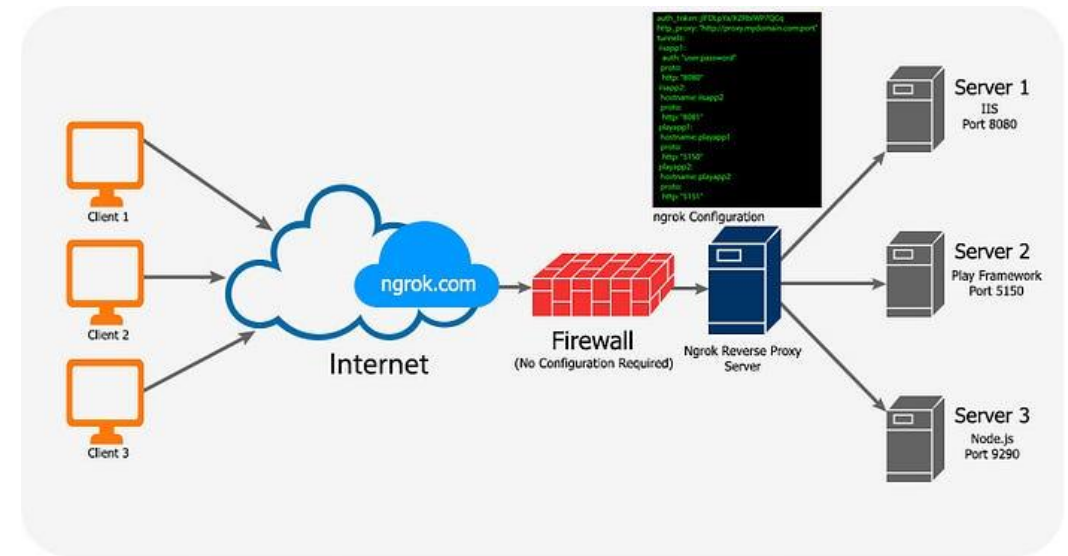
Step 4: Deploy App Online

Expose local service to internet using ngrok.

1. Sign up with <https://ngrok.com/>
2. Complete the instructions to install ngrok on your local machine and add authtoken.
3. Deploy:

```
> streamlit run app.py --server.port 8080
```

```
> ngrok http 8080
```



Note: All servers are in Internal Network running behind http proxy and not in DMZ

```
ngrok (Ctrl+C to quit)
Build better APIs with ngrok. Early access: ngrok.com/early-access
Session Status      online
Account             daveho (Plan: Free)
Version             3.5.0
Region              Asia Pacific (ap)
Latency              3ms
Web Interface        http://127.0.0.1:4040
Forwarding           https://6e4a-137-132-27-105.ngrok-free.app -> http://localhost:8080
Connections          ttl    opn    rt1    rt5    p50    p90
                    9      1      0.02   0.02   1.20   90.01
```

Use static domains and configuration file to run multiple tunnels.

Ref: ngrok. 2023. *ngrok Agent Configuration File*.
<https://ngrok.com/docs/agent/config/>

1. In your group, develop and deploy a machine learning application of your choice with Streamlit Community Cloud.
2. Submit Powerpoint slides that contains:
 - i. Group members name (Cover page)
 - ii. Streamlit application URL (Enable [sharing](#))
 - iii. Screenshot of application



Thank You