Model Deployment

Applied Machine Learning

university of groningen

faculty of science and engineering

So you have a nice model...

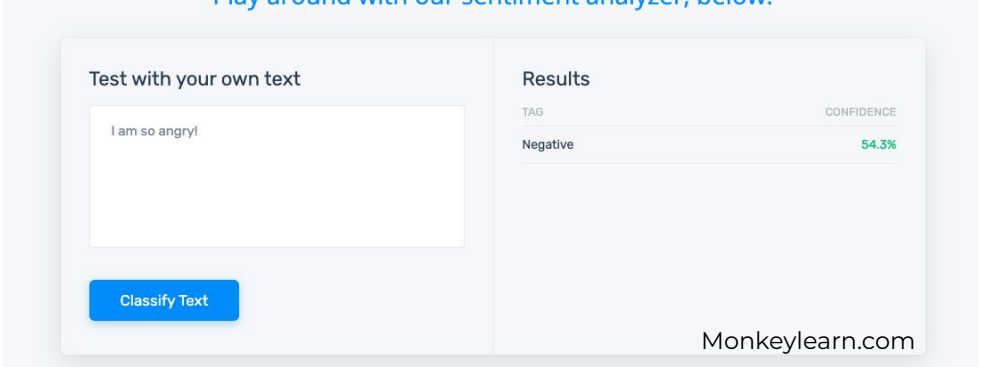
How do you let people use it?

Make a .exe for people to download?

Send people your github repo?

Make a website!

```
from src.data import get_train
from src.model import get_model
import pickle as pkl

x_train, y_train = load_dataset('./data/diabetes_train.csv')
clf = get_model()
clf.fit(x_train, y_train)

with open('./models/production_model.pkl', 'wb') as model_file:
  pkl.dump(clf, model_file)
```



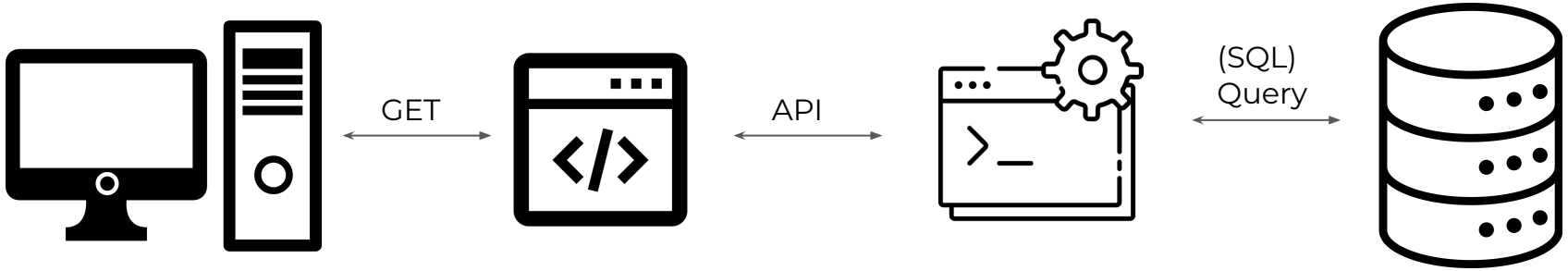
Model deployment

Outline

- Internet & REST
- APIs & FastAPI
- Containerization with Docker
- Streamlit demos

How does the internet work?

A normal website (basics)



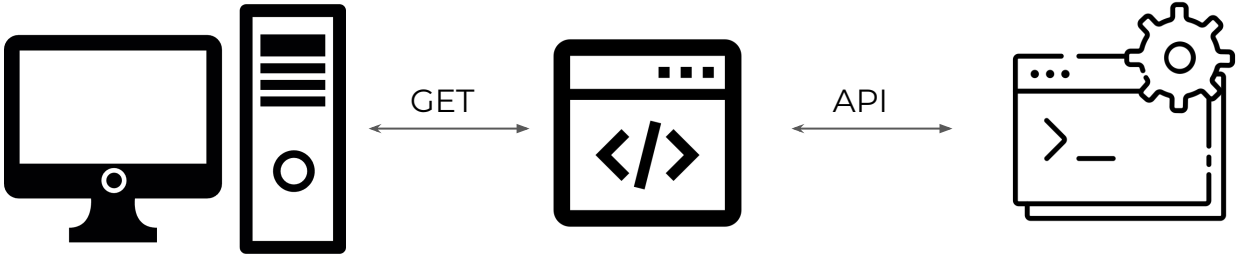
Client

Frontend
HTML & CSS

Backend

Database

Our situation



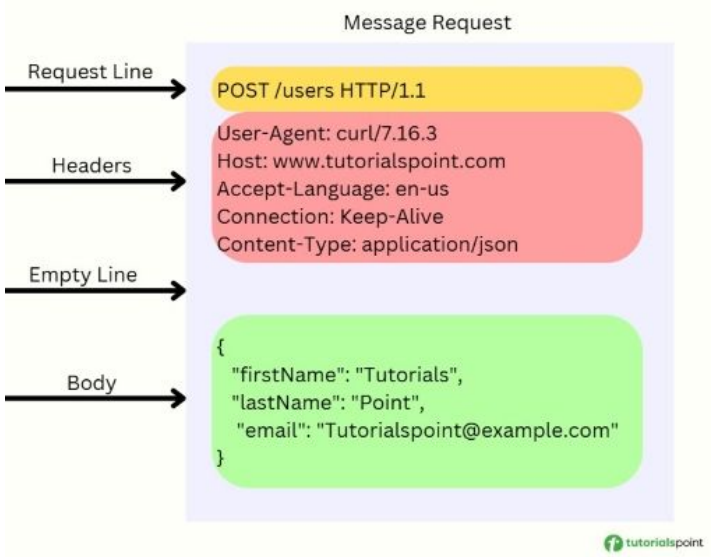
Client

Frontend
Runs client-side

Backend
Runs server-side
ML models!

How do these things communicate?

- **HTTP request**
- **Create Read Update Delete (CRUD)** = POST, GET, PUT, DELETE
- **Target** (with parameters), headers, and a **body** (data to send)



How do these things communicate?

- **HTTP request**
- **Create Read Update Delete (CRUD)** = POST, GET, PUT, DELETE
- **Target** (with parameters), headers, and a **body** (data to send)

```
ivopascal@RN-145-90-167-197 ~ % curl -X GET https://www.rug.nl/
<!DOCTYPE html><html prefix="og: http://ogp.me/ns# article: http://ogp.me/
<meta content="NOODP" name="ROBOTS">
<meta content="nl" http-equiv="Content-Language">
<title>Top 100 university | Rijksuniversiteit Groningen</title>
<link href="https://www.rug.nl/?lang=nl" hreflang="nl" rel="alternate">
<link href="https://www.rug.nl/?lang=en" hreflang="en" rel="alternate">
<link href="https://www.rug.nl/" hreflang="x-default" rel="alternate">
<meta charset="utf-8">
```

- Responds with a body & a **status code** (& headers)
- 200 = **OK**, 404 = **client** error: not found, 500 = internal **server** error

Methods & status code are **agreed standards**. It's your job to follow it.

Sidenote 418: “I’m a teapot”

- Internet Engineering Task Force (IETF) proposed it as a joke
  - **Not officially approved**
- Implemented in Python and others
- IETF asked for it’s removal, but failed

Status codes are a social construct.

Rules for APIs

Your API exists in larger infrastructure, so it should follow certain rules.

They should be RESTful:

- **Stateless**
  - API should not keep session information
  - Restarting API or adding a server has no effect
- **Cacheable**
  - Same request = same response
  - Usually possible for ML models!
- **Uniform interface**
  - HTML, XML or JSON. Not internal server representation

ML model as a RESTful API

Internet rules don’t really work for us. Here’s my advice:

- POST request with JSON body
- Respond with 200 and output in JSON
- Use 400 for bad requests
- Use 500 for bugs

```
curl --request POST
    --header "Content-Type: application/json"
    --data {"age": 42, "bmi": 28.4, "glu": 95}
    http://0.0.0.0:80/classify_diabetes
-----------------------------------------------
Status code: 200
Body:
{
    "diabetes": false
}
```

Illustrative response

Implementing with FastAPI

Why FastAPI?

- Quick way to turn your model into API
- Built-in data validation & error reporting
- **Automatic documentation**
- Slightly disconnected from how clients send requests ☹️
  - Hides what's actually happening

First things on FastAPI

- @app. get, post, put and a path
  - “decorators”
- Return the API response
- Asynchronous
  - Clients call a function
- Run with uvicorn
  - -- reload to automatically update API

```python
from fastapi import FastAPI

app = FastAPI()


@app.get("/") # Get endpoint on root
async def root():
    return {"message": "Hello World"} # json response



$ uvicorn main:app --reload # special way to start
```

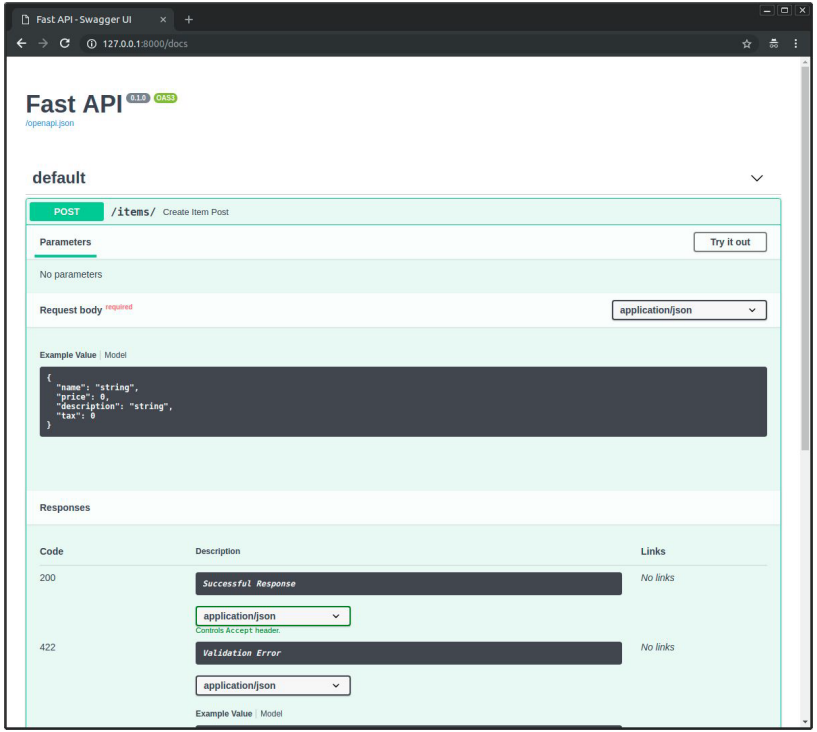
Request bodies with Pydantic

- Pydantic defines data structure
- FastAPI will reject bad requests
- FastAPI will make documentation
  - Pydantic default value
  - Pydantic example
  - Pydantic response model

```python
from fastapi import FastAPI
from pydantic import BaseModel
from src.model import load_model, make_prediction

class ModelInput(BaseModel): # FastAPI reads this json
    age: int
    bmi: float
    glu: int


app = FastAPI()
model = load_model()


@app.post("/classify_diabetes/")
async def predict_diabetes(features: ModelInput):
    prediction = make_prediction(model, features)
    prediction_bool = prediction[0] == 1
    return {"diabetes": prediction_bool}
```

FastAPI documentation with Swagger

- `host:port/docs` for documentation
- Reads details from FastAPI init
  - Title, description, version, summary
- Reads Pydantic request and response
- Checks possible error codes
  - raise `HTTPException` to tell client
- Has “try it out” for easy demos



Spot all the problems!

- Retrain each request
- Bad documentation
- Bad variable names
- Wrong error code
- Bad endpoint name

- Not REST:
- Not uniform
- Not cacheable
- Not stateless

```python
# find the problems

app = FastAPI()

previous_request_body = None

@app.post('/api_post/')
def inf(data: Dict[str, Any]):
    if data == previous_request_body:
        raise HTTPException(status_code=404, details="Don't ask me twice")
    previous_request_body = data
    m = train_model('./dataset.csv')
    p = m.predict(data['bmi'], data['age'], data['glu'])
    return str(p)
```

This works, but poorly!

Take away from FastAPI

- @app.post for asynchronous functions
- Use Pydantic to define inputs and outputs
- Documentation with Swagger

Let’s look at it in practice!

Sidenote: Missing step

- Localhost, 0.0.0.0, 127.0.0.1 = **local network**

- Expose port to public internet = **port forwarding**
  - **Accept requests from the internet**
- Setting in router

Virtual Machines & Containers

“But it worked on my computer!” ~ every developer

- Then we’ll ship your computer!

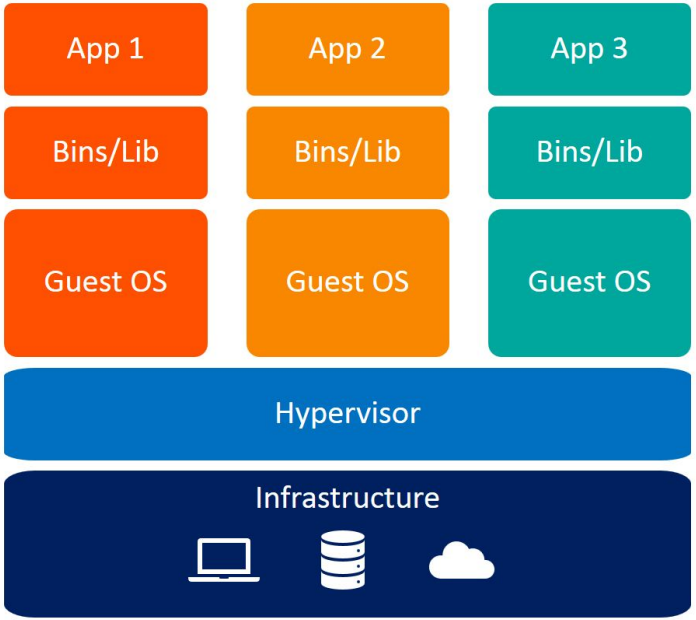
- Virtual Machine: whole Operating System inside another computer
  - Designed for time-sharing (1960s)
  - Used for running e.g. Linux in Windows

1. Develop product in VM
2. Send VM to server to run
3. Need to handle more requests? Add a server with same VM
4. Guaranteed success!

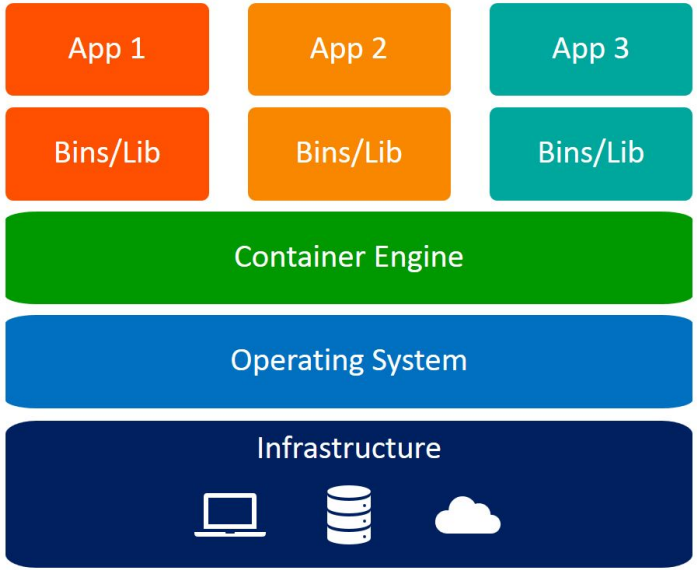
But VMs are ***big*** and **slow**

Containerization

VM without OS
Lightweight
Faster startup



Virtual Machines



Containers

Benefits of containerization

- Reproducible code
- Multiple copies of same API
- Crash? Restart a new one!
- Standard “package” to deliver & update
  - Containerize ML model API
  - Devops will take care of the rest

Containerization with Docker

How to make a Dockerfile:

Start with a base image

Install all you need

Copy files

Start your code

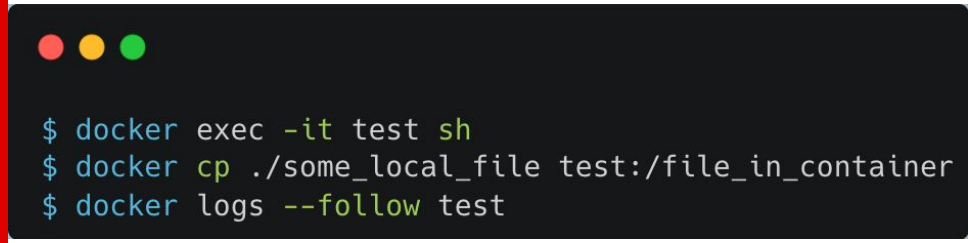
```
# Based on Ubuntu container
FROM ubuntu


# Install python
RUN sudo apt-get update
RUN apt-get -y install \
    build-essential \
    python-dev \
    python-setuptools \
    python-pip

# Install SKLearn
RUN apt-get -y install \
    python-numpy \
    python-scipy \
    libatlas-dev \
    libatlas3-base \
    python-sklearn

WORKDIR /code
COPY ./requirements.txt /code/requirements.txt
#Install fastapi
RUN pip install -r /code/requirements.txt


# Add code folder
COPY ./src /code/src

# Start server
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "80"]
```

1. Build image
2. Run container
    a. With open port!
3. Watch container run
4. Stop container

```
$ docker build -t mnist-classifier .
$ docker run -d -p 80:80 mnist-classifier --name test
$ docker ps
CONTAINER_ID    IMAGE    ...
test            Ubuntu   ...
$ docker stop test
$ docker rm test
$ docker image rm mnist-classifier
```

You can do operations to a running Docker container.

This is helpful for debugging, but shouldn't be used in deployment.



Beyond your container: Docker Compose

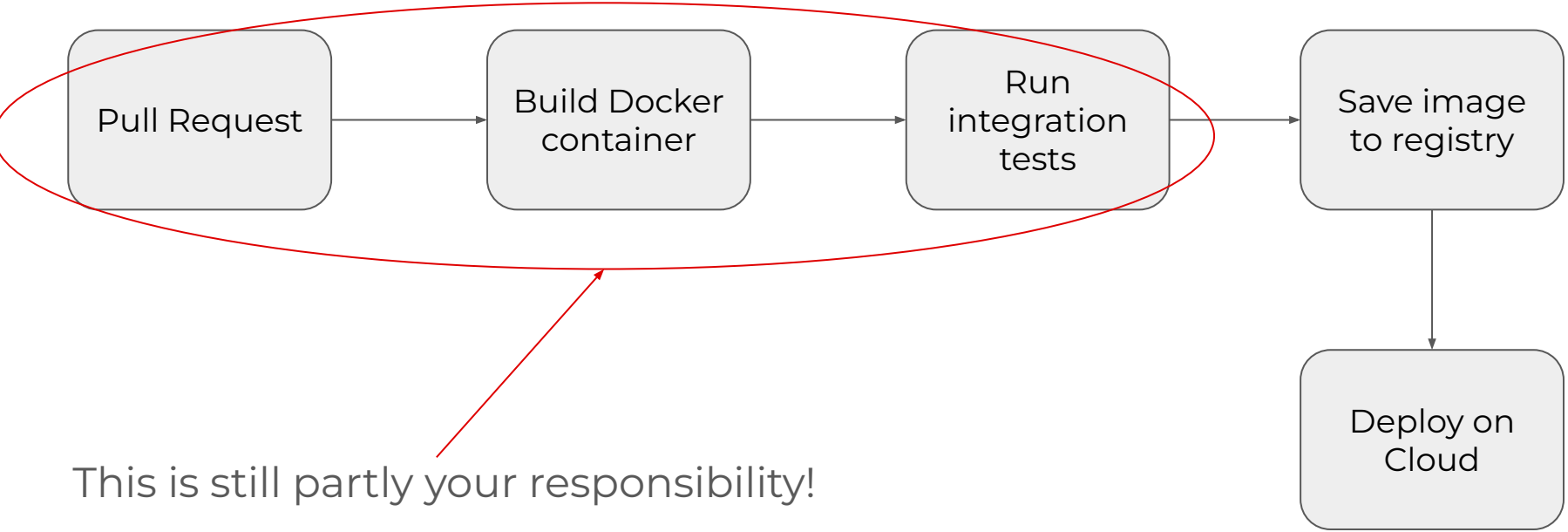
- Different containers working together (frontend, backend, database)
- Automatically restart containers that crash
- Duplicates of same container

- Define everything in a `compose.yaml`

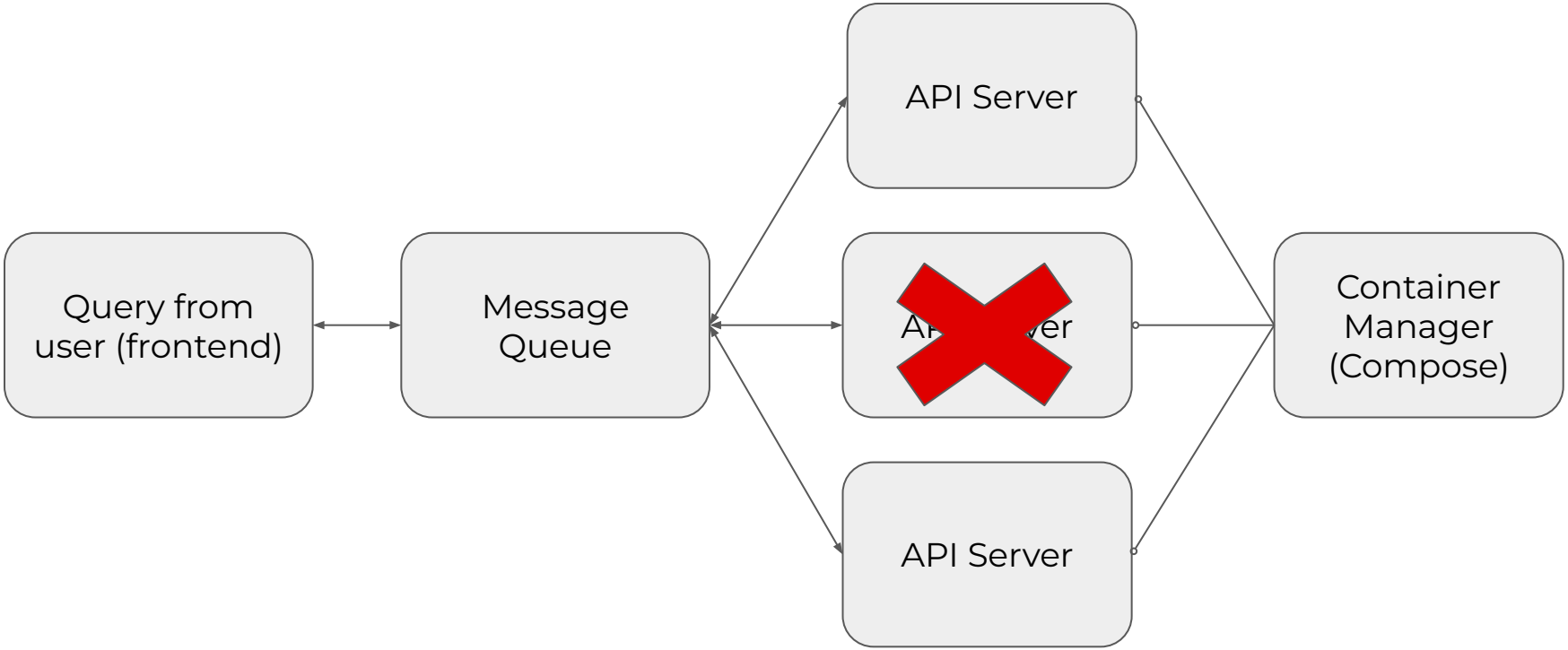
- Not ready for production-grade

Containers in Production

What your friends in DevOps are doing for CI/CD



What your friends in DevOps are doing (production)

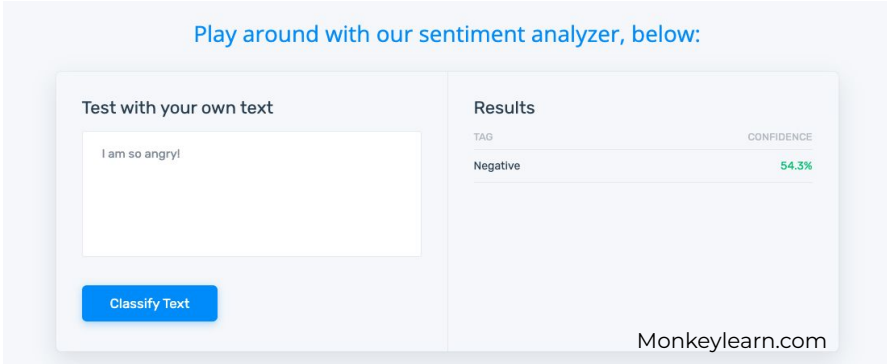


Takehome from Docker

- Ensure your code is reproducible
- Copy of your Machine, without OS
- Allows for scaling up & down
- Essential for reliable professional infrastructure

Demos with Streamlit

What I promised



Frontend
HTML & CSS = difficult work

What I delivered

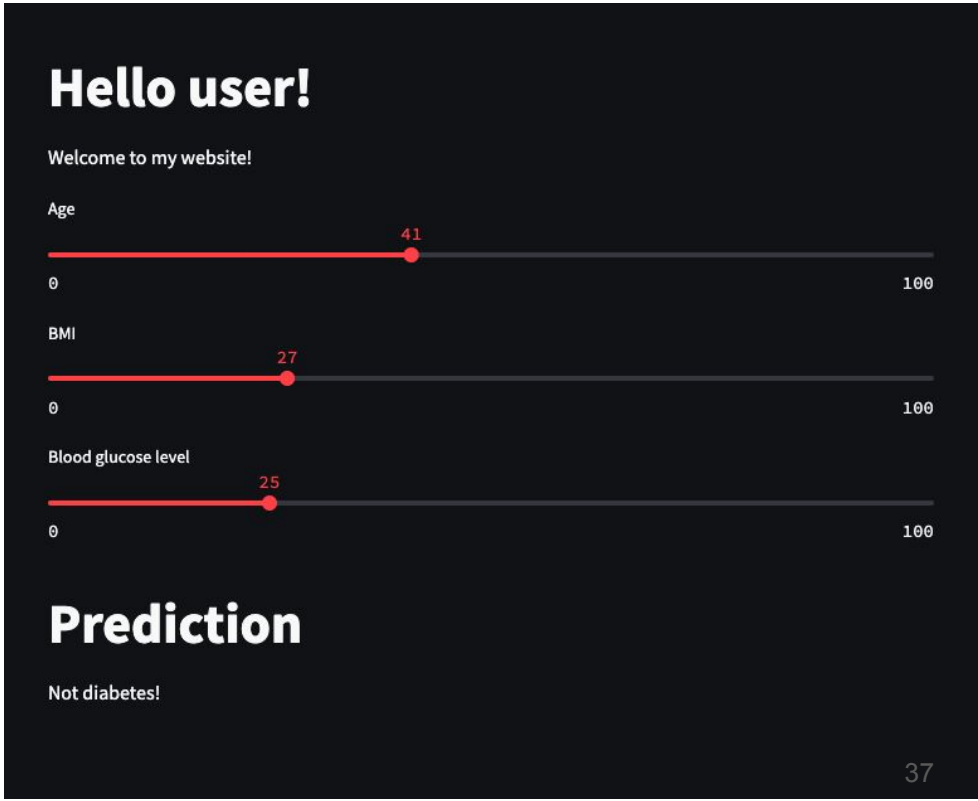
```
curl --request POST
    --header "Content-Type: application/json"
    --data {"age": 42, "bmi": 28.4, "glu": 95}
    http://0.0.0.0:80/classify_diabetes
-----------------------------------------------
Status code: 200
Body:
{
    "diabetes": false
}
```

Backend

Simple streamlit demo

```python
import streamlit as st
from src.model import get_model, make_prediction

if __name__ == "__main__":
    welcome = """
    # Hello user!
    Welcome to my website!
    """

    st.write(welcome) # Write arbitrary text to website

    # Set slides for values
    age = st.slider("Age")
    bmi = st.slider("BMI")
    glu = st.slider("Blood glucose level")

    model = get_model("models/trained_model.pkl")

    st.write("# Prediction")
    if make_prediction(age, bmi, glu, model): # Make prediction
        st.write("Diabetes!") # Write result based on prediction
    else:
        st.write("Not diabetes!")
```

```
$ python -m streamlit run main.py
```



Streamlit possibilities

- Many features for **visualisation & input**
- **Limited options** for customizability
- Does not **scale**

- Great for **quick** (online) **demos**, but **not for production**

What is expected of your model deployment?

- Required:
  - FastAPI running locally
  - Good API documentation
  - Model better than random guessing
- Would be nice:
  - Streamlit demo
- Above & beyond:
  - Docker container

Relation to jobs

Jr Machine Learning Ops Engineer

Albert Heijn | 1506 Zaandam

Who you are:

- You have an MSc in Computer Science, Artificial Intelligence, Data Science, or a related discipline
- ✓ You enjoy writing pythonic, clear, extendable, testable code, and can apply Software Engineering best practices
- ✓ You can build a data pipeline using SQL, Spark, Pandas, or other relevant tools for transformation of relational data
- ✓ You are familiar with the concepts of version control and continuous integration and delivery
- You know the fundamentals of Machine Learning and Statistics
- Knowledge of the following topics would be a plus:
- ✓ Containerization
- ⊘ Cloud platforms
- ✓ REST APIs

Github Actions, Precommit

Relation to jobs

Graduate Data Scientist (2024 Start)

Optiver | Amsterdam

In terms of skills and qualifications, we're looking for:

- Excellent data analysis, scientific and critical thinking skills, as well as an eye for detail
- A decent level of Python development ability, experience with scientific Python stack (e.g. Pandas, Numpy and SciPy) is highly desirable
- Outstanding communication skills to collaborate with multiple stakeholder groups, challenge thinking and obtain buy-in
- The ability to present complex ideas with data to expedite decision making
- A focus on pragmatic outcomes, and the ability to prioritise effectively
- A strong interest in the world of trading, however no experience in the financial industry required

What’s next?

- Model deployment due next week Thursday
- Should be with a model > random guessing

- Next week lecture on Model Comparison & hyperparameter optimization
- After that lecture on Responsible AI
    - Explainability, uncertainty, robustness,