FSDL 2022

Deployment

Josh Tobin

September 5, 2022

"All-in-one"

Amazon SageMaker

gradient by Paperspace

DOMINO DATA LAB

Data

Versioning
Pachyderm, DVC, LIQUIDATA

Labeling
Label Studio, scale, prodigy

Processing
dbt, DAGSTER

Exploration
pandas, dask, RAPIDS

Datasets

Sources
S3, databricks, Parquet, snowflake

Development

Frameworks & Distributed Training
PYTORCH, fast.ai, DeepSpeed, HOROVOD, FairScale

Experiment and Model Management
comet, Weights & Biases, TensorBoard, mlflow, neptune.ai

Resource Management
Determined AI, PY, slurm workload manager

Software Engineering
jupyter, Streamlit, git

Compute
CoreWeave, Lambda

Deployment

Feature Store
tecton

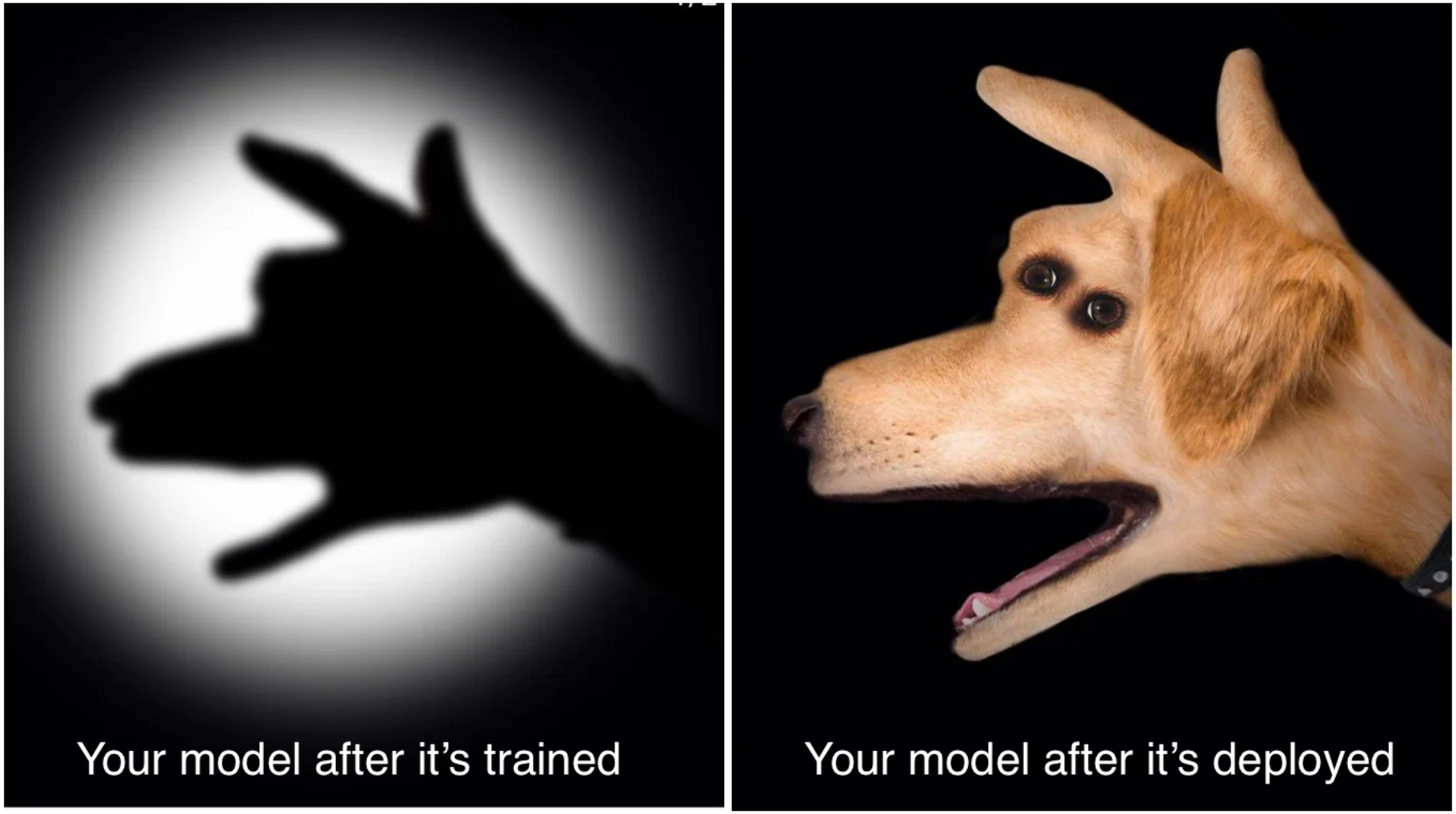
Monitoring
fiddler

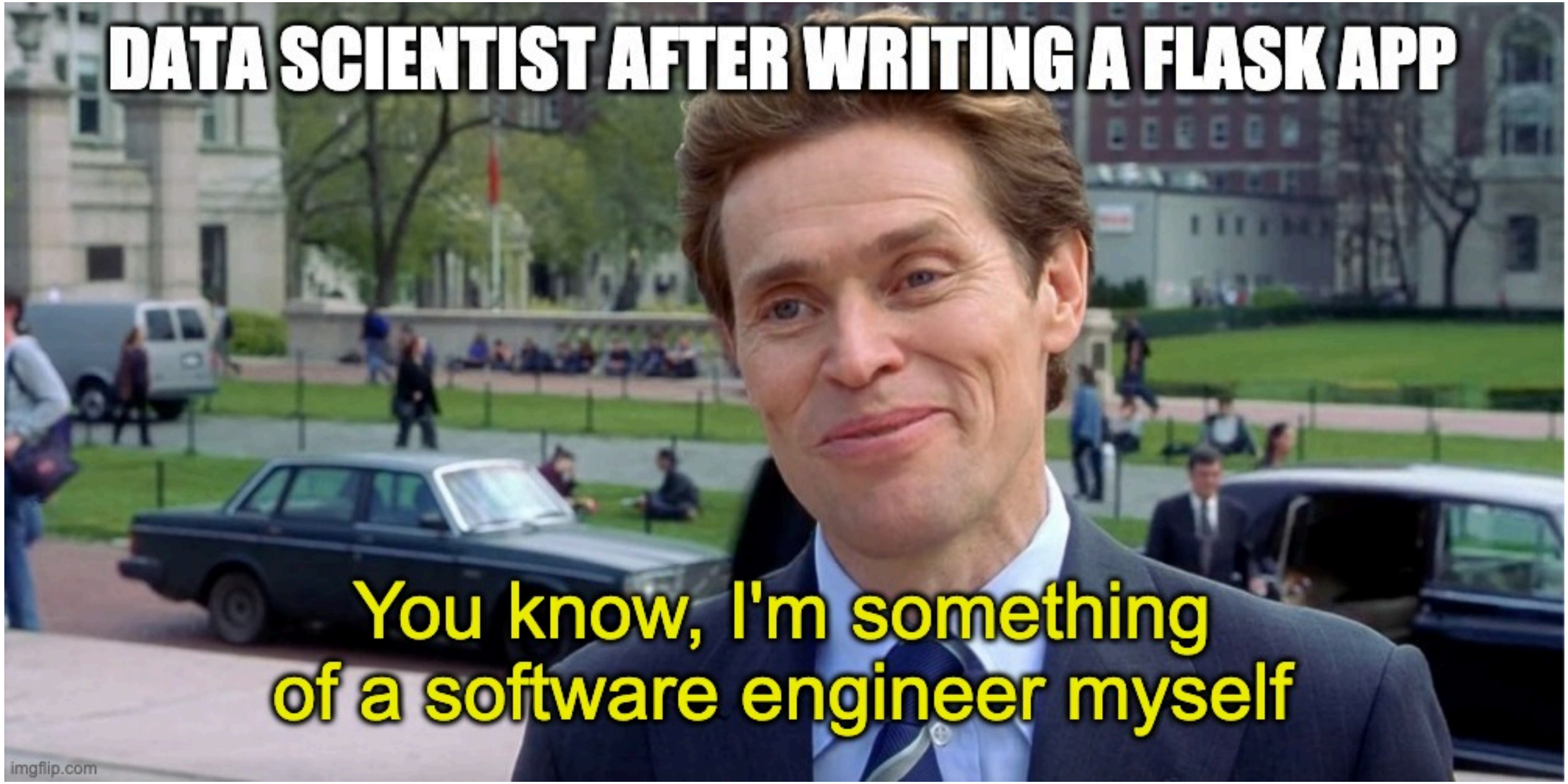
Edge
ML, OctoML, NVIDIA TensorRT, TensorFlowLite, ONNX

Web
ALGORITHMIA, SELDON

CI / Testing
Buildkite, great_expectations

or or


Your model after it’s trained
Your model after it’s deployed

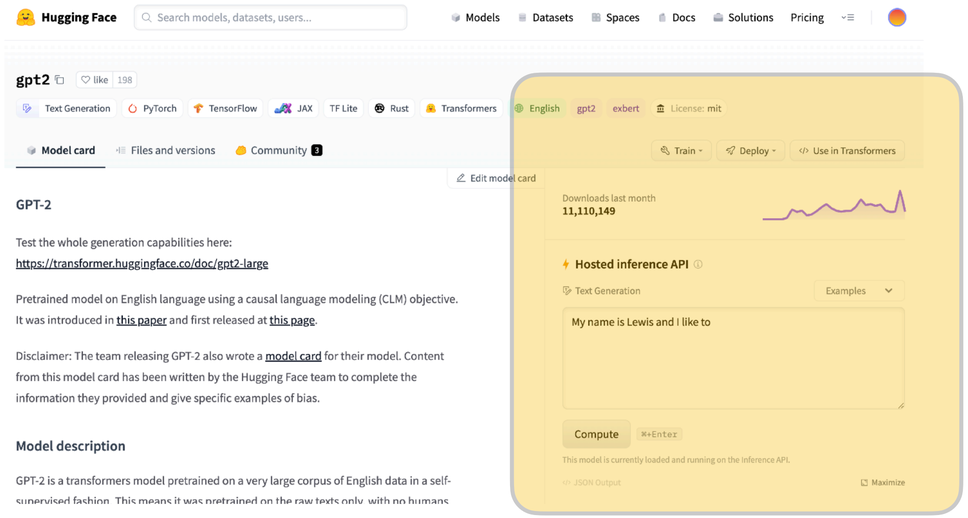

DATA SCIENTIST AFTER WRITING A FLASK APP
You know, I'm something
of a software engineer myself
imgflip.com

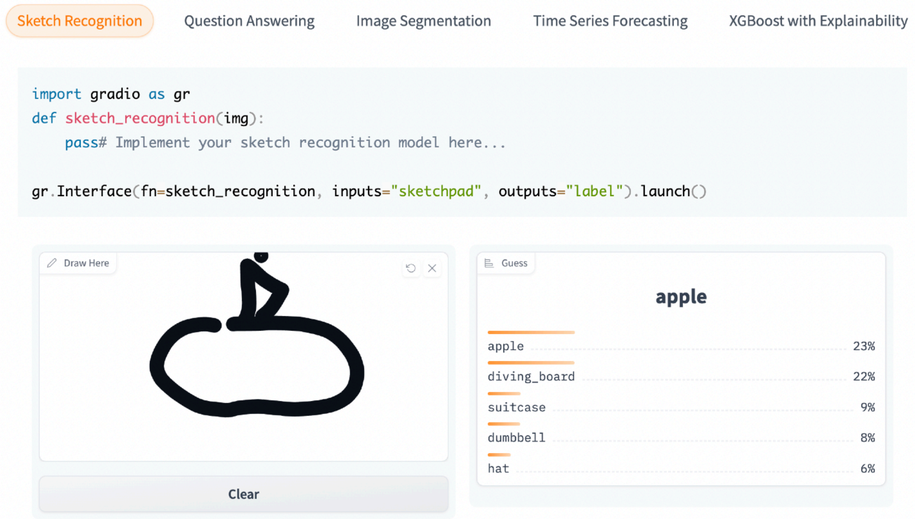
Only IRL do you see how your model **actually** works

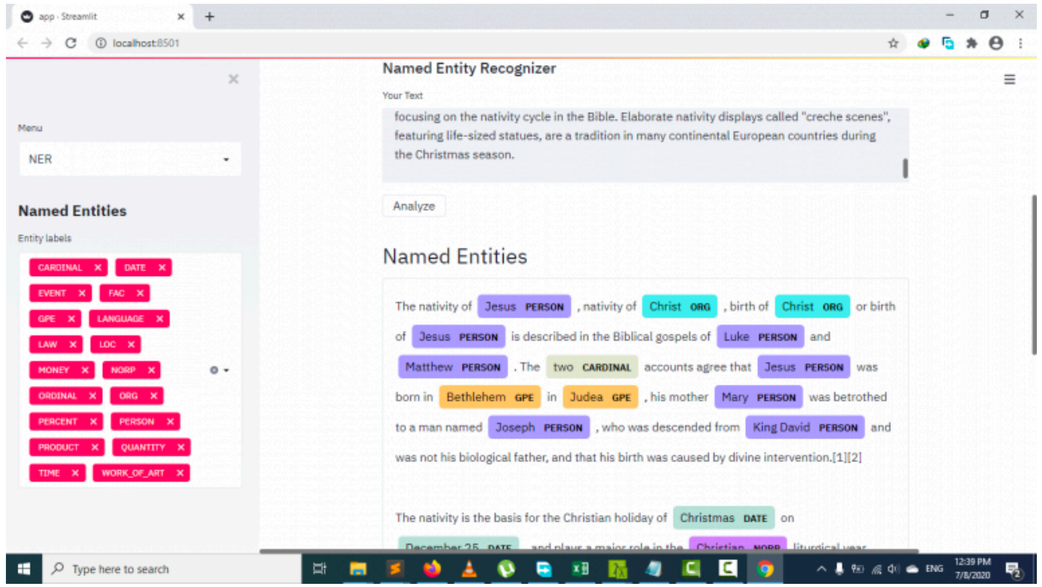
- Deploy early, deploy often
- Keep it simple, and add complexity later
  - Build a prototype
  - Separate your model and UI
  - Learn the tricks to scale
  - Consider moving your model to the edge when you *really* need to go fast ⚡

Step 1: build a prototype you and your friends / teammates can interact with

Tools for prototype deployment







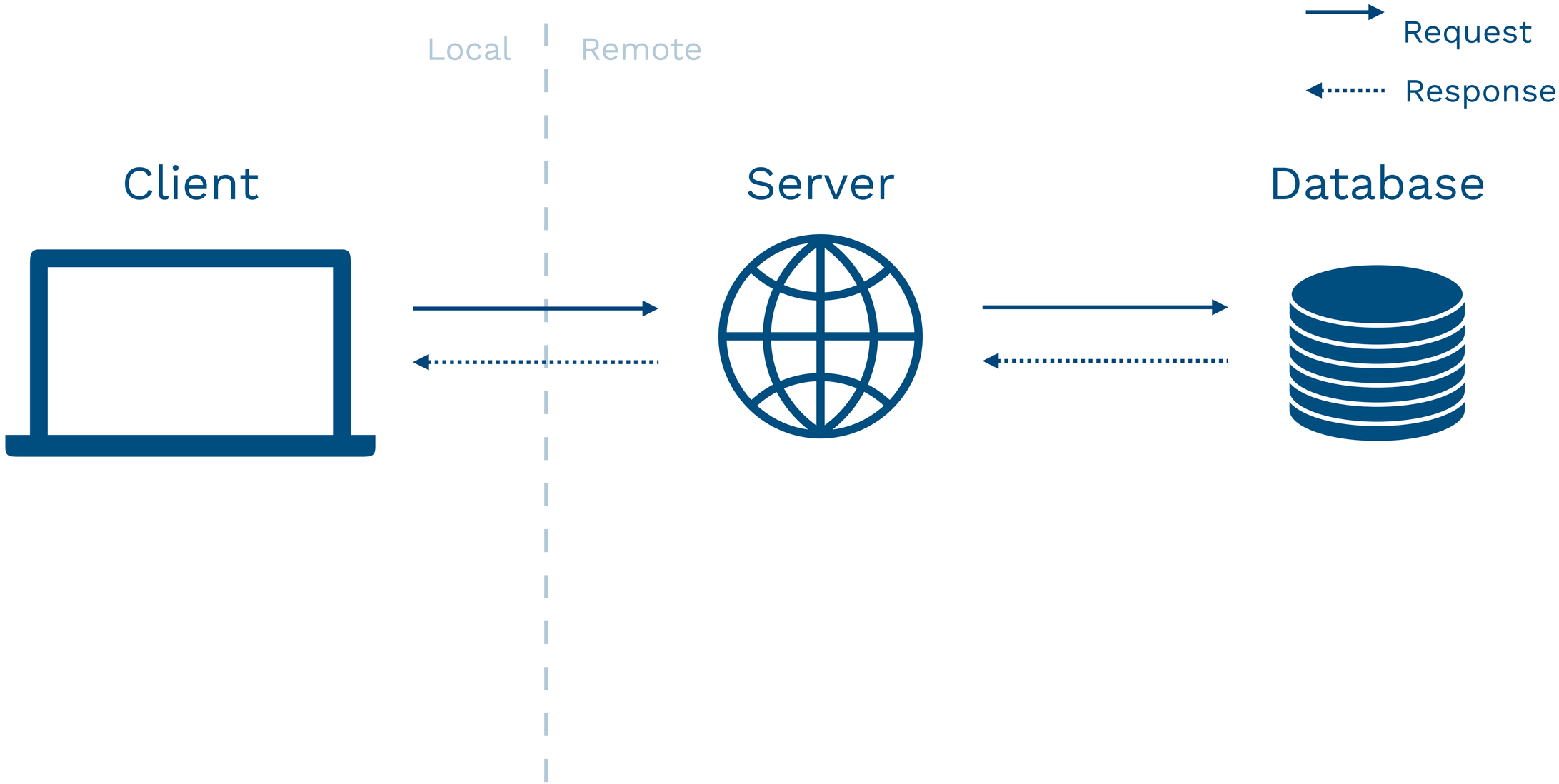
Prototype deployment: best practices

- Have a basic UI
  - Easier for other folks to try it and give feedback
  - Gradio & Streamlit are your friends here
- Put it behind a web URL
  - Easier to share
  - Cloud versions of streamlit and huggingface are helpful here
- Don’t stress too much

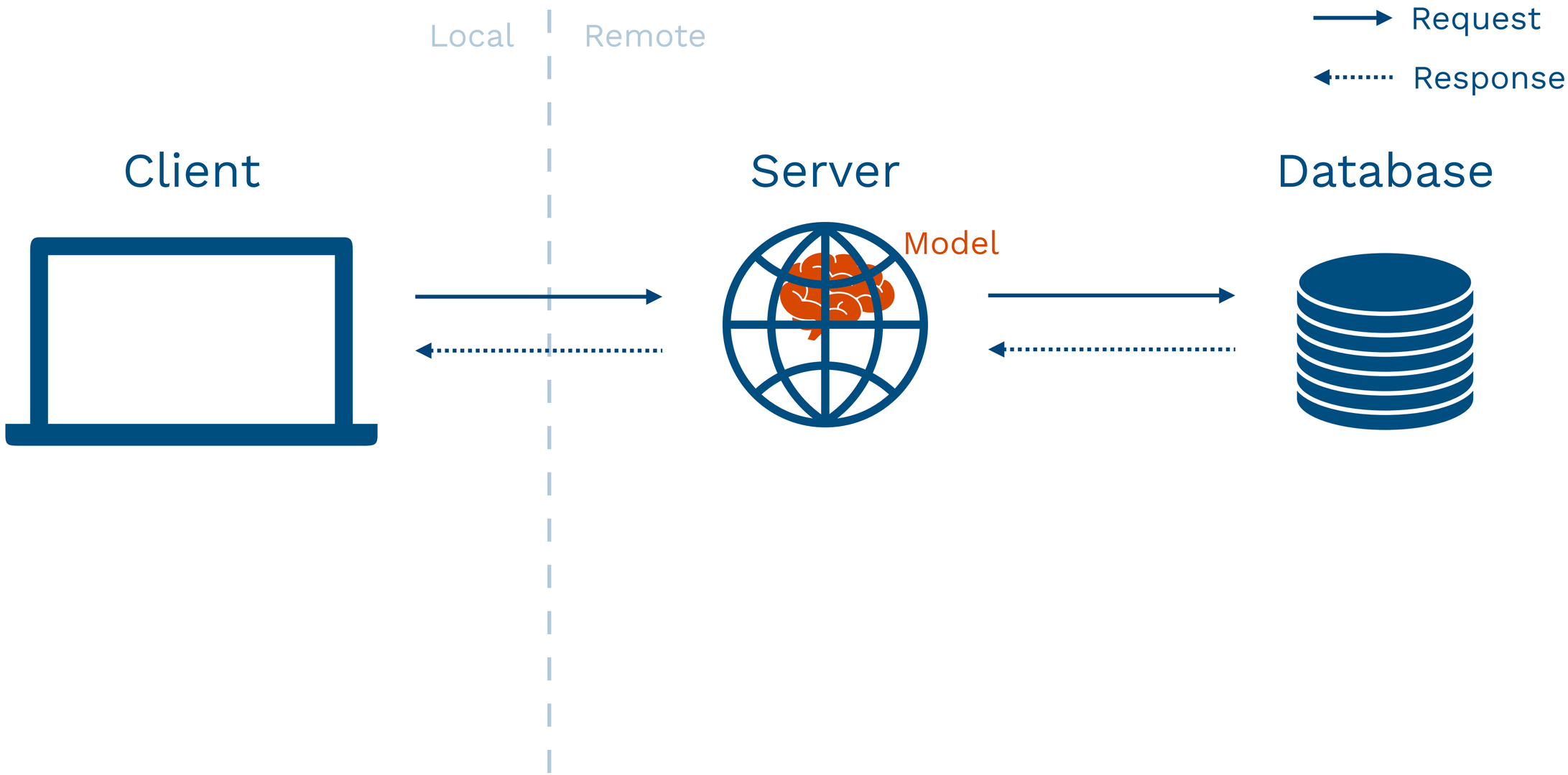
Where will this fail?

- Limited frontend flexibility
- They don’t scale to many concurrent requests: the model becomes the bottleneck

Where in the architecture should your model go?



Model-in-service

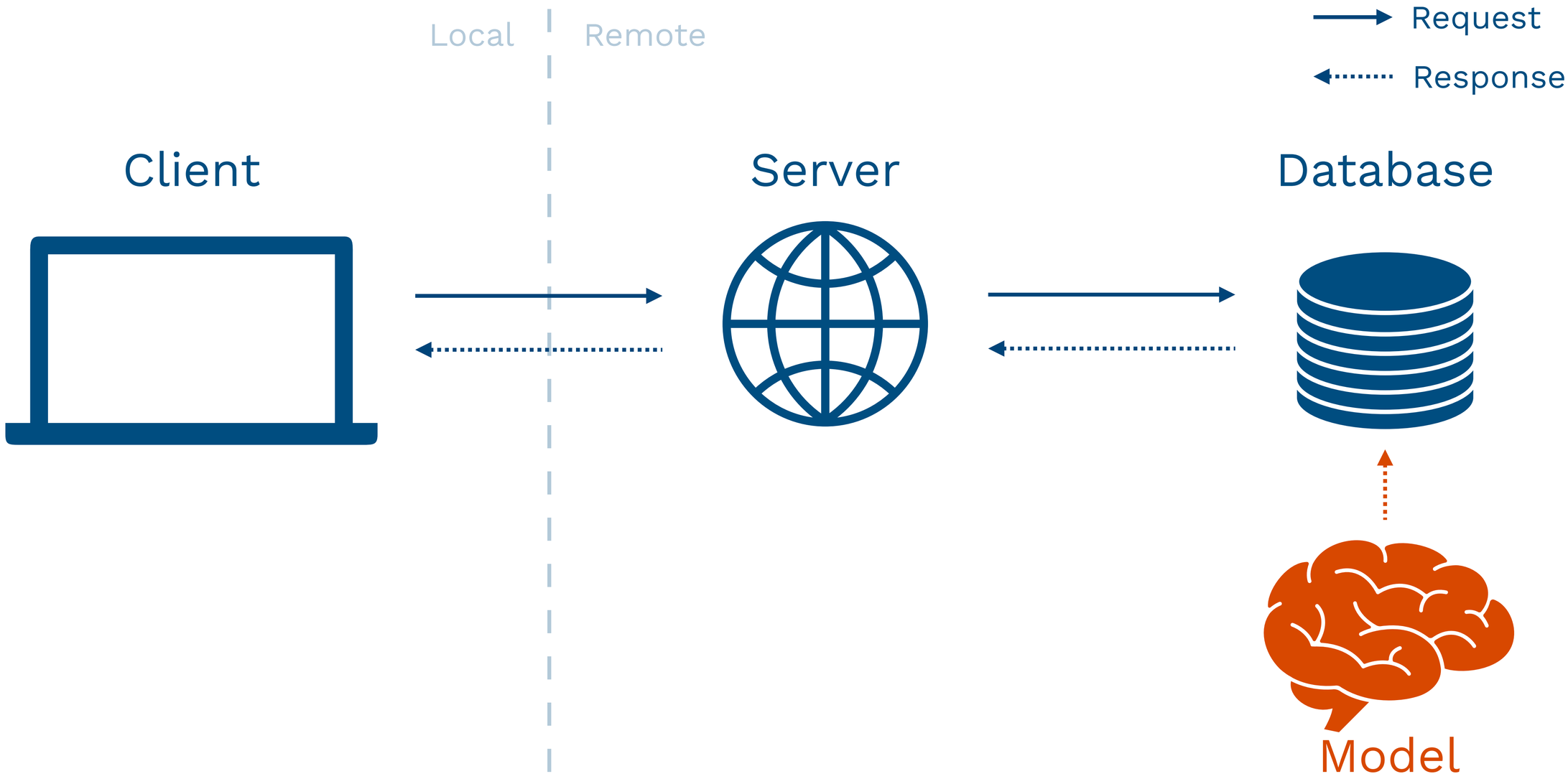


Model-in-service

Pros	Cons
• Re-uses your existing infrastructure	• Web server may be written in a different language • Models may change more frequently than server code • Large models can eat into the resources for your web server • Server hardware not optimized for your model (e.g., no GPUs) • Model & server may scale differently

Step 2: separate your model from your UI

Option 1: Batch prediction



Batch prediction

- Periodically run your model on new data and save the results in a database
- Works if the universe of inputs is relatively small (e.g., 1 prediction per user, per client, etc)
  - Recommender systems
  - Marketing automation (e.g., lead segmentation)

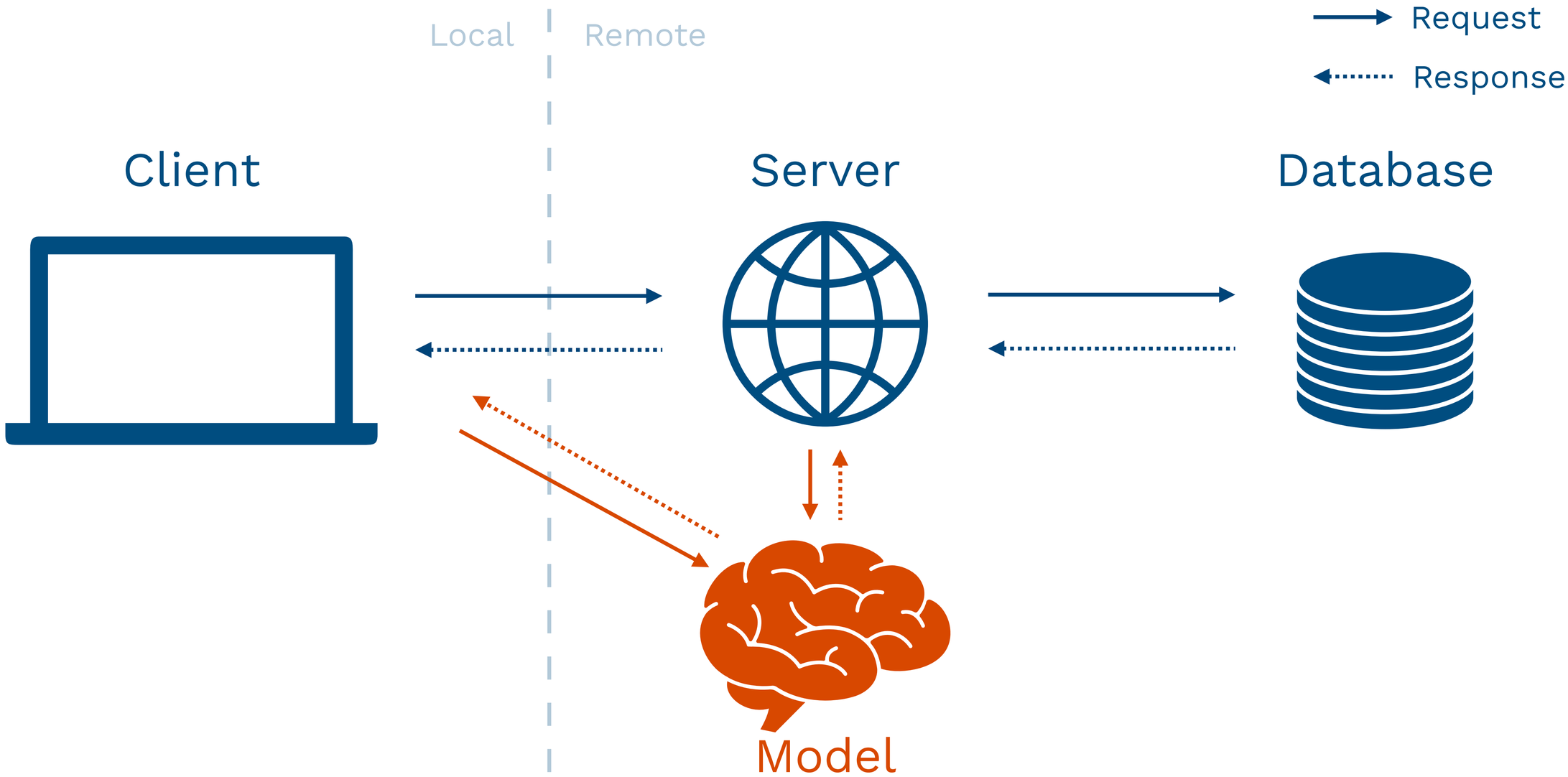
Data processing / workflow tools work well here

- Re-run preprocessing
- Load the model
- Run predictions
- Store predictions

Batch prediction

Pros	Cons
• Simple to implement • Scales easily • Used in production by large-scale production systems for years • Fast to retrieve the prediction	• Doesn’t scale to complex input types (user-specified queries, etc) • Users don’t get the most up-to-date predictions • Models frequently become “stale”, which can be hard to detect

Model-as-service



Model-as-service

- Run your model on its own web server
- The backend (or the client itself) interact with the model by making requests to the model service and receiving responses back

Model-as-service

Pros

- Dependability — model bugs less likely to crash the web app
- Scalability — choose optimal hardware for the model and scale it appropriately
- Flexibility — easily reuse a model across multiple apps

Cons

- Can add latency
- Adds infrastructural complexity
- Now you have to run a model service…

Building a model service: the basics

- REST APIs
- Dependency management
- Performance optimization
- Horizontal scaling
- Rollout
- Managed options

Building a model service: the basics

- **REST APIs**
- Dependency management
- Performance optimization
- Horizontal scaling
- Deployment
- Managed options

REST APIs

- Serving predictions in response to canonically-formatted HTTP requests
- There are alternatives like GRPC (which is actually used in tensorflow serving) and GraphQL (not terribly relevant to model services)

REST API example

```
josh@Joshs-MacBook-Pro ~ % curl -X POST --url https://api.fullstackdeeplearnin
g.com/predict -d '{"name": "josh", "year":2021, "subject": "ml", "pages": 20}'
```

Formats for requests and responses

- Sadly, no standard yet

Google Cloud

gcloud | REST API

Make each instance an item in a JSON array, and provide the array as the `instances` field of a JSON object. For example:

```
{"instances": [
  {"values": [1, 2, 3, 4], "key": 1},
  {"values": [5, 6, 7, 8], "key": 2}
]}
```

Azure

The REST API expects the body of the request to be a JSON document with the following structure:

JSON | Copy

```
{
    "data":
        [
            <model-specific-data-structure>
        ]
}
```

AWS Sagemaker

```
{
  "instances": [
    {"in0": [863], "in1": [882]},
    {"in0": [278], "in1": [311]},
    {"in0": [705], "in1": [578]},
    …
  ]
}
```

Building a model service: the basics

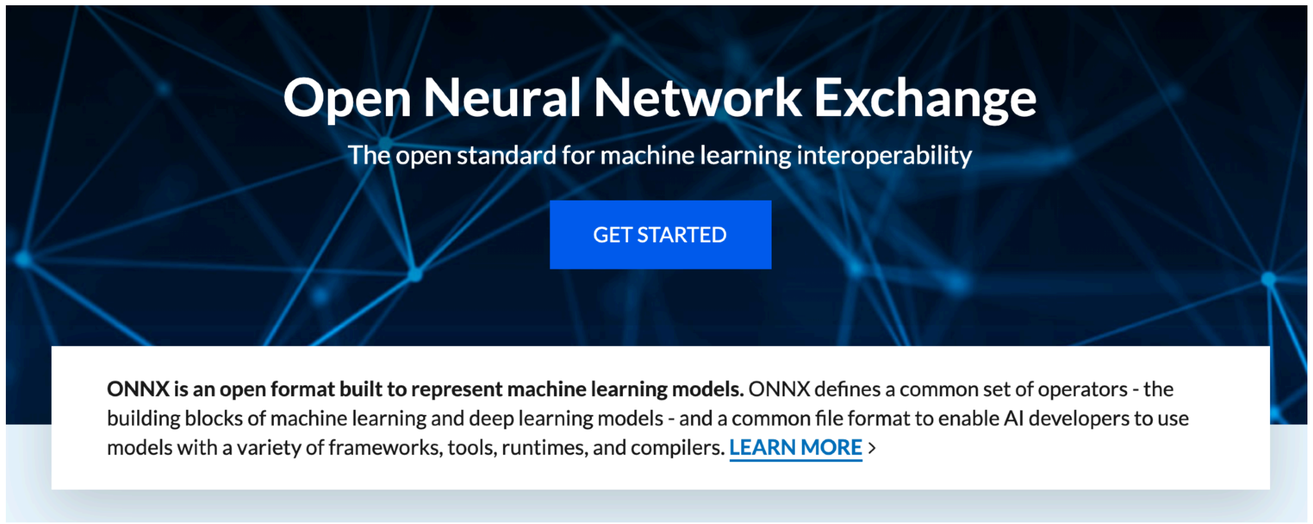
- REST APIs
- **Dependency management**
- Performance optimization
- Horizontal scaling
- Deployment
- Managed options

Dependency management for model servers

- Model predictions depend on **code, model weights,** and **dependencies**. All need to be present on your web server
- **Dependencies cause trouble**.
  - Hard to make consistent
  - Hard to update
  - Even changing a tensorflow version can change your model
- Two strategies:
  - Constrain the dependencies for your model
  - Use containers

Constraining model dependencies

A standard neural net format: ONNX



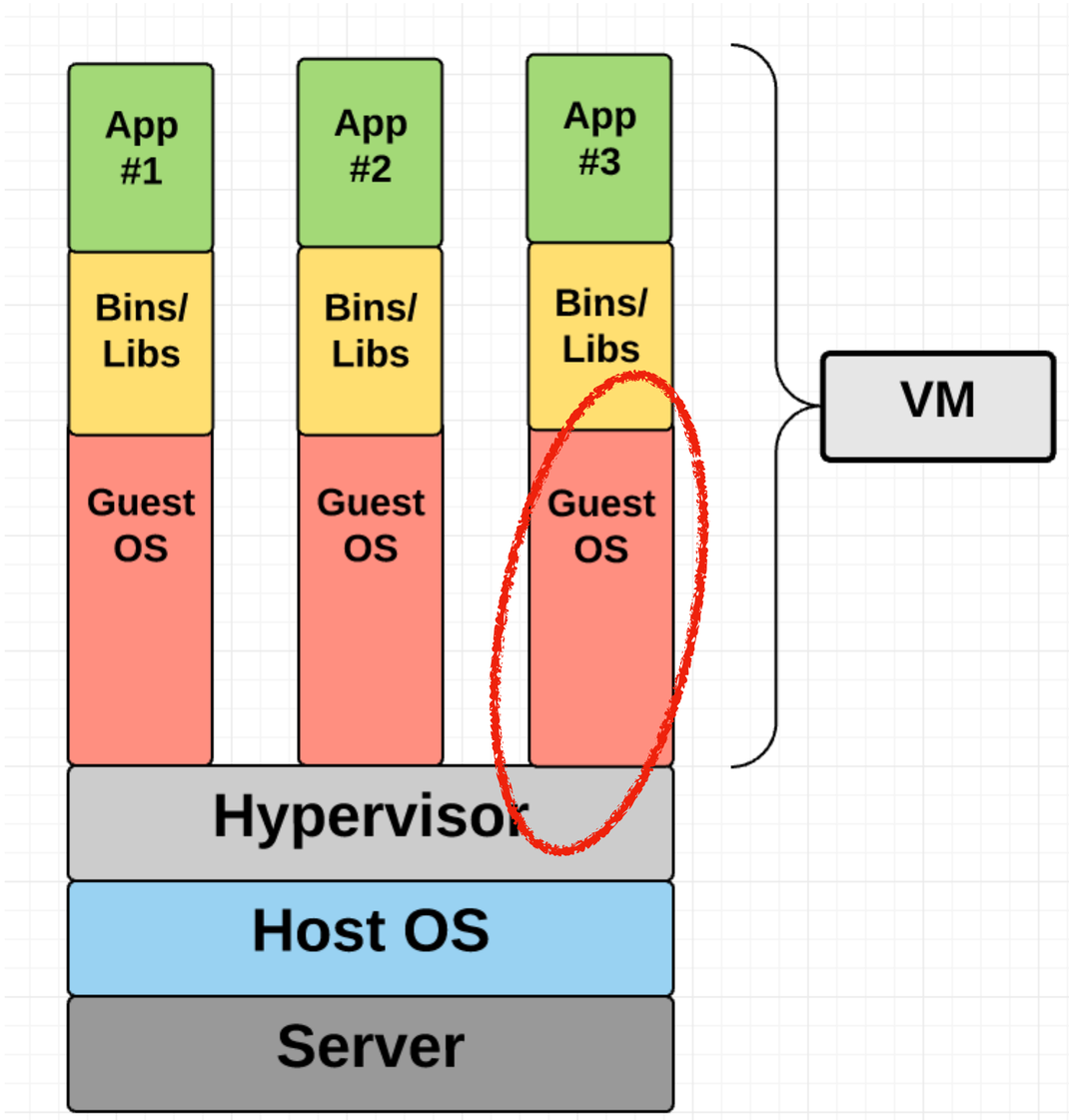
- The promise: define network in any language, run it consistently anywhere
- The reality: since the libraries change quickly, there are often bugs in the translation layer
- What about non-library code like feature transformations?

https://github.com/sayakpaul/ml-deployment-k8s-fastapi/

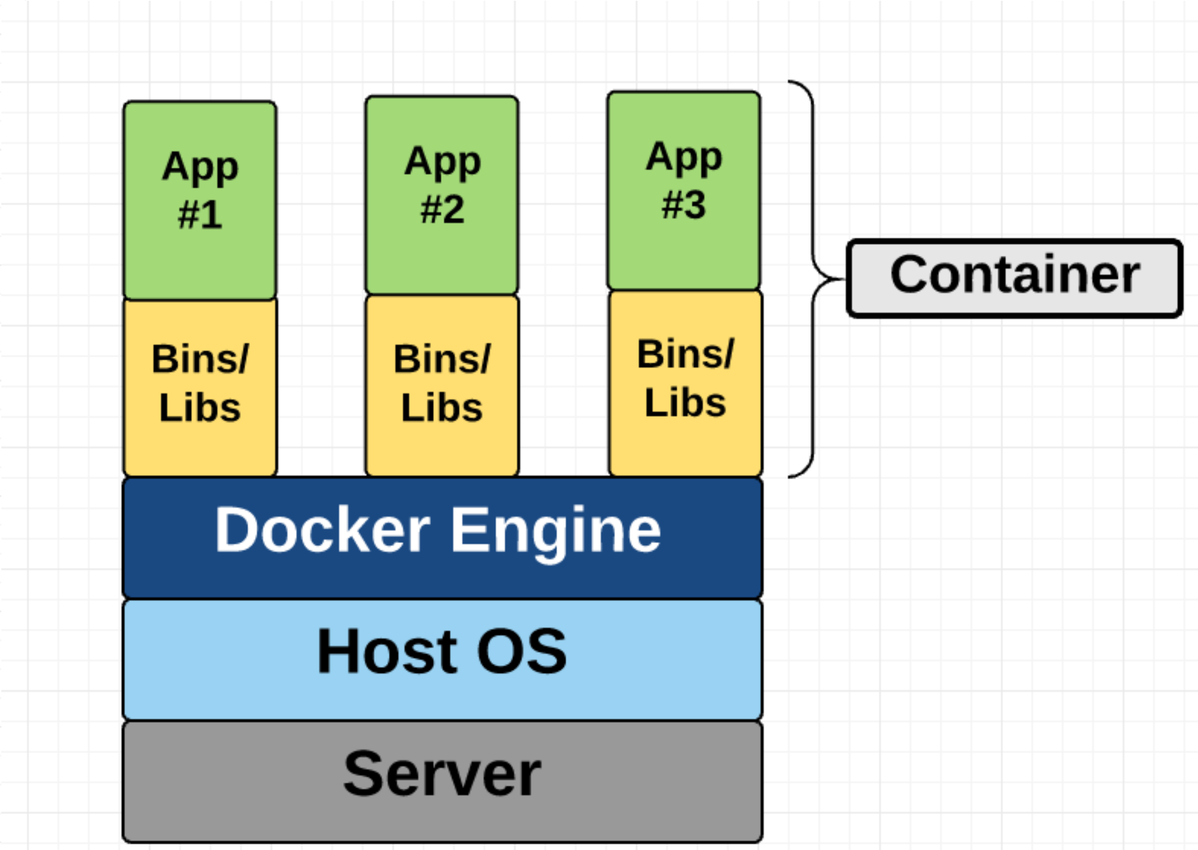
Containers

Managing dependencies with containers (i.e., Docker)

- Docker vs VM
- Dockerfile and layer-based images
- DockerHub and the ecosystem
- Wrappers around Docker for ML



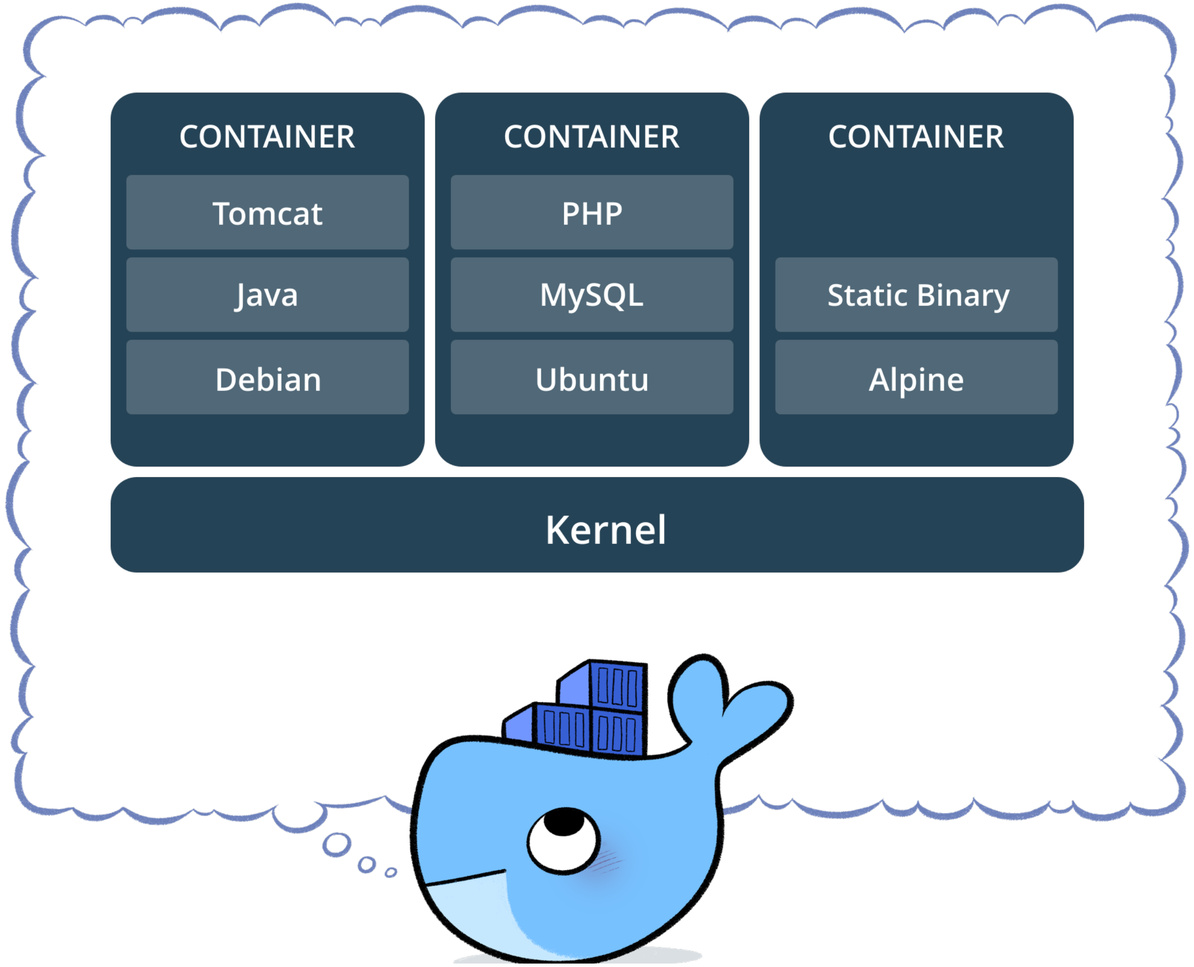
No OS -> Light weight



https://medium.freecodecamp.org/a-beginner-friendly-introduction-to-containers-vms-and-docker-79a9e3e119b

Lightweight -> heavy use

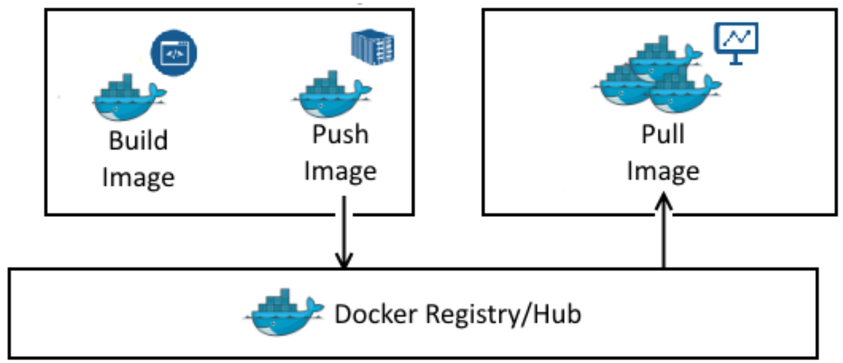
- Spin up a container for every discrete task
- For example, a web app might have four containers:
  - Web server
  - Database
  - Job queue
  - Worker



https://www.docker.com/what-container

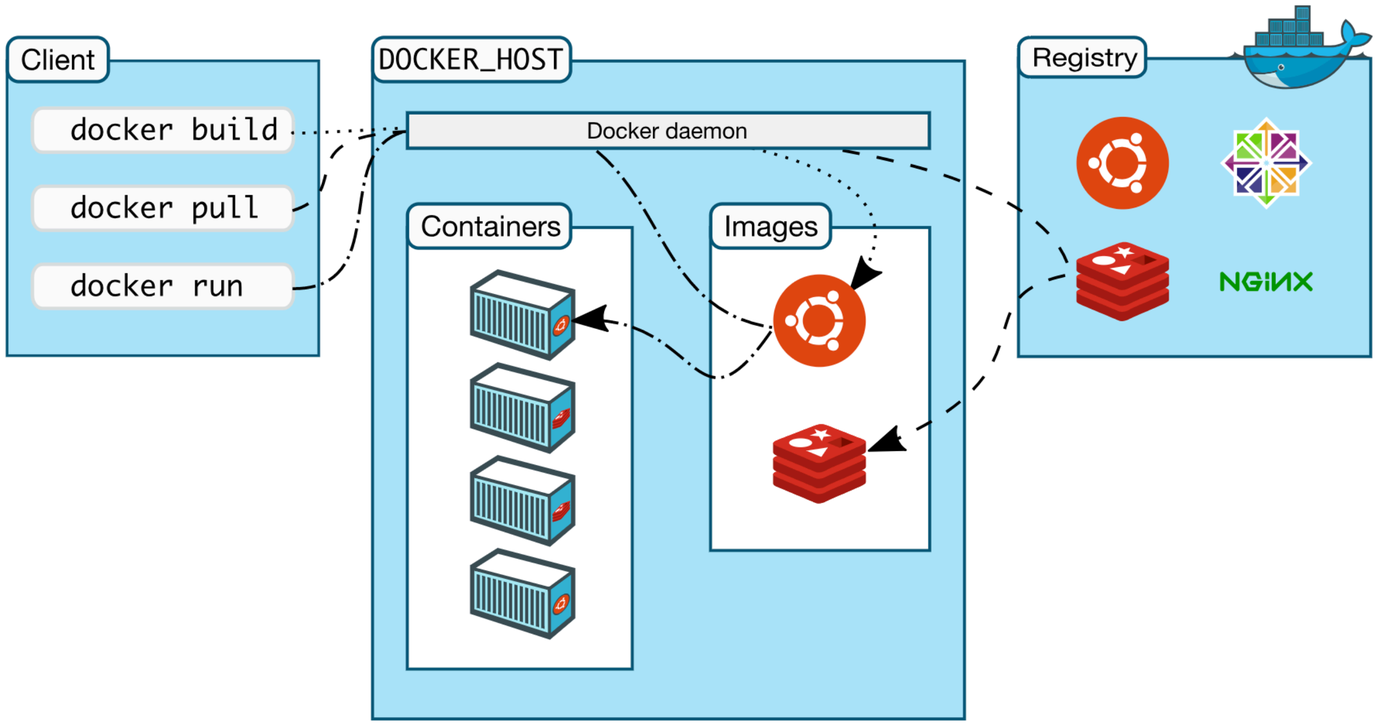
Dockerfile

```
# Use an official Python runtime as a parent image
FROM python:2.7-slim

# Set the working directory to /app
WORKDIR /app

# Copy the current directory contents into the container at /app
ADD . /app

# Install any needed packages specified in requirements.txt
RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Make port 80 available to the world outside this contain
EXPOSE 80

# Define environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python", "app.py"]
```




```
$ docker run -p 4000:80 gordon/get-started:part2
Unable to find image 'gordon/get-started:part2' locally
part2: Pulling from gordon/get-started
10a267c67f42: Already exists
f68a39a6a5e4: Already exists
9beaffc0cf19: Already exists
3c1fe835fb6b: Already exists
4c9f1fa8fcb8: Already exists
ee7d8f576a14: Already exists
fbccdcced46e: Already exists
Digest: sha256:0601c866aab2adcc6498200efd0f754037e909e5fd42069adeff72d1e2439068
Status: Downloaded newer image for gordon/get-started:part2
 * Running on http://0.0.0.0:80/ (Press CTRL+C to quit)
```

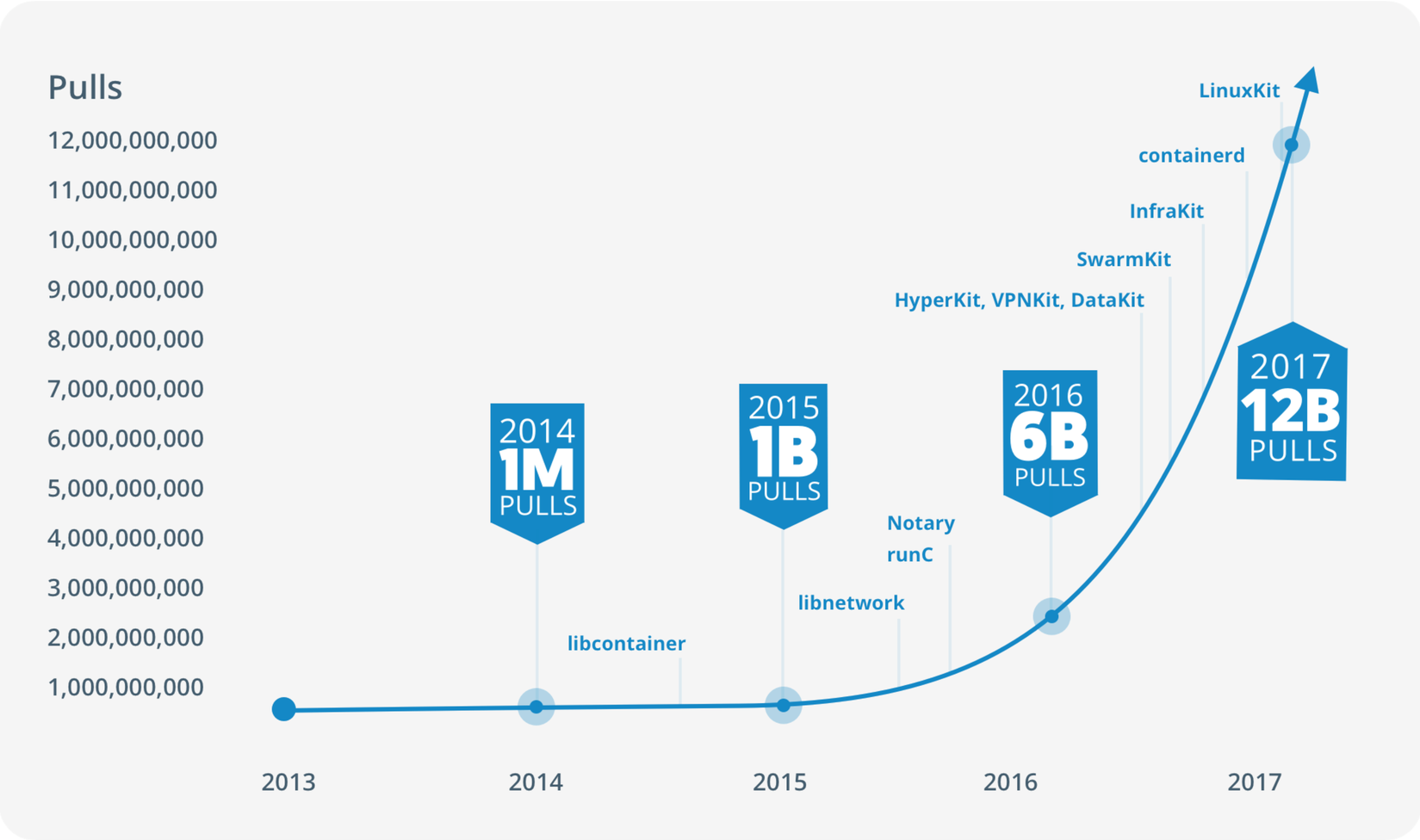
Strong Ecosystem



- Images are easy to find, modify, and contribute back to DockerHub
- Private images easy to store in same place

https://docs.docker.com/engine/docker-overview

Docker is incredibly popular: near ubiquitous



https://www.docker.com/what-container#/package_software

This seems hard, can't we simplify it?

Cog: Containers for machine learning

Cog is an open-source tool that lets you package machine learning models in a standard, production-ready container.

Define the Docker environment your model runs in with `cog.yaml`:

```yaml
build:
  gpu: true
  system_packages:
    - "libgl1-mesa-glx"
    - "libglib2.0-0"
  python_version: "3.8"
  python_packages:
    - "torch==1.8.1"
predict: "predict.py:Predictor"
```

Define how predictions are run on your model with `predict.py`:

```python
from cog import BasePredictor, Input, Path
import torch

class Predictor(BasePredictor):
    def setup(self):
        """Load the model into memory to make running multiple predictions efficient"""
        self.model = torch.load("./weights.pth")

    # The arguments and types the model takes as input
    def predict(self,
            image: Path = Input(title="Grayscale input image")
    ) -> Path:
        """Run a single prediction on the model"""
        processed_image = preprocess(image)
        output = self.model(processed_image)
        return postprocess(output)
```

BentoML

Building a Bento 🍱

Once the service definition is finalized, we can build the model and service into a `bento`. Bento is the distribution format for a service. It is a self-contained archive that contains all the source code, model files and dependency specifications required to run the service.

To build a Bento, first create a `bentofile.yaml` file in your project directory:

```yaml
service: "service:svc"  # Same as the argument passed to `bentoml serve`
labels:
   owner: bentoml-team
   stage: dev
include:
- "*.py"  # A pattern for matching which files to include in the bento
python:
   packages:  # Additional pip packages required by the service
   - scikit-learn
   - pandas
```

TRUSS

```python
!pip install --upgrade scikit-learn truss

import truss
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_iris

# Load the iris data set
iris = load_iris()
data_x = iris['data']
data_y = iris['target']

# Train the model
rfc = RandomForestClassifier()
rfc.fit(data_x, data_y)

# Create the Truss (serializing & packaging model)
tr = truss.mk_truss(rfc, target_directory="iris_rfc_truss")

# Serve a prediction from the model
tr.server_predict({"inputs": [[0, 0, 0, 0]]})
```

```yaml
model_framework: sklearn
model_metadata:
  model_binary_dir: model
  supports_predict_proba: true
python_version: py39
requirements:
- scikit-learn==1.0.2
- threadpoolctl==3.0.0
- joblib==1.1.0
- numpy==1.20.3
- scipy==1.7.3
```

Building a model service: the basics

- REST APIs
- Dependency management
- **Performance optimization**
- Horizontal scaling
- Rollout
- Managed options

Making inference on a single machine more efficient

- GPU or no GPU?
- Concurrency
- Model distillation
- Quantization
- Caching
- Batching
- Sharing the GPU
- Libraries

GPU or no GPU?

- **GPU pros**
  - Same hardware you trained on probably
  - In the limit of model size, batch size tuning, etc usually higher throughput
- **GPU cons**
  - More complex to set up
  - Often more expensive

Just because your model was trained on a GPU, it does not mean you need to serve it on a GPU

Concurrency

- **What?**
  - Multiple copies of the model running on different CPUs or cores
- **How?**
  - Be careful about thread tuning

```python
import torch

class BertModelInference:
  def __init__(self, model_path, do_quantize=False):

    # Omitting code that loads the Bert model, for clarity

    torch.set_num_threads(1)

  def predict(self, message: str) -> List[float]:
    # Omitting code that calls the Bert model, for clarity
    ...
```

https://blog.roblox.com/2020/05/scaled-bert-serve-1-billion-daily-requests-cpus/

Model distillation

- **What?**
  - Train a smaller model to imitate your larger one
- **How?**
  - Several techniques outlined below
  - Can be finicky to do yourself — infrequently used in practice
  - Exception — pretrained distilled models like DistilBERT

https://heartbeat.fritz.ai/research-guide-model-distillation-techniques-for-deep-learning-4a100801c0eb

Quantization

- **What?**
  - Execute some or all of the operations in your model with a smaller numerical representation than floats (e.g., INT8)
  - Some tradeoffs with accuracy
- **How?**
  - PyTorch and Tensorflow Lite have quantization built-in
  - Can also run **quantization-aware training,** which often results in higher accuracy

https://pytorch.org/blog/introduction-to-quantization-on-pytorch/

Quantization: tools

🤗 Optimum

🤗 Optimum is an extension of 🤗 Transformers, providing a set of performance optimization tools enabling maximum efficiency to train and run models on targeted hardware.

Features	ONNX Runtime	Intel Neural Compressor
Post-training Dynamic Quantization	✅	✅
Post-training Static Quantization	✅	✅
Quantization Aware Training (QAT)	Stay tuned! ⭐	✅
Pruning	N/A	✅

Caching

- **What?**
  - For some ML models, some inputs are more common than others
  - Instead of always calling the model, first check the cache
- **How?**
  - Can get very fancy
  - Basic way uses functools

`@functools.cache(user_function)`

Simple lightweight unbounded function cache. Sometimes called "memoize".

Returns the same as `lru_cache(maxsize=None)`, creating a thin wrapper around a dictionary lookup for the function arguments. Because it never needs to evict old values, this is smaller and faster than `lru_cache()` with a size limit.

For example:

```
@cache
def factorial(n):
    return n * factorial(n-1) if n else 1

>>> factorial(10)      # no previously cached result, makes 11 recursive calls
3628800
>>> factorial(5)       # just looks up cached value result
120
>>> factorial(12)      # makes two new recursive calls, the other 10 are cached
479001600
```

New in version 3.9.

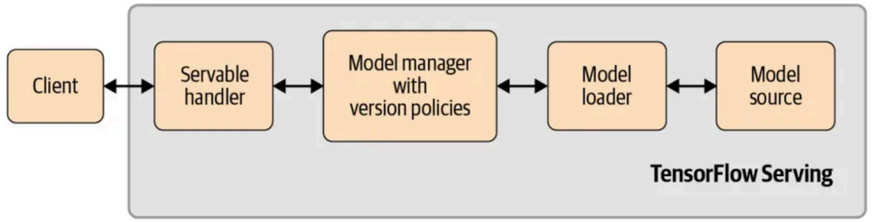
Batching

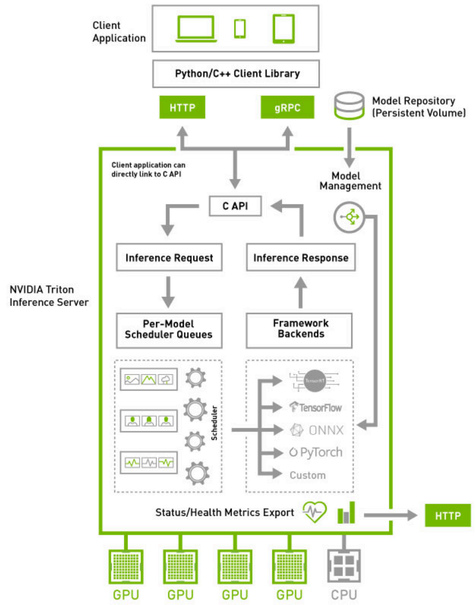
- **What?**
  - ML models often achieve higher throughput when doing prediction in parallel, especially in a GPU
- **How?**
  - Gather predictions until you have a batch, run prediction, return to user
  - Batch size needs to be tuned
  - You need to have a way to shortcut the process if latency becomes too long
  - Probably don’t want to implement this yourself

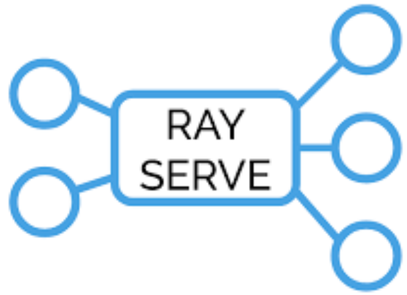
Sharing the GPU

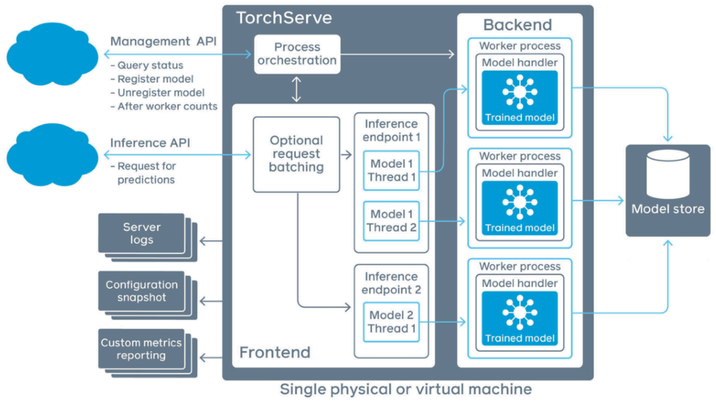
- **What?**
  - Your model may not take up all of the GPU memory with your inference batch size. Why not run multiple models on the same GPU?
- **How?**
  - You'll probably want to use a model serving solution that supports this out of the box

Model serving libraries









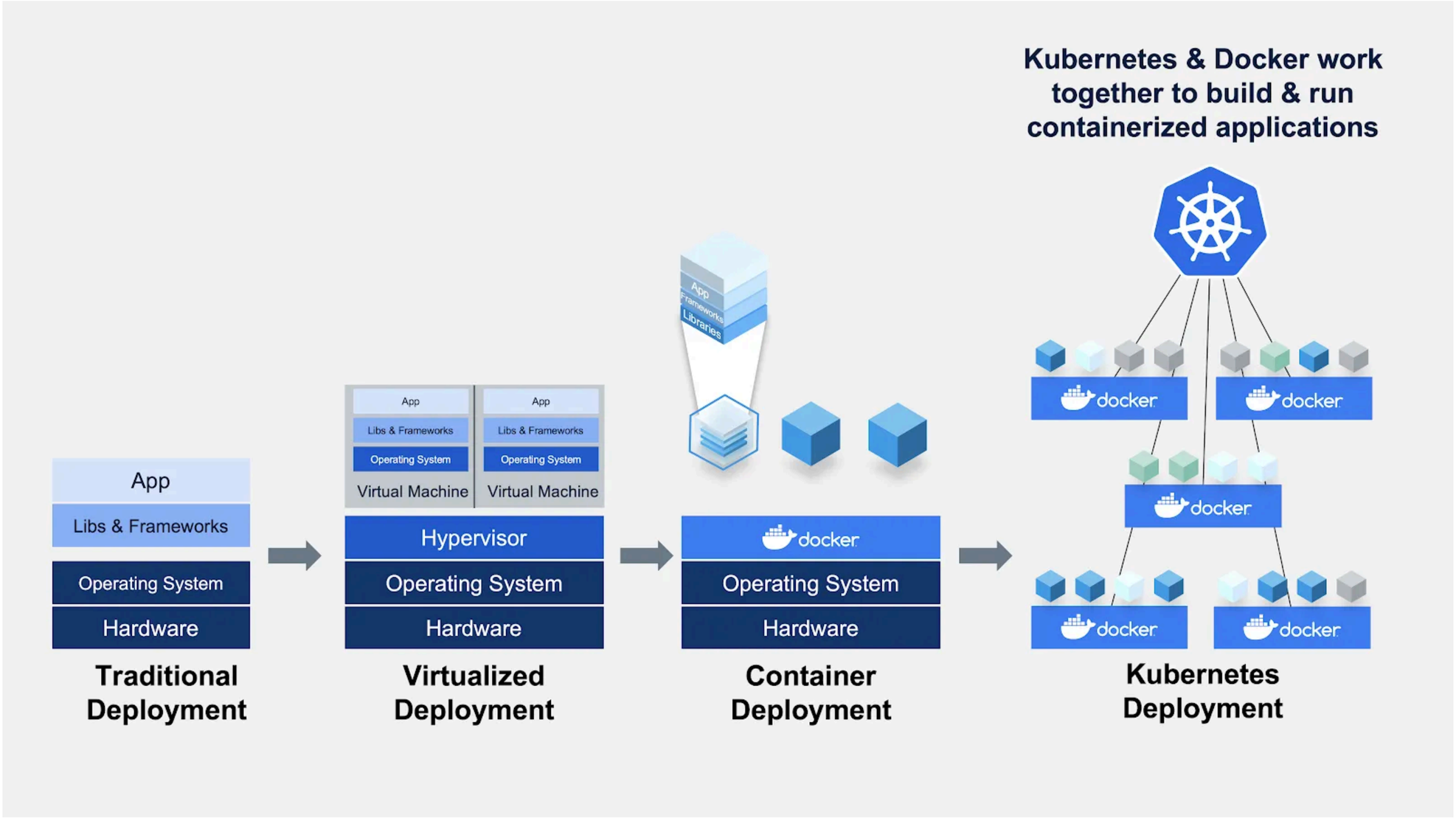
Building a model service: the basics

- REST APIs
- Dependency management
- Performance optimization
- **Horizontal scaling**
- Deployment
- Managed options

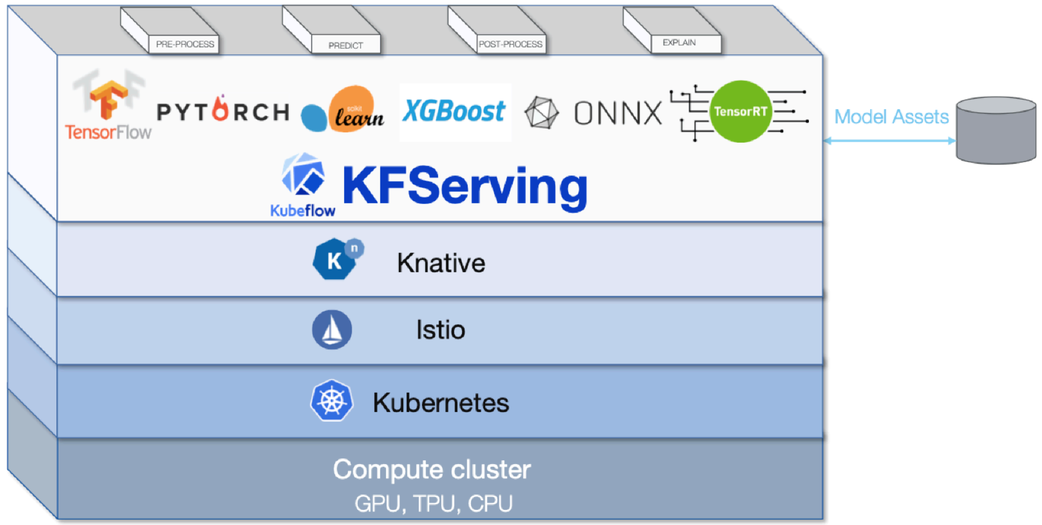
Horizontal scaling

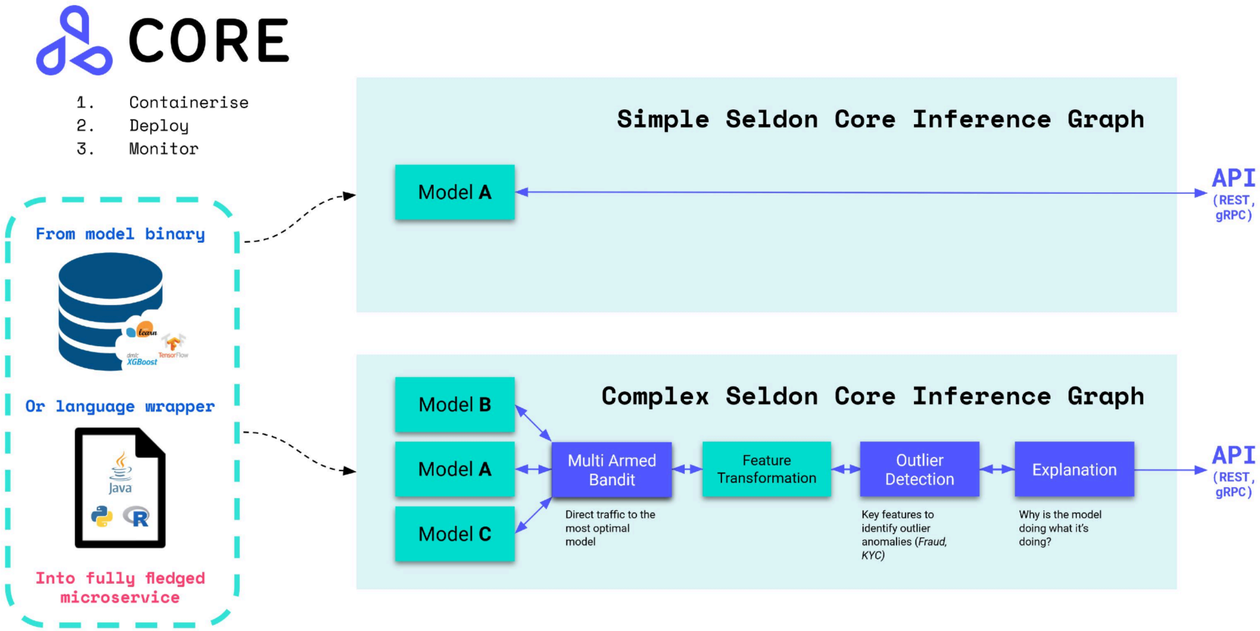
- **What?**
  - If you have too much traffic for a single machine, split traffic among multiple machines
- **How?**
  - Spin up multiple copies of your service and split traffic using a load balancer
  - In practice, two common methods
    - Container orchestration (i.e., Kubernetes)
    - Serverless (e.g., AWS Lambda)

Container orchestration

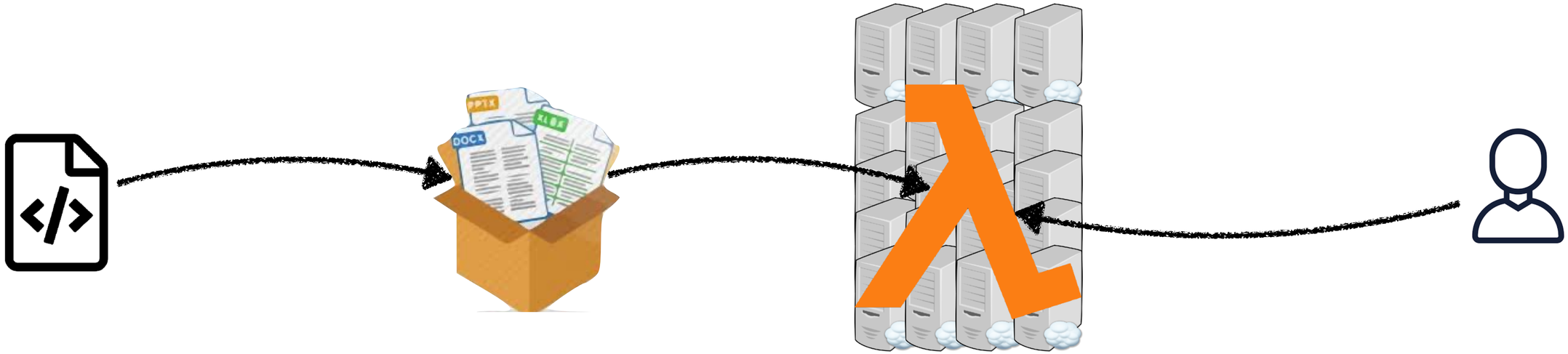


Frameworks for ML deployment on kubernetes



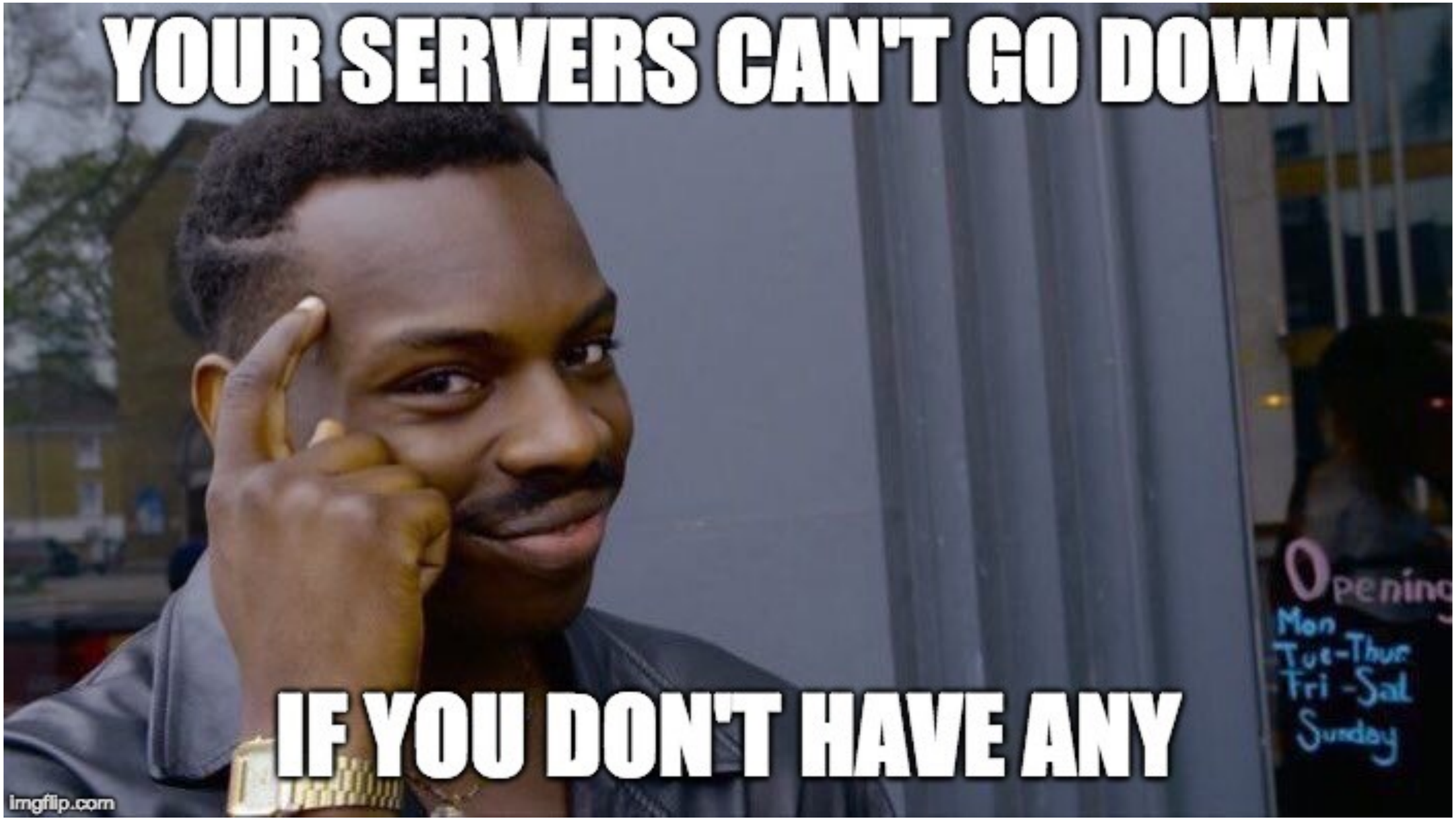


Deploying code as serverless functions

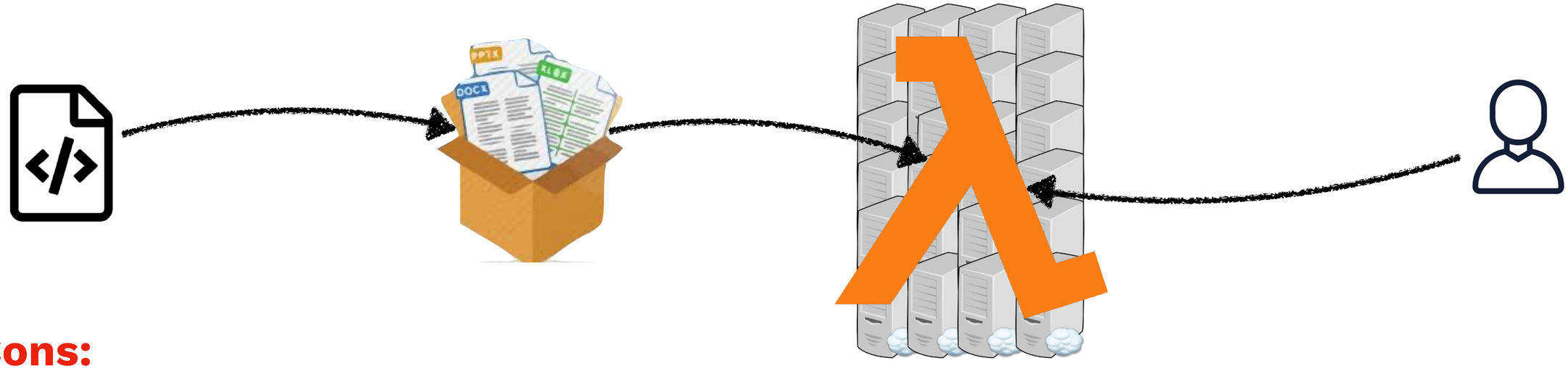


- App code and dependencies are packaged into .zip files or docker containers with a single entry point function
- AWS Lambda (or Google Cloud Functions, or Azure Functions) manages everything else: instantiating to 10,000+ requests per second, load balancing, etc.
- Only pay for compute-time

Start here!

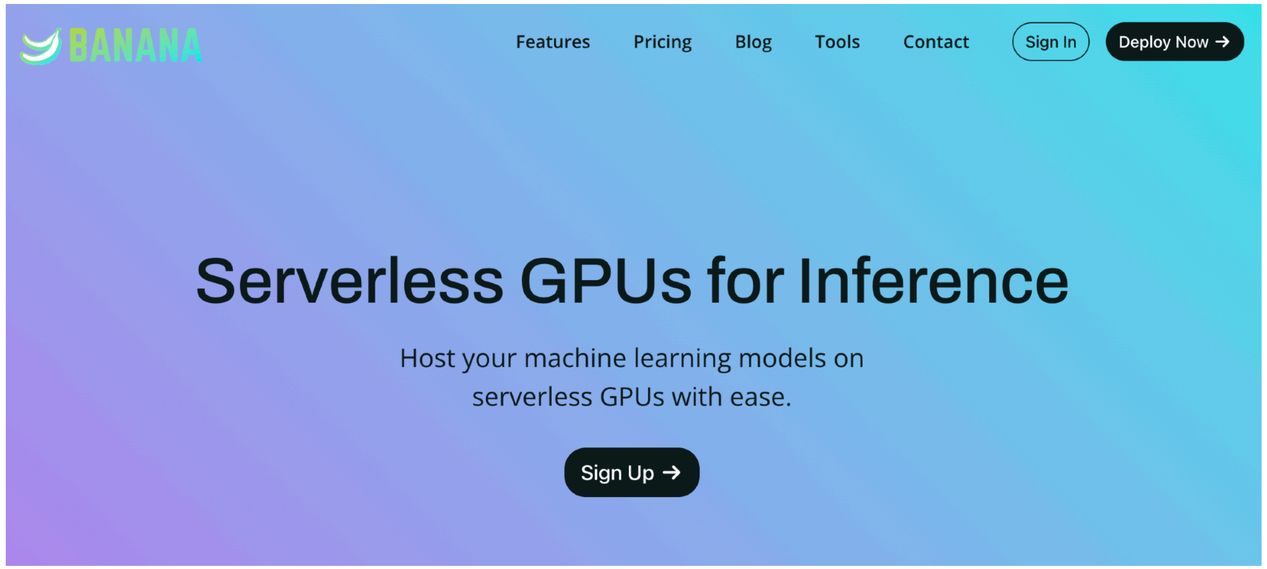

YOUR SERVERS CAN'T GO DOWN
IF YOU DON'T HAVE ANY
Opening
Mon
Tue-Thur
Fri -Sat
Sunday
imgflip.com

Deploying code as serverless functions



- **Cons:**
  - Limited size of deployment package
  - Cold start
  - Can be challenging to build pipelines of models
  - Little to no state management (e.g., for caching)
  - Limited deployment tooling
  - CPU-only, limited execution time

How far away are we from serverless GPUs?



Building a model service: the basics

- REST APIs
- Dependency management
- Performance optimization
- Horizontal scaling
- **Rollout**
- Managed options

Model rollouts

- **What?**
  - If serving is how you turn a model into something that can respond to requests, **rollouts** is how you manage, and update these services
- **How?**
  - You probably want to be able to **roll out gradually**, **roll back instantly**, **split traffic between versions, and deploy pipelines of models**
  - This is a challenging infra problem, and beyond the scope of this lecture
  - Your deployment library (or infra team) may take care of this for you

Building a model service: the basics

- REST APIs
- Dependency management
- Performance optimization
- Horizontal scaling
- Rollout
- **Managed options**

Managed options

Cloud providers

Google AI

Azure Machine Learning

End-to-end ML platforms

databricks

DataRobot

Startups

cortex

BENTOML

BANANA

SELDON

NEURAL MAGIC

Double-click on Sagemaker

Deploy after training

To deploy your model directly after training, ensure all required files are saved in your training script, including the tokenizer and the model.

If you use the Hugging Face `Trainer`, you can pass your tokenizer as an argument to the `Trainer`. It will be automatically saved when you call `trainer.save_model()`.

```
from sagemaker.huggingface import HuggingFace

############ pseudo code start ############

# create Hugging Face Estimator for training
huggingface_estimator = HuggingFace(....)

# start the train job with our uploaded datasets as input
huggingface_estimator.fit(...)

############ pseudo code end ############

# deploy model to SageMaker Inference
predictor = hf_estimator.deploy(initial_instance_count=1, instance_type="ml.m5.xlarge")

# example request: you always need to define "inputs"
data = {
   "inputs": "Camera - You are awarded a SiPix Digital Camera! call 09061221066 fromm landline. Delive
}

# request
predictor.predict(data)
```

It’s easy, but more expensive

- ~50-100% more expensive than raw EC2, depending on instance type
- Serverless is a better deal: ~20% more expensive than Lambda

Building a model service: takeaways

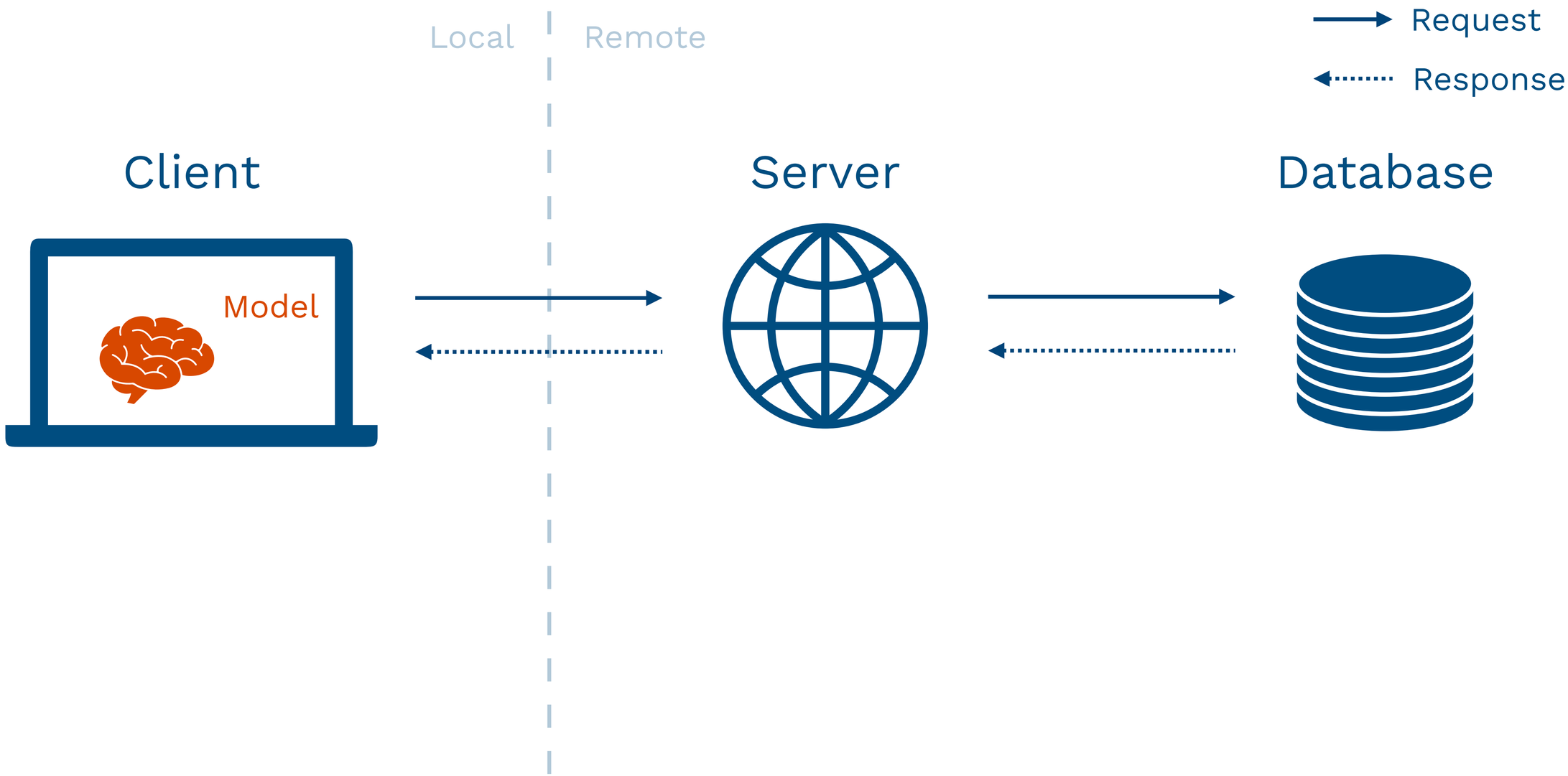
- If you are doing CPU inference, can get away with scaling by launching more servers, or going serverless
- Serverless makes sense if you can get away with CPUs and traffic is spiky or low-volume
- Sagemaker is a perfectly good way to get started if you're on AWS, but can get expensive
- If using GPU inference, serving tools like TF serving, Triton, and torch serve will save you time
- Worth keeping an eye on the startups in this space for GPU inference

Step 3: move to the edge?

When should you consider edge deployment?

- Sometimes, it's obvious
  - No reliable internet connection
  - Very strict data security / privacy requirements
- Otherwise:
  - Accuracy and latency both affect end-user experience
  - Latency includes the network roundtrip and model prediction time
  - Once you've exhausted options for reducing model pred time, consider edge

Edge prediction



Edge prediction

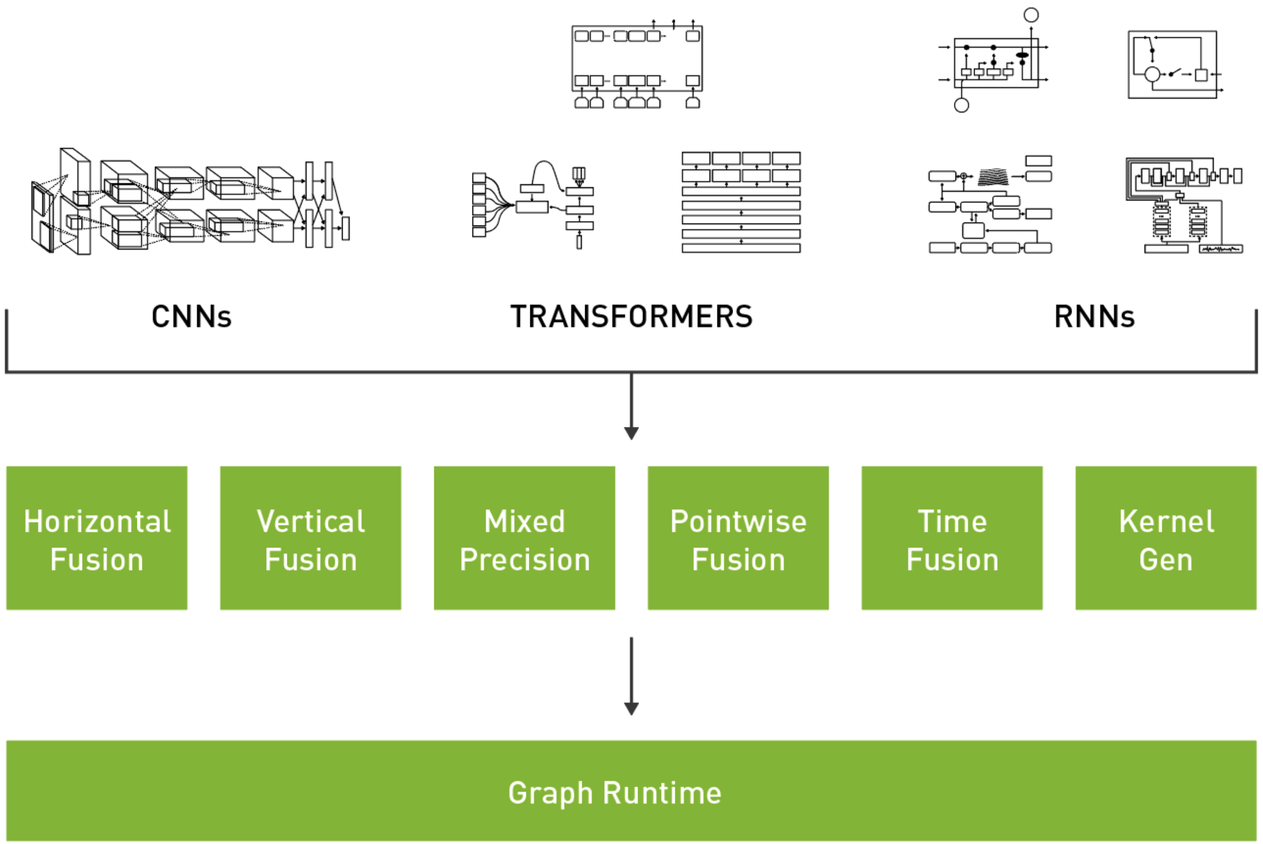
- Send model weights to the client device
- Client loads the model and interacts with it directly

Edge deployment

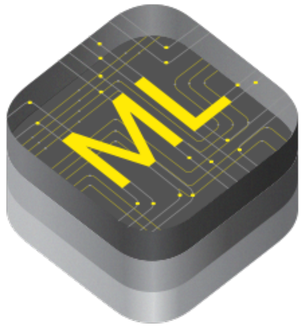
Pros	Cons
• Lowest-latency • Does not require an internet connection • Data security — data doesn't need to leave the user's device • Scale comes "for free"	• Often limited hardware resources available • Embedded and mobile frameworks are less full featured than tensorflow / pytorch • Difficult to update models • Difficult to monitor and debug when things go wrong

Frameworks

Tools for edge deployment

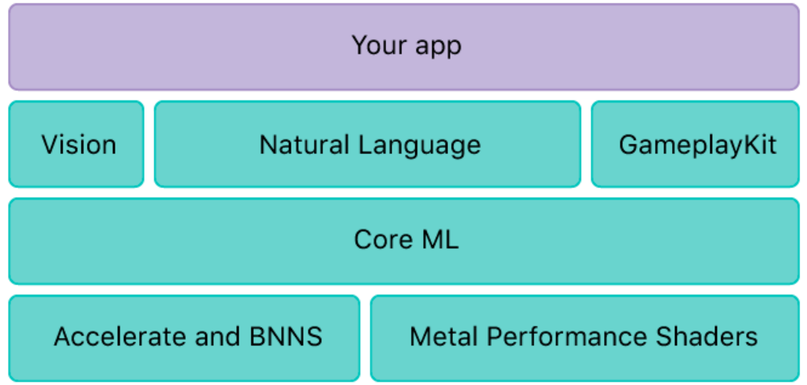


TensorRT: Model optimizer and inference runtime for tensor flow + NVIDIA devices



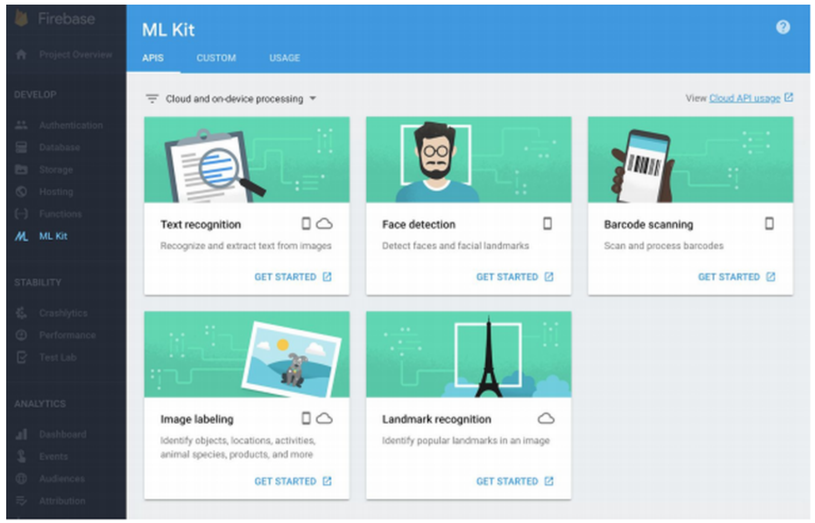
CoreML

- Released at Apple WWDC 2017
- Inference only
- https://coreml.store/

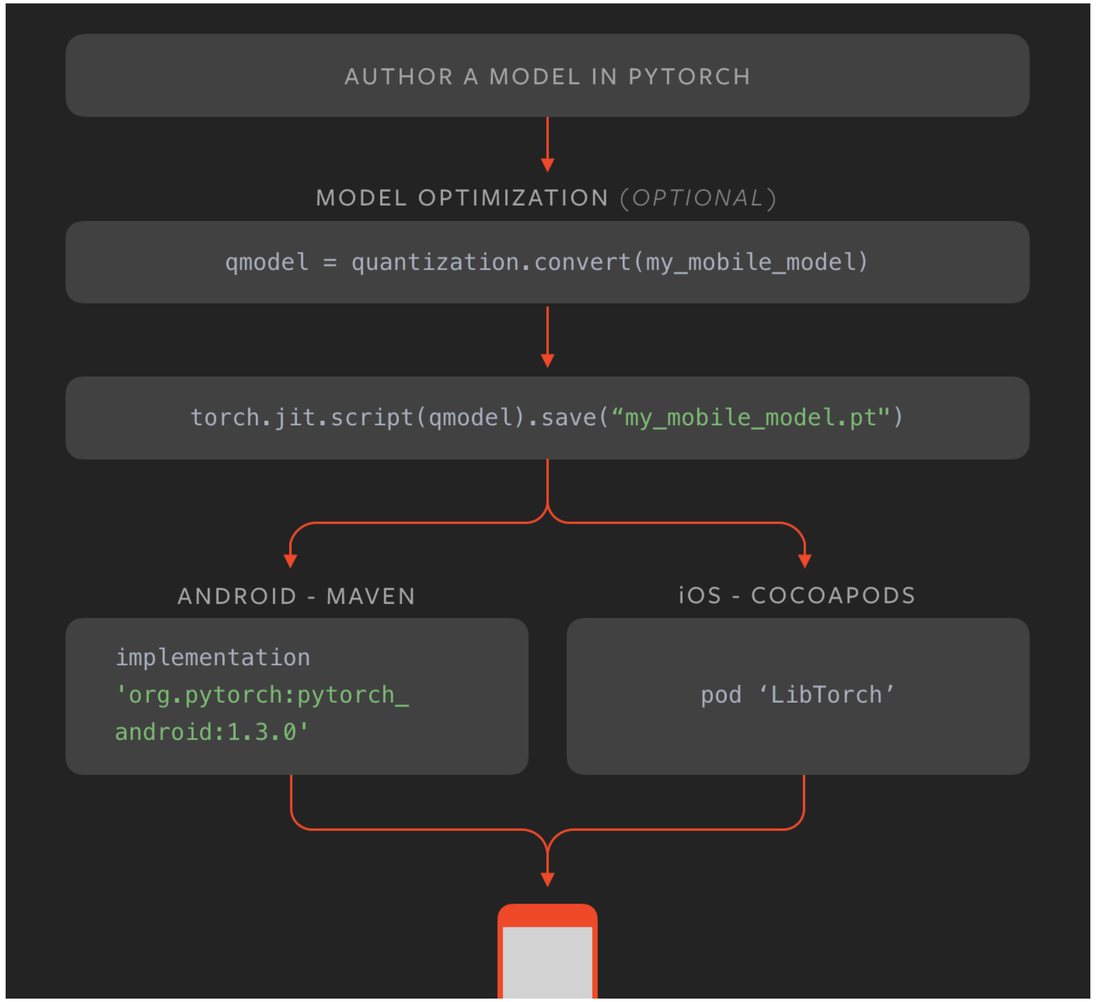


ML Kit

- Announced at Google I/O 2018
- Either via API or on-device
- Offers pre-trained models, or can upload Tensorflow Lite model

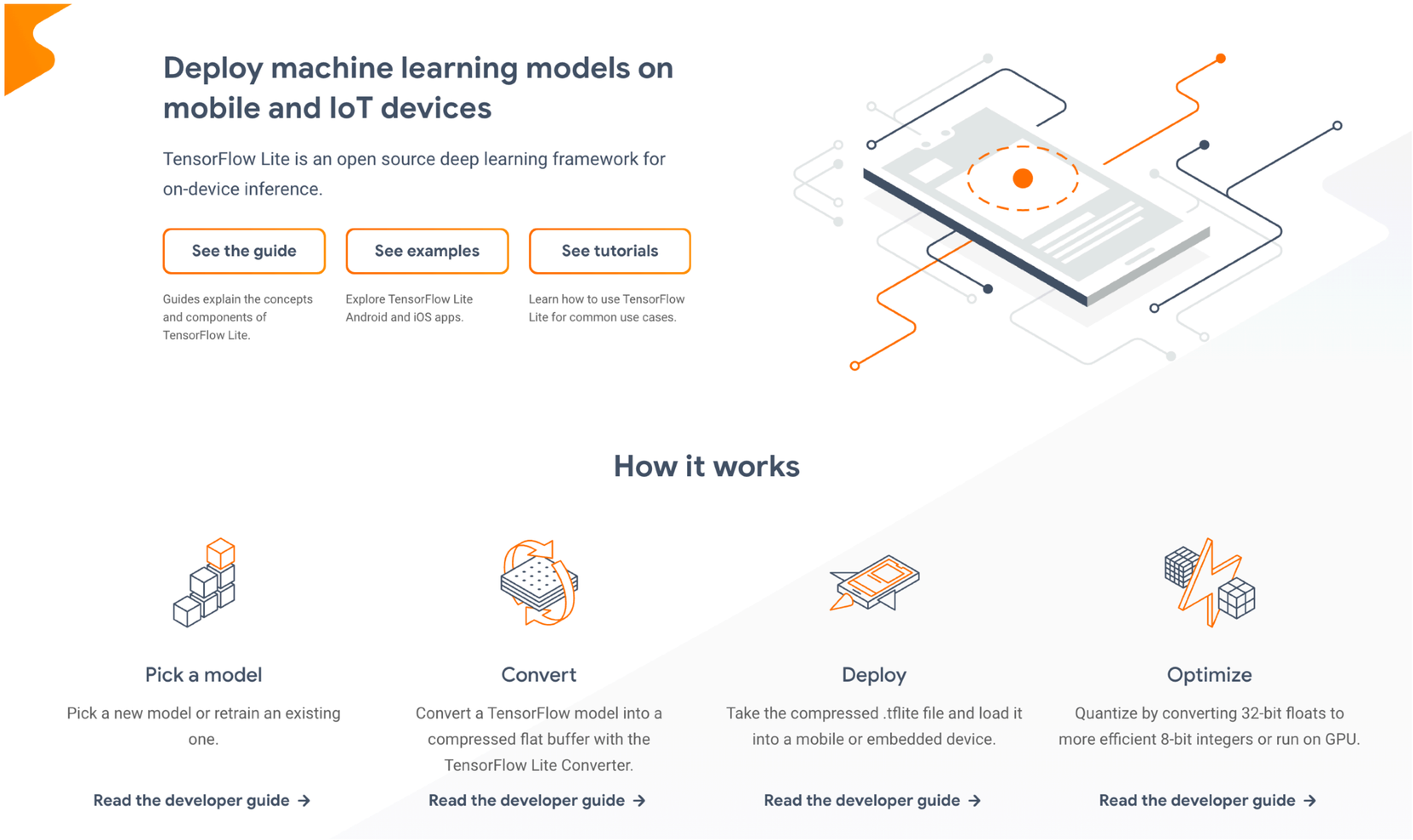


Tools for edge deployment



PyTorch mobile: PyTorch on iOS and Android

Tools for edge deployment



TFLite: tensor flow on mobile / edge devices

Tools for edge deployment

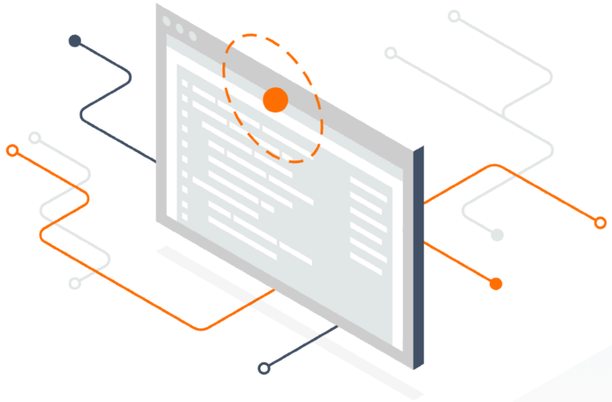
TensorFlow.js is a library for machine learning in JavaScript

Develop ML models in JavaScript, and use ML directly in the browser or in Node.js.

See tutorials — Tutorials show you how to use TensorFlow.js with complete, end-to-end examples.

See models — Pre-trained, out-of-the-box models for common use cases.

See demos — Live demos and examples run in your browser using TensorFlow.js.



How it works

Run existing models

Use off-the-shelf JavaScript models or convert Python TensorFlow models to run in the browser or under Node.js.

Use official TensorFlow.js models →
Convert Python models →

Retrain existing models

Retrain pre-existing ML models using your own data.

Use Transfer Learning to customize models →

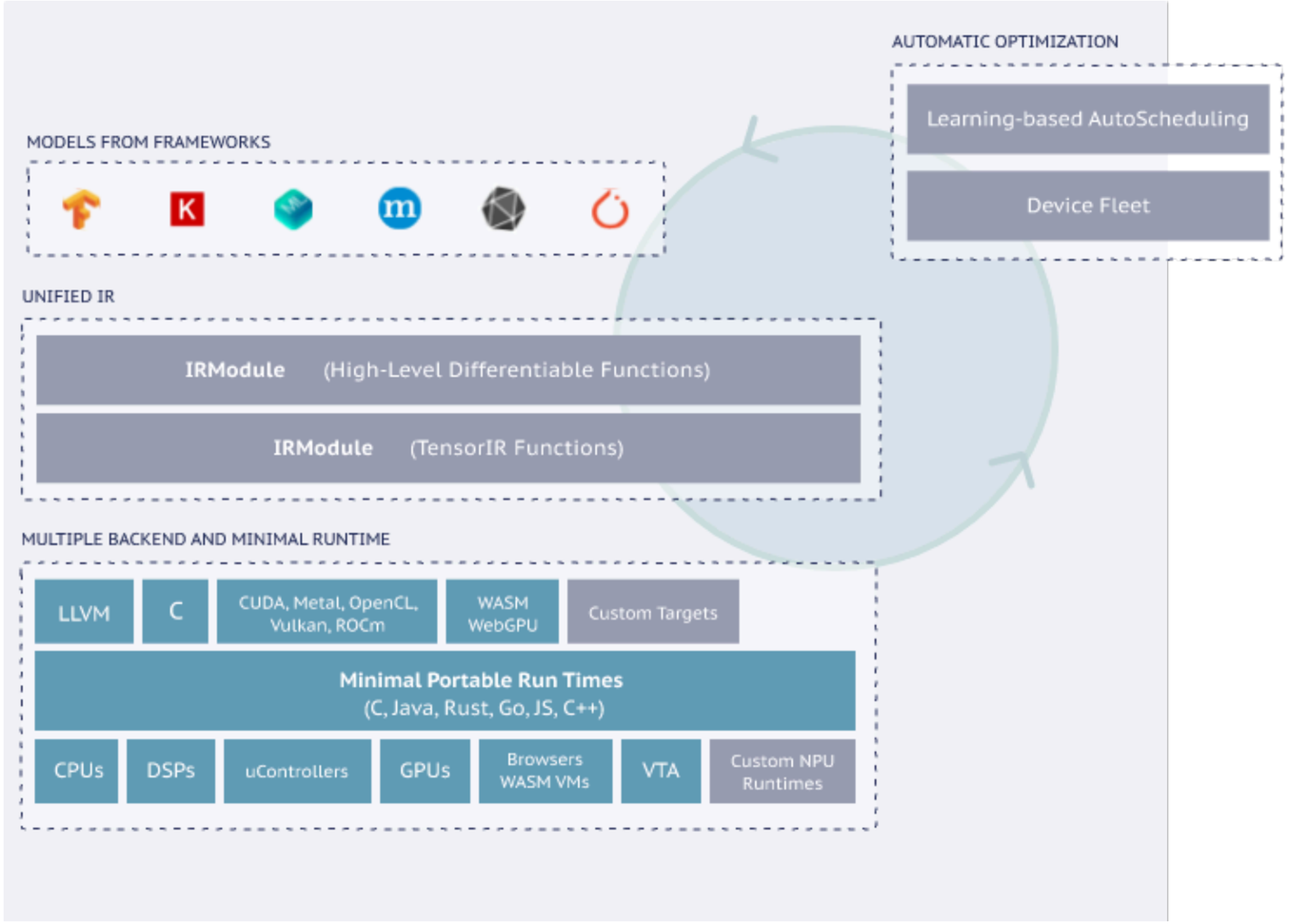
Develop ML with JavaScript

Build and train models directly in JavaScript using flexible and intuitive APIs.

Get started with TensorFlow.js →

TensorFlow.js: tensorflow in the browser

Tools for edge deployment



Apache TVM: Library-agnostic and target-device agnostic inference runtime

Watch this space

MLIR

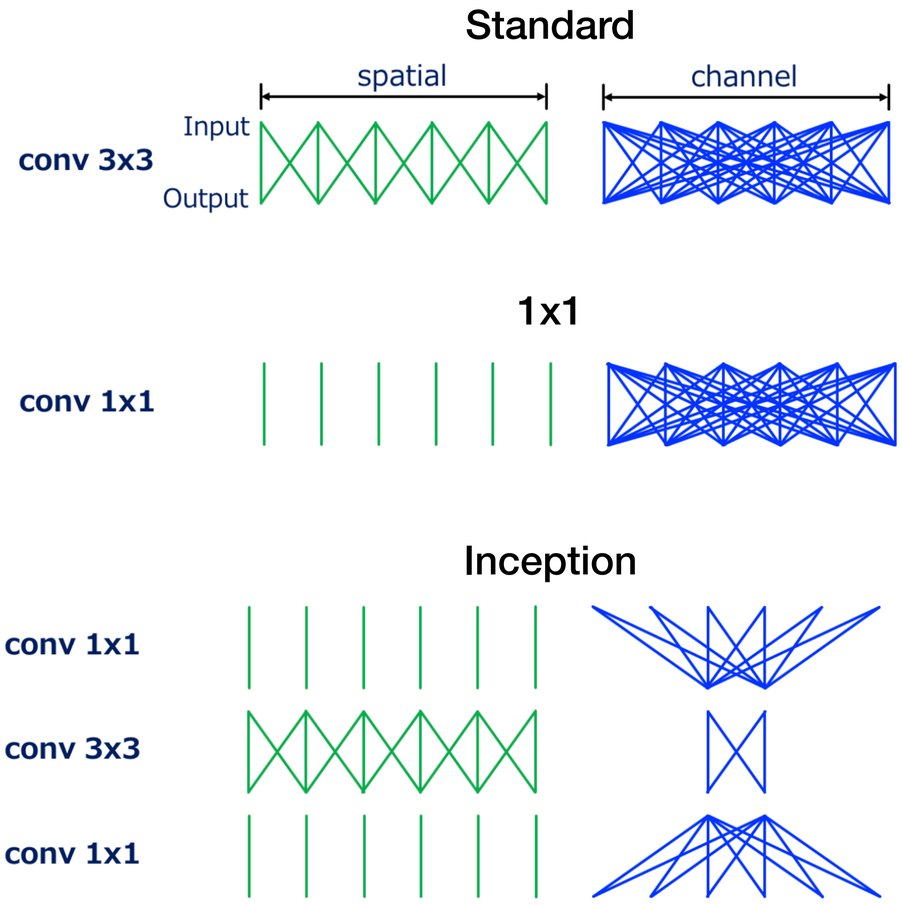
Modular

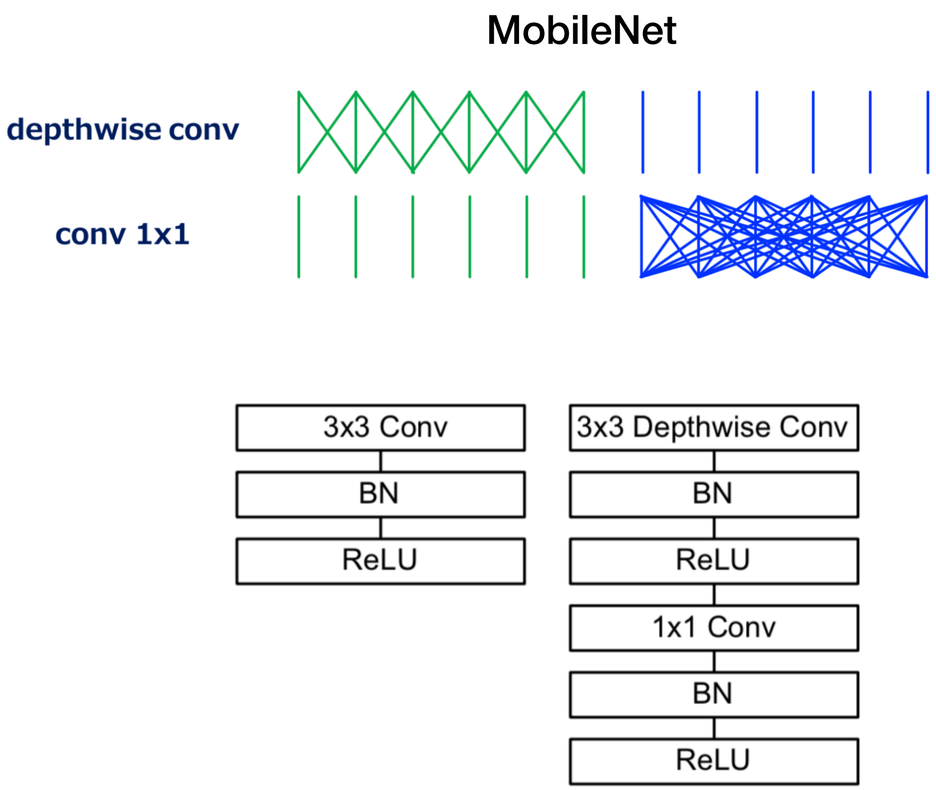
Efficiency

More efficient models

- Quantization and distillation from above
- Mobile-friendly model architectures

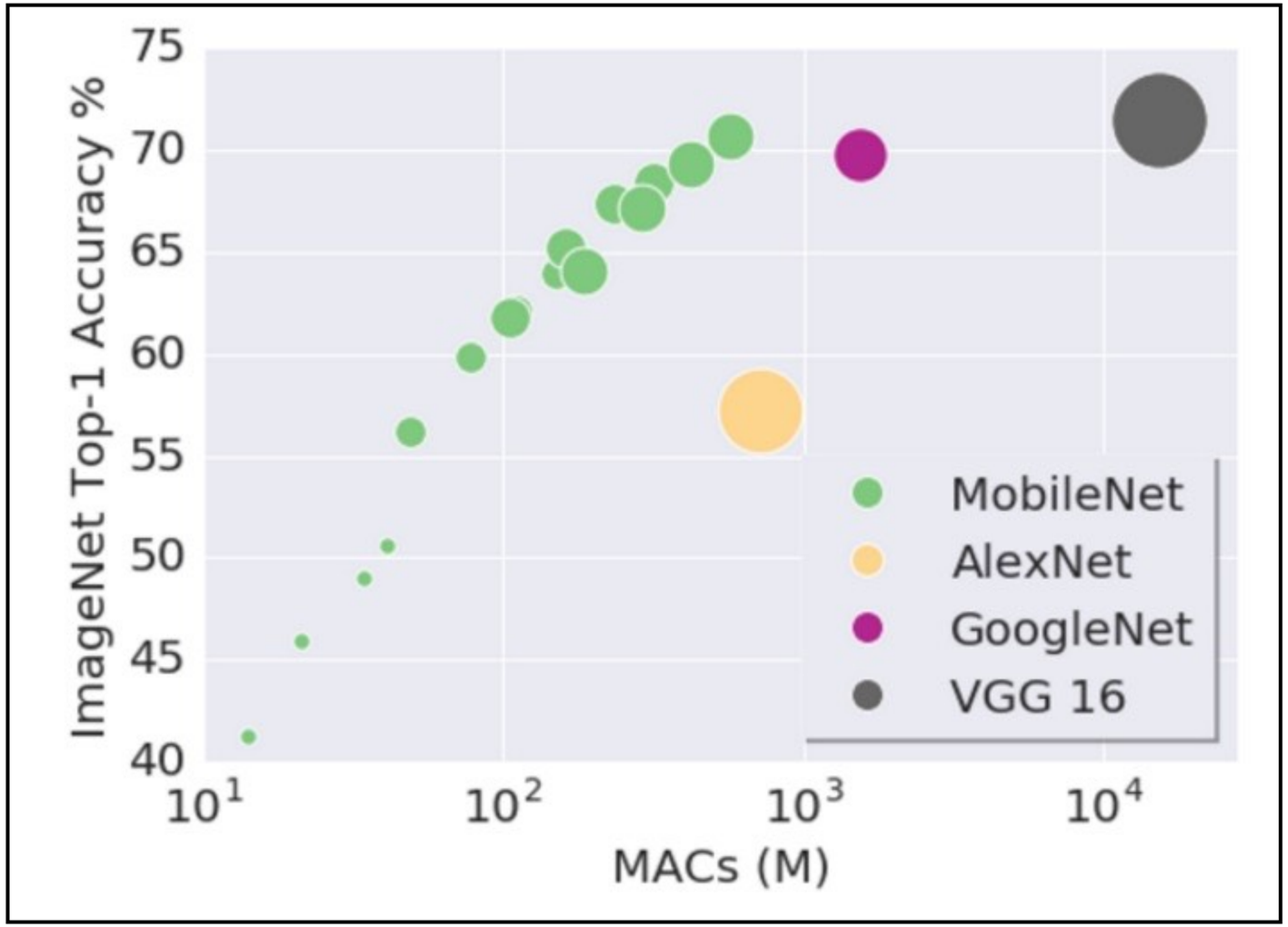
MobileNets





https://medium.com/@yu4u/why-mobilenet-and-its-variants-e-g-shufflenet-are-fast-1c7048b9618d

MobileNets



https://medium.com/@yu4u/why-mobilenet-and-its-variants-e-g-shufflenet-are-fast-1c7048b9618d

Recommended case study: DistillBERT

Using the teacher signal, we are able to train a **smaller language model**, we call **DistilBERT**, from the **supervision of BERT** 👩‍🍼 (we used the English `bert-base-uncased` version of BERT).

Following Hinton et al., the training loss is a linear combination of the *distillation loss* and the masked *language modeling loss*. Our student is a small version of BERT in which we *removed the token-type embeddings and the pooler* (used for the next sentence classification task) and kept the rest of the architecture identical while reducing the numbers of layers by a factor of two.

Overall, our distilled model, DistilBERT, has about half the total number of parameters of BERT base and retains 95% of BERT's performances on the language understanding benchmark GLUE.

https://medium.com/huggingface/distilbert-8cf3380435b5

Mindsets for edge deployment

- Choose your architecture with your target hardware in mind
  - You can make up a factor of 2-10 through distillation, quantization, and other tricks, but not more than that
- Once you have a model that works on your edge device, you can iterate locally as long as you add model size and latency to your metrics and avoid regressions
- Treat tuning the model for your device as an additional risk in the deployment cycle and test it accordingly
  - E.g., always test your models on production hardware before deploying
- Since models can be finicky, it's a good idea to build fallback mechanisms into the application in case the model fails or is too slow

Edge deployment: conclusion

- Web deployment is easier, so use it if you need to
- Choose your framework to match the available hardware and corresponding mobile frameworks, or try TVM to be more flexible
- Start considering hardware constraints at the beginning of the project and choose architectures accordingly

Only IRL do you see how your model **actually** works

- Deploy early, deploy often
- Keep it simple, and add complexity later
  - Build a prototype
  - Separate your model and UI
  - Learn the tricks to scale
  - Consider moving your model to the edge when you *really* need to go fast ⚡