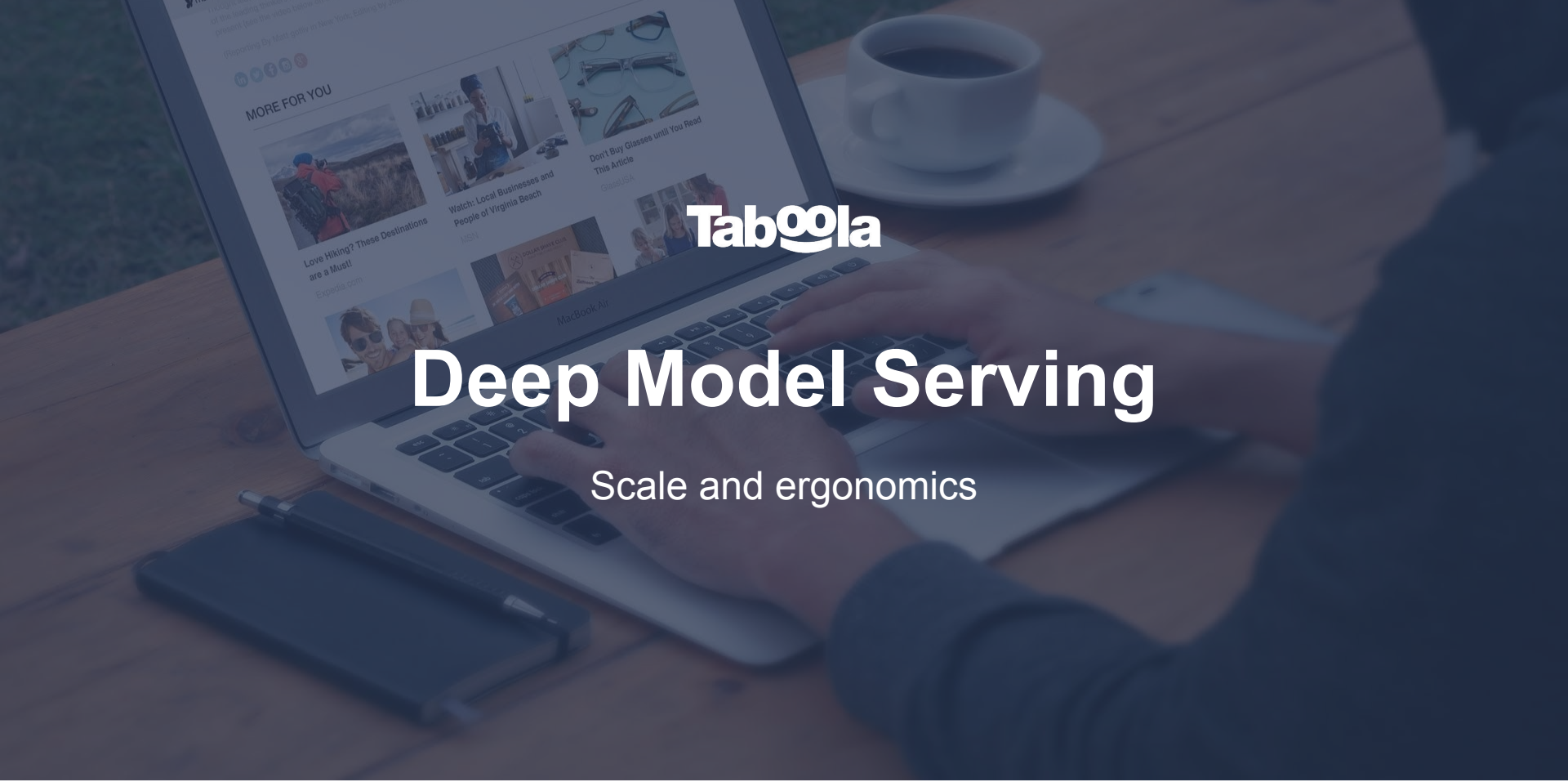


Taboola

Deep Model Serving

Scale and ergonomics

Jenia Gorokhovsky - Algorithms Team Lead @ Taboola

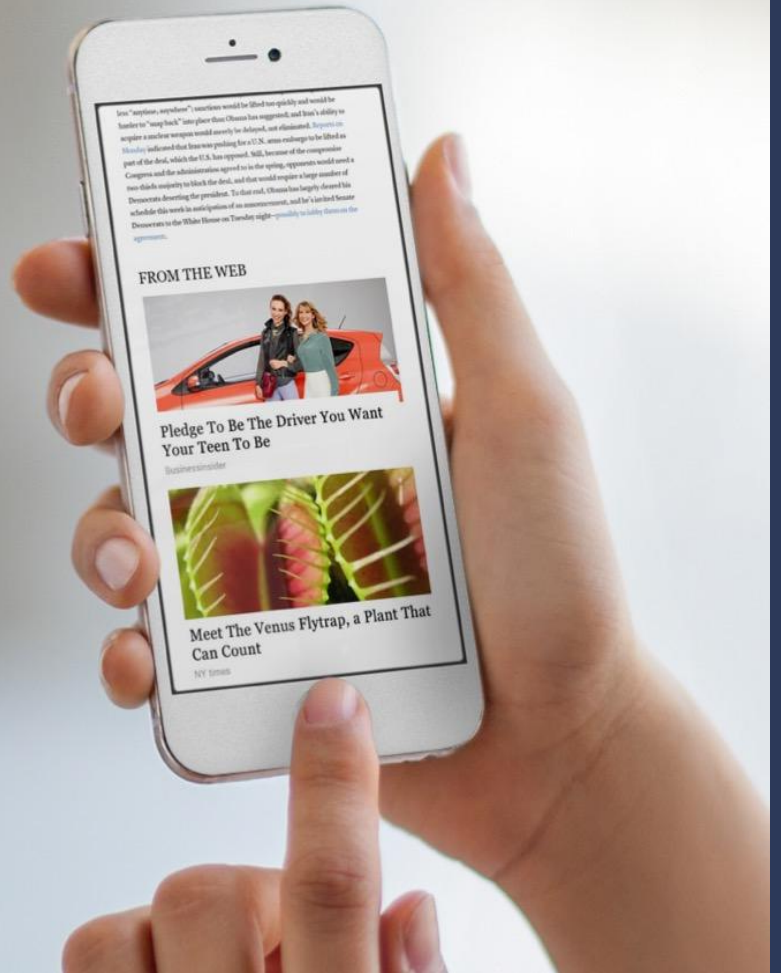
June 2018

Outline

- Problem setup
- What do we want from deep model serving
  - From the Data Science Perspective
  - From the Engineering Perspective
- Design Considerations
- Implementation

You've Seen Us Before

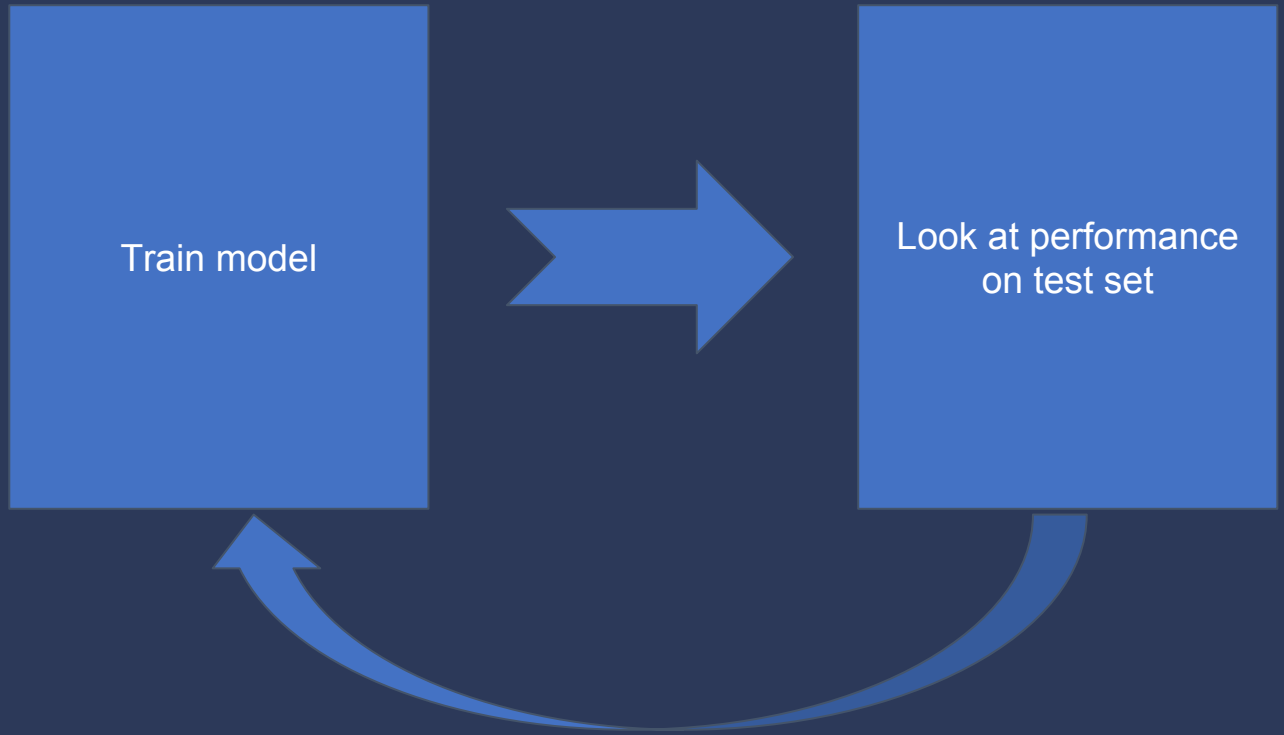
Enabling people to discover information at that moment when they're likely to engage



AIgo Engineer =
Software Engineer + Data Scientist

Online vs. Offline Data Science

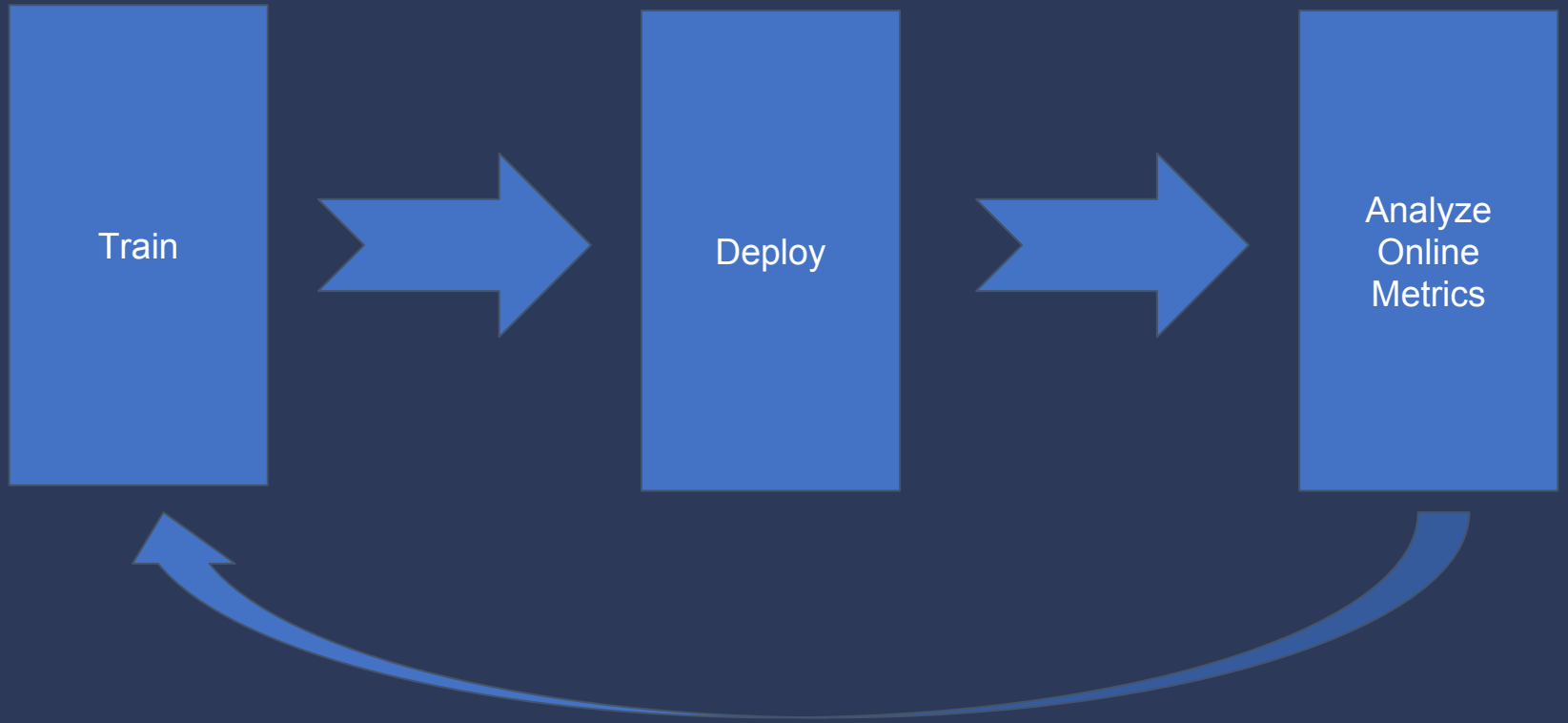
Offline Data Science Cycle



Why go for online experiments?

- Many models are not measured by test set accuracy
  - Recommendation systems
  - Digital assistants
  - Art generation services
- Always useful to see how models are actually used

Online Data Science Cycle



Serving
for
Data Scientists

A Simple Recommender Flow

Model uses *title* and *time of day* to predict click probability

For ex. P(click | <“Buy shaving razors”, 7:30am>)

A Simple Recommender Flow

Items with highest probability are displayed

A Simple Recommender Flow

Report the time, date, and how many items were clicked on my model

Non-trivial Experiment Design

- Timeframes matter a lot
  - Model freshness
  - Time of day / day of week / holidays
- Stickiness sometimes matters
- Traffic required to reach confidence
  - Depends on variance/measurement noise
  - Depends on independence assumptions
- Traffic for experiments might need to be fixed

Scale for data science

Concurrency - how many different experiments are running?

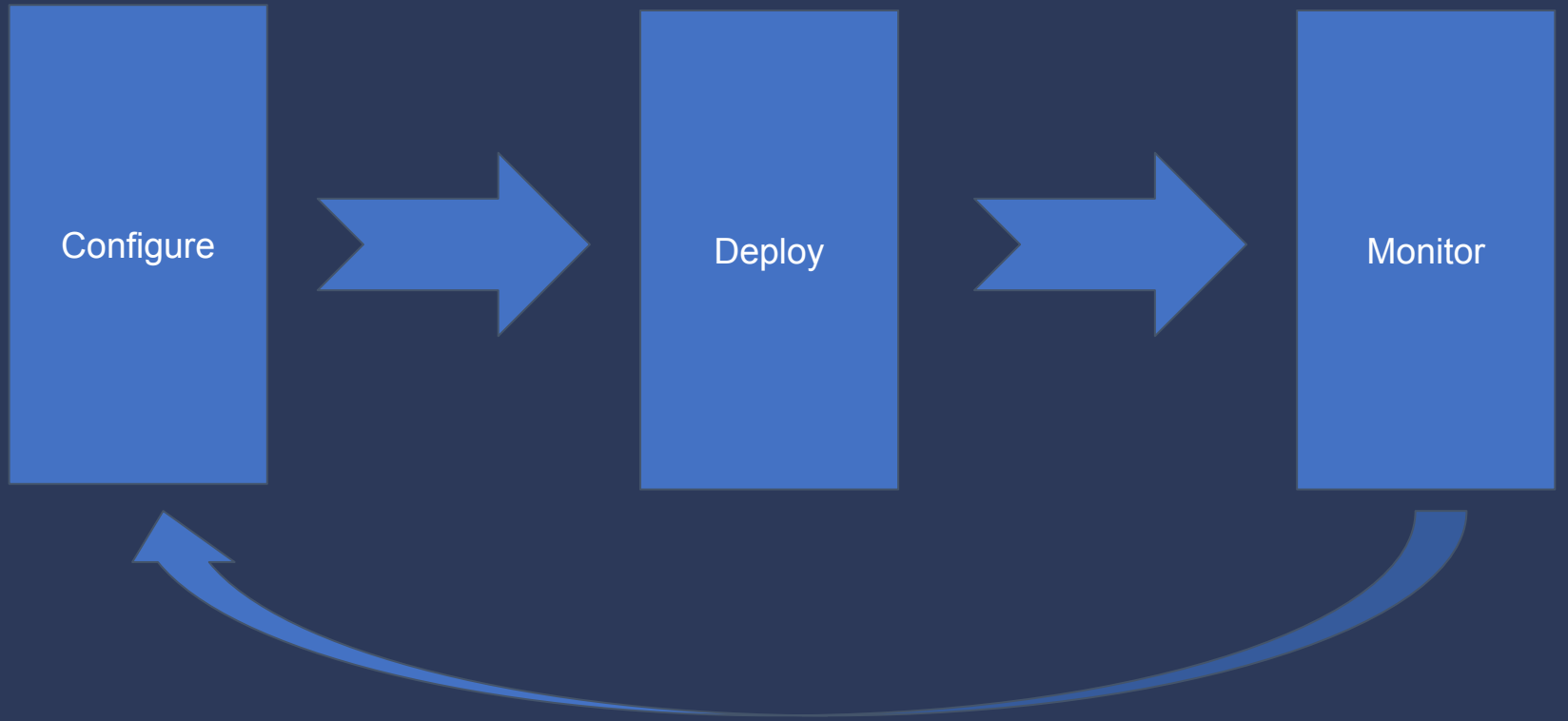
Throughput - how much of the traffic is helping answer questions?

Visibility - can I know what happened with each experiment?

Reliability - how certain I am that results are correct?

Serving for Engineers

Production Loop



Deploying Deep Learning Models

- Single model resource usage is complex
  - Changes with geography and time of day
  - Depends on architecture properties
  - Changes with inputs
  - Mostly not linear
- Multiple experiments require complex deployments
  - Could be coordinated
  - Could be periodic
  - Distinguish between default / experiment / debug models

Scale for production systems

Latency - how quickly can I get an answer?

Throughput - how many requests I can serve on given hardware?

Reliability - how likely I am to get an answer?

Visibility - what is running? where? With what performance?

Having it all

Data Science

Most traffic used for experiments

Flexibility in designing these experiments

Ability to look at experiment results

Reliability

Engineering

As much traffic as hardware allows

Responses return within latency budget

Ability to monitor running services

Reliability

Data Science

Most traffic used for experiments

Flexibility in designing these experiments

Ability to look at experiment results

Reliability

Engineering

As much traffic as hardware allows

Responses return within latency budget

Ability to monitor running services

Reliability

Need to prioritize

Data Science

Most traffic used for experiments

Flexibility in designing these experiments

Ability to look at experiment results

Reliability

Engineering

As much traffic as hardware allows

Responses return within latency budget

Ability to monitor running services

Reliability

Need to coordinate

The “variant” concept

Configuration for Training

Configuration for Deployment

Service name for Routing

ID for Monitoring/Reporting

Our Example variant

Recommends the same items every hour! Maybe we need to add features!

Latency is >20ms (too high). We need more cores!

Does better than the default model on weekends!

```
name: time_only_fully_connected
num_layers: 32
dropout_rate: 0.8
input_features:
  - "time_of_day"
  - "title"
min_cores: 25
memory_per_server: 2GB
```

But how?

Designing a serving solution

Micro service

Stateless

Horizontally scalable

No external dependencies

Compute only

Black box

Hard to validate outputs

No runtime config

No internal state to monitor

Complicated resource footprint

Train -> Deploy Pipeline

Train -> Deploy Pipeline

Training code (over Tensorflow) takes configuration and trains the model

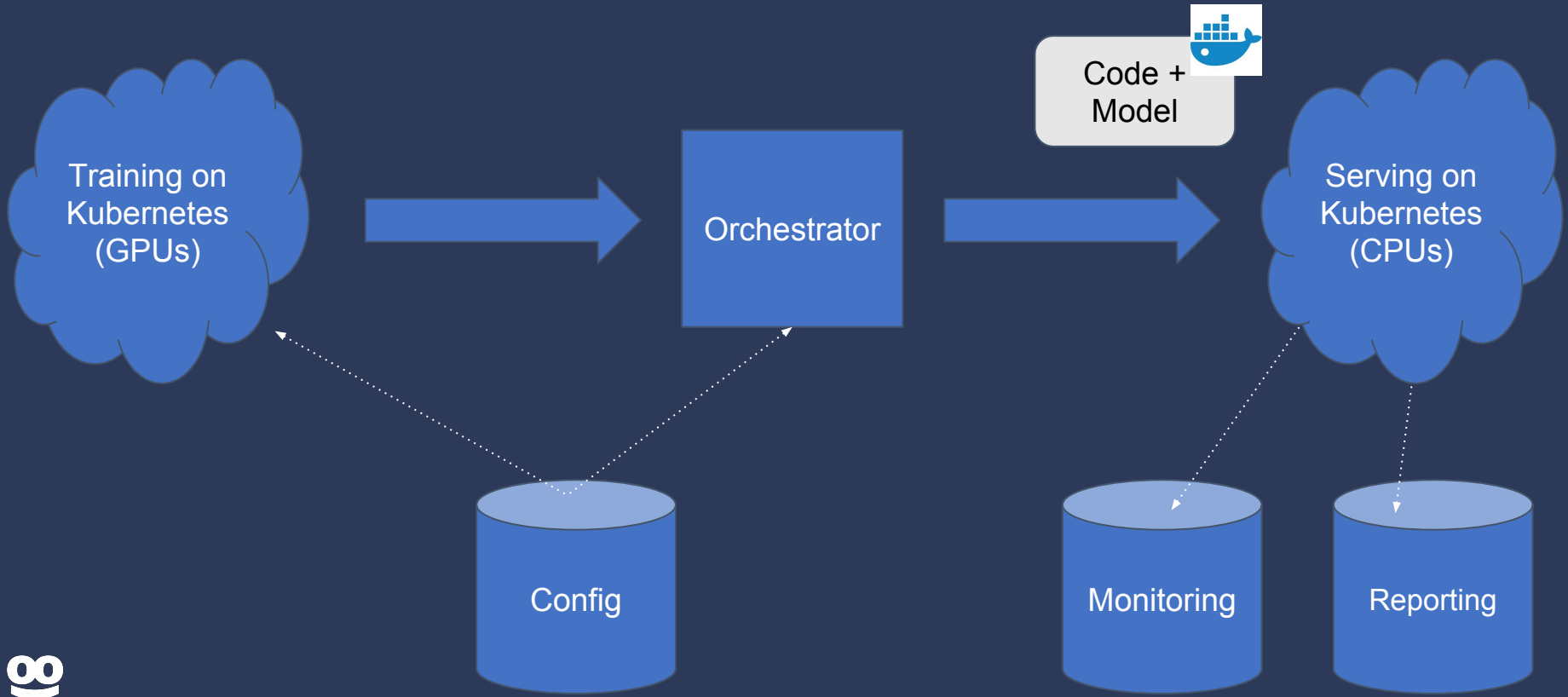
Train -> Deploy Pipeline

Model is packaged into a docker running Tensorflow Serving

Train -> Deploy Pipeline

A Kubernetes service runs as many dockers as needed

Components


Training on Kubernetes (GPUs)
Orchestrator
Code + Model
Serving on Kubernetes (CPUs)
Config
Monitoring
Reporting

Nuts and Bolts

implementation

Train -> Deploy Pipeline

Training code takes configuration and trains the model

Building Graphs

- Use Tensorflow
  - Compose pre-made or custom `tf.Layer`
  - Stock or custom `tf.Estimator`
  - Prefer high level reusable components
- Use external configuration
- Save as `SavedModel`

Solid configuration model

- Descriptive
  - What is in the model?
  - Who put it there?
  - What models should go together?
- Hierarchical
  - Often we only want to test a small thing
  - Often makes sense to compare to “default”
- Thorough
  - Shouldn’t be necessary to look at code at given point
- Points to the code

Spring cloud config

- Uses git as backend
- Has REST interface (for Python!)
- Flexible
  - Multiple profiles
  - Inheritance
  - Dynamic profile loading
  - Work from a branch
- Reliable

```
deploy:
  k8s:
    time_only_fully_connected:
      spec.minReplicas: 2
      replicaCount: 5
      spec.maxReplicas: 32
      resources.limits.cpu: 4
      resources.limits.memory: 8Gi
      resources.requests.cpu: 4
      resources.requests.memory: 8Gi
```

```
schedule_plan:
  num_training_days: 7
scheduled:
  - input_type: "users_sampled_5_percent"
    variants:
      - "time_only_fully_connected"
      - "day_only_fully_connected"
```

Train -> Deploy Pipeline

Model is packaged into a docker running Tensorflow Serving

TensorFlow Serving

Serves TensorFlow models!

- Written in C++
- Uses gRPC for communications
  - A protobuf extension
  - Wire protocol is HTTP/2.0
  - Good load balancer support
- Efficient
  - Low overhead
  - Good threading model
  - Automatic batching

```
apt-get install tensorflow-model-server

tensorflow_model_server \
--port 9000 \
--model_name mnist \
--model_base_path=/tmp/mnist_model/
```

Train -> Deploy Pipeline

A Kubernetes service runs as many dockers as needed

Containerized deployment

- Dynamic scaling over variants
  - Let’s give these two 5% traffic
- Scaling over time
  - The Japanese language model should get more traffic now
- Monitor at different granularities
  - Individual variant (for optimizations)
  - Individual host/data centre
  - The entire system (for sizing)
- Integration tests

Why Kubernetes?

- Out-of-the-box scaling
  - Dynamic scaling
  - Service discovery
  - Load balancers (pluggable)
- Deployment and monitoring is easy
  - Blue/green deployments
  - Liveness and readiness tests
- Easy to use
  - Great UI with dynamic editing
  - IP address for every container

The Recipe

1. Train your model with TensorFlow
2. Put TF Serving + model in docker
3. Deploy on Kubernetes
4. Serve real traffic
5. Analyze online metrics

Most routine data science is done in #1 and #5 (and in Python)

But what about..

more tips

Managing Schema

The one predictable part of the service

- Which features?
- What are the shapes?
- Is padding used?
- What is the vocabulary?

Managing Schema

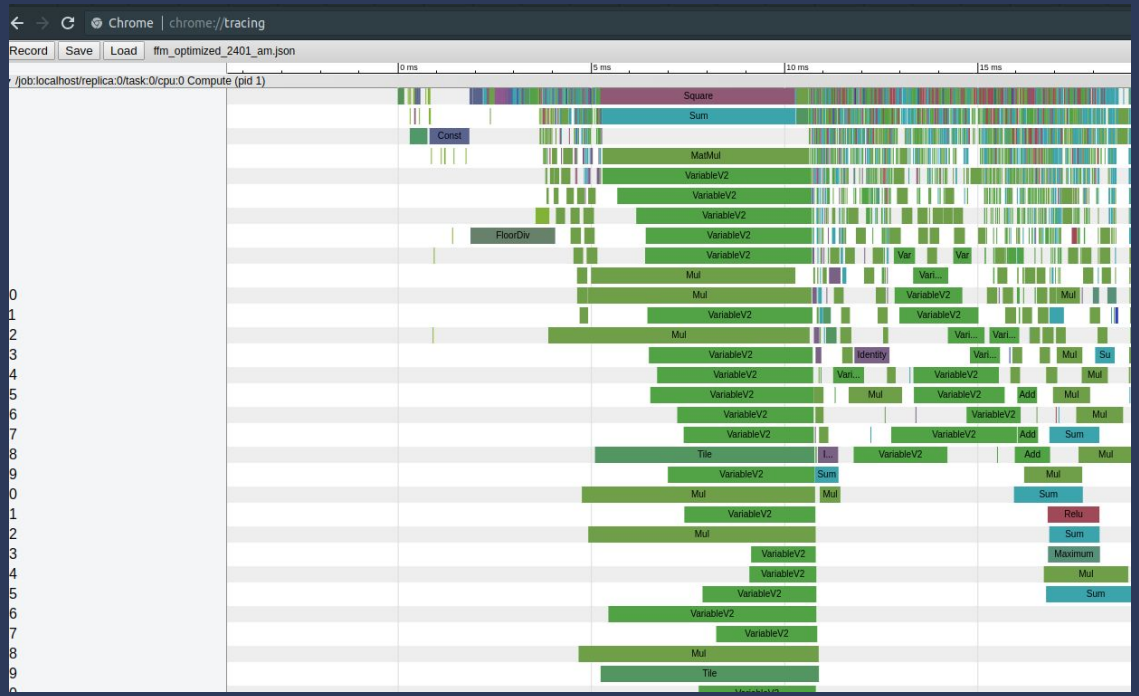
Only move forward!

- Features are added to all services
- All models “read” all features but only use some
- Deprecate and stop filling instead of removing
- Like protobuf

Tracing

Supported by Tensorflow Serving

Can be enabled *per request*



And does it work?

DEEP MODEL SERVING

Deep Learning in Taboola

150K+
Requests/sec

20ms
Typical response time

5
Data centers

40
Data Scientists
Using the platform

60
Models trained / day

4
Different algorithmic families