Machine Learning Systems Design

Deployment and Monitoring

Lecture 19: Model Serving

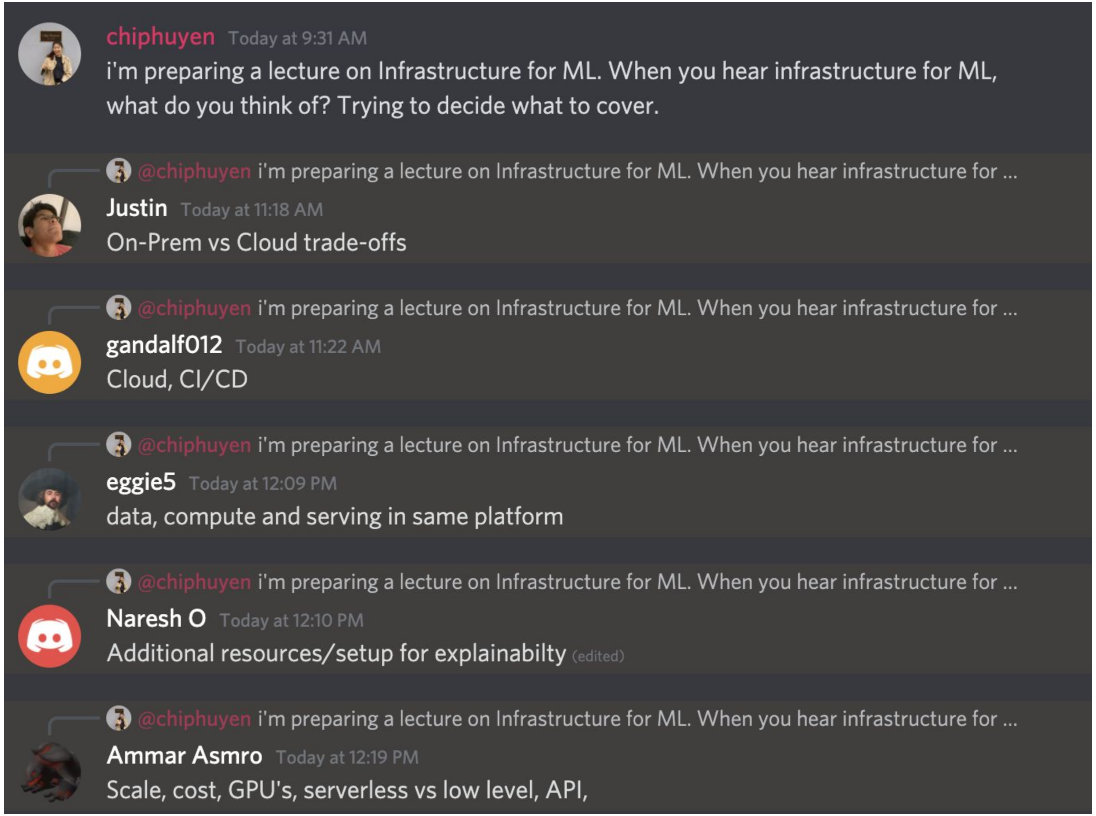
CE 40959 Spring 2023
Ali Zarezade
SharifMLSD.github.io

Agenda

1. Stream Serving
2. Batch vs Online Serving
3. Model Serving Considerations
4. **ML Infrastructure**
5. **Resource Management**
6. **ML Platforms**

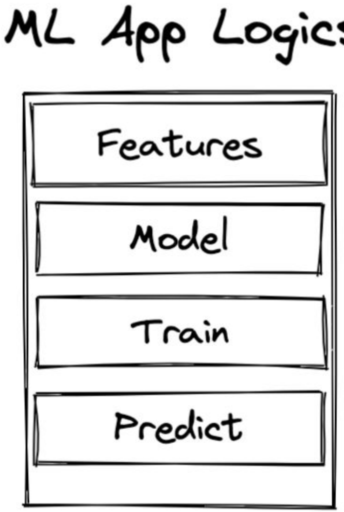
4. ML Infrastructure

What does infrastructure mean?



ML systems are complex

- More components
  - A request might jump 20-30 hops before response
  - A problem occurs, but where?

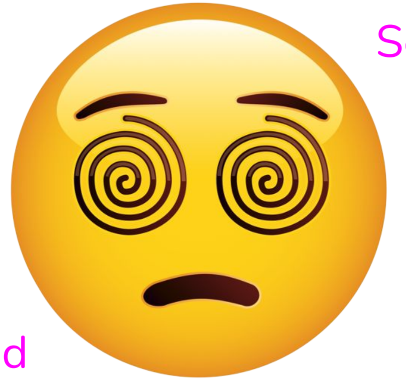


Containers

Schedulers

Serverless

Lambda functions

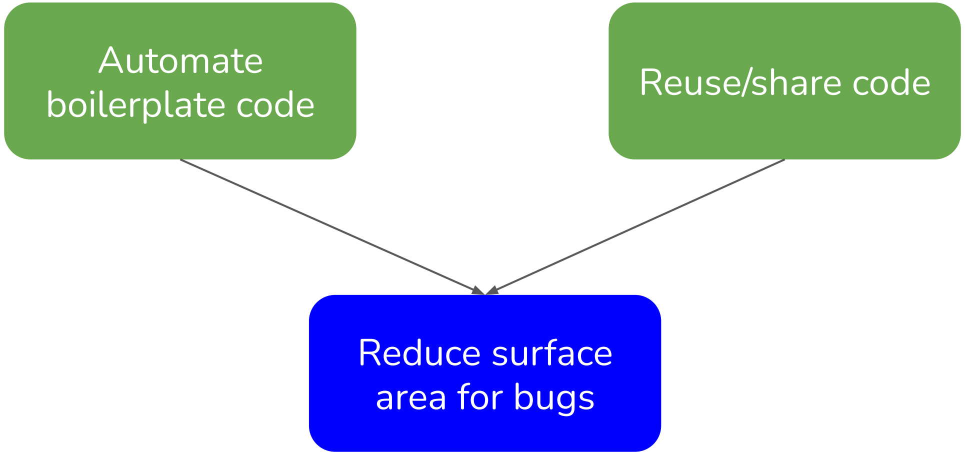


Mesh routing

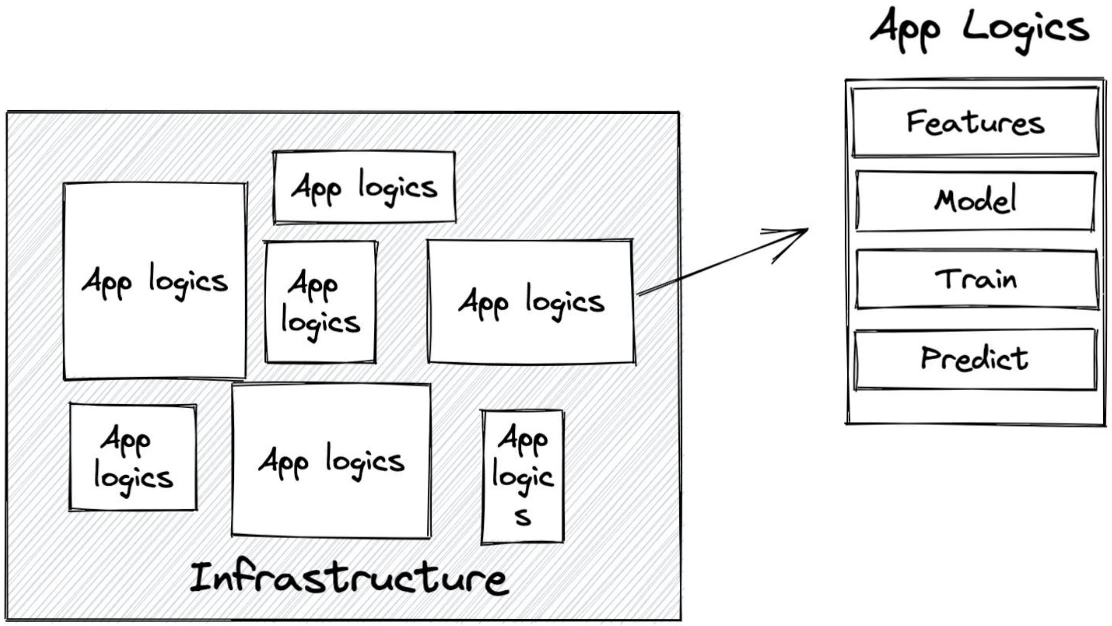
Load balancers

Microservices

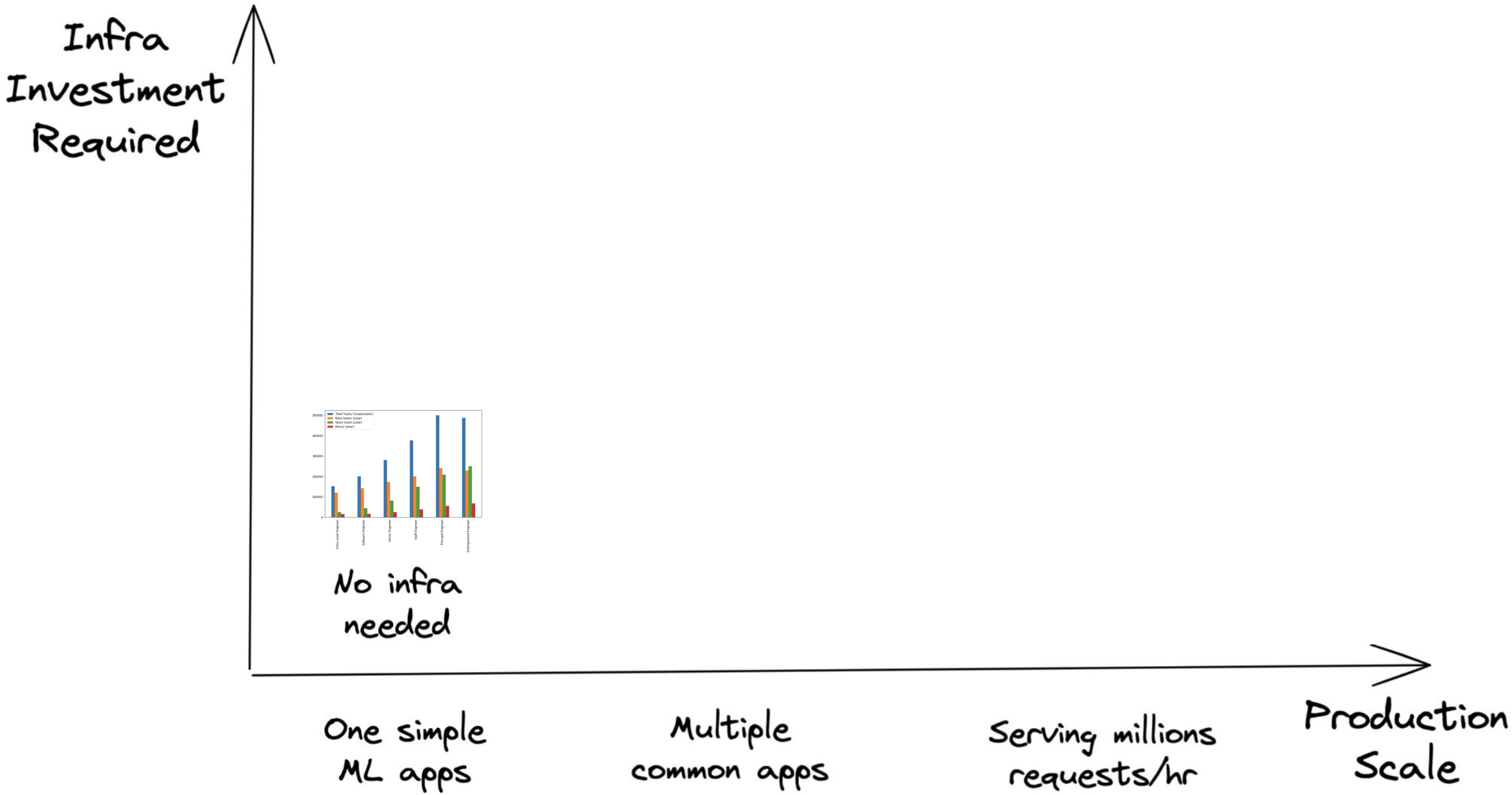
More complex systems, better infrastructure needed



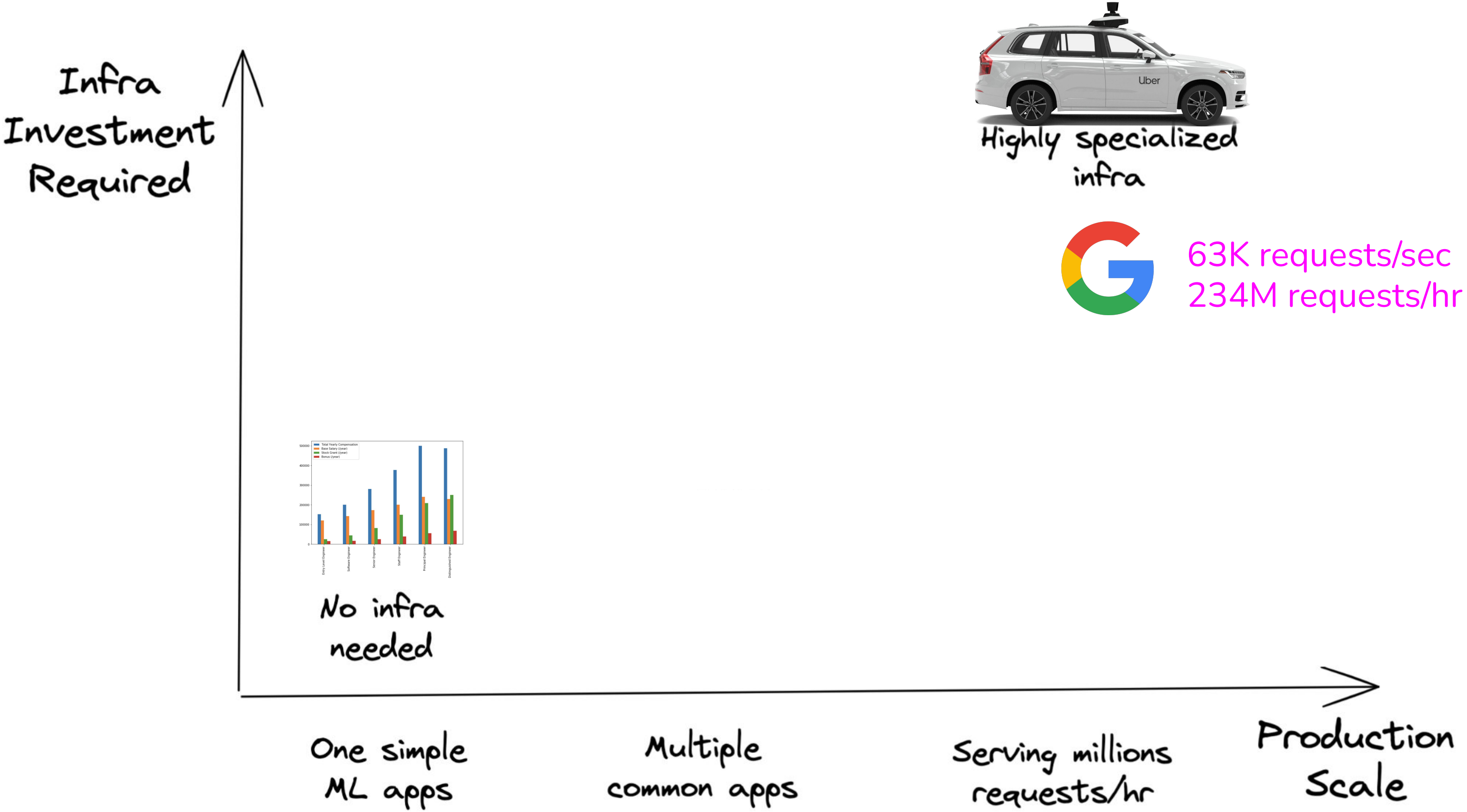
- Infrastructure: the set of fundamental facilities and systems that support the sustainable functionality of households and firms.
- ML infrastructure: the set of fundamental facilities that support the development and maintenance of ML systems.



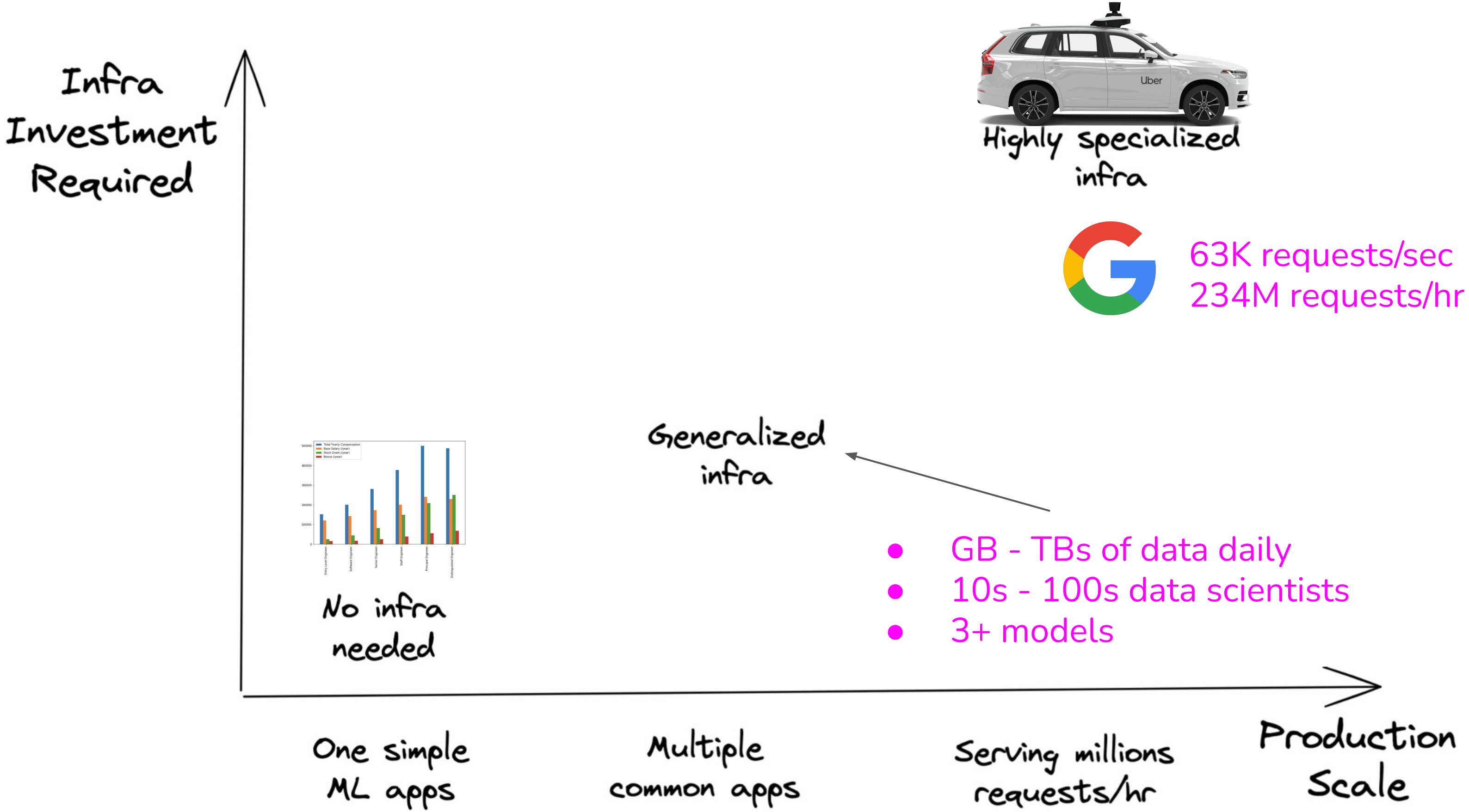
Every company’s infrastructure needs are different



Every company's infrastructure needs are different



Every company's infrastructure needs are different



Every company's infrastructure needs are different

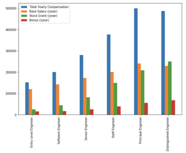
Infra Investment Required

Highly specialized infra

63K requests/sec
234M requests/hr

Generalized infra

Vast majority of apps
(reasonable scale)



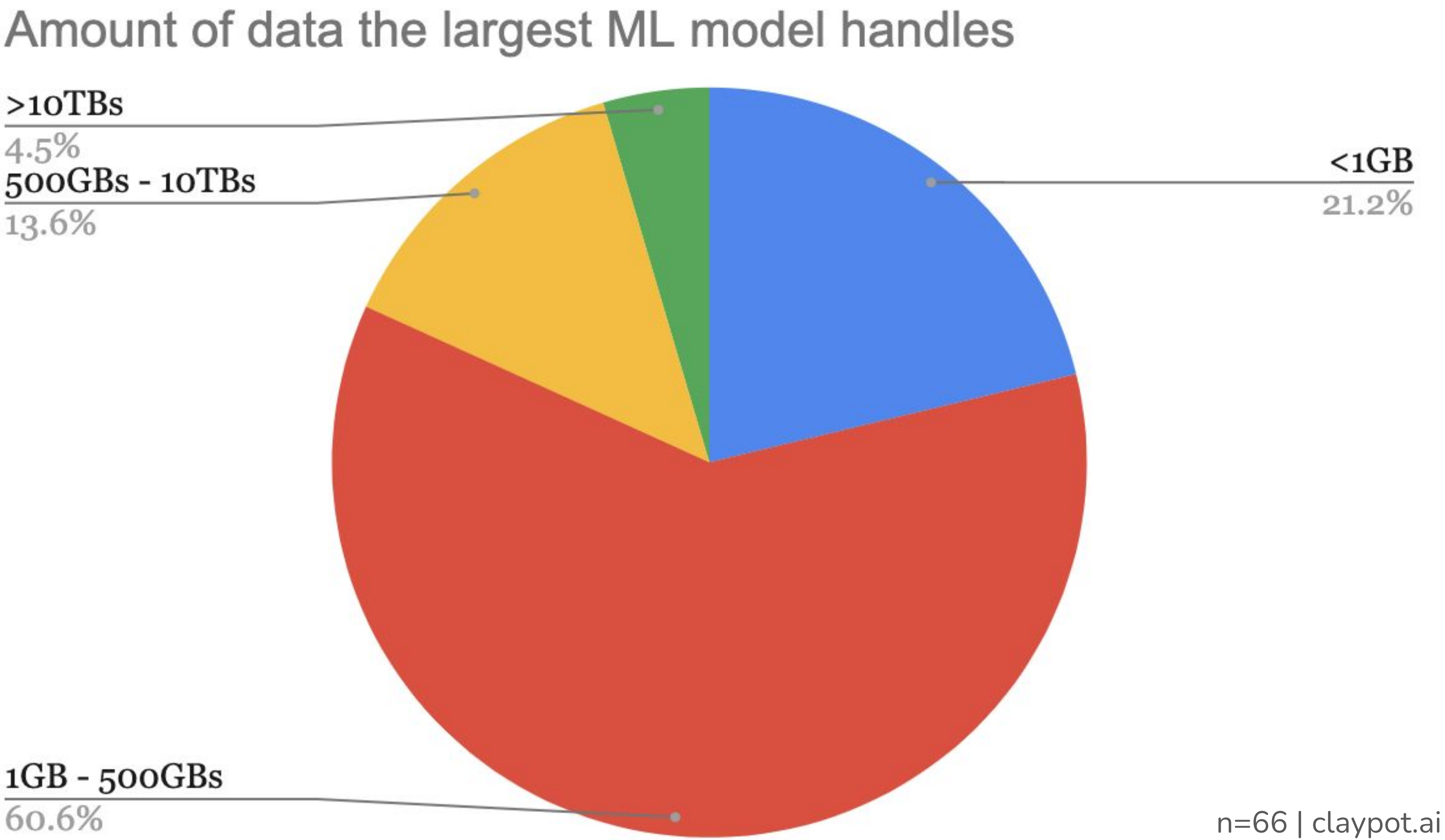
No infra needed

One simple ML apps

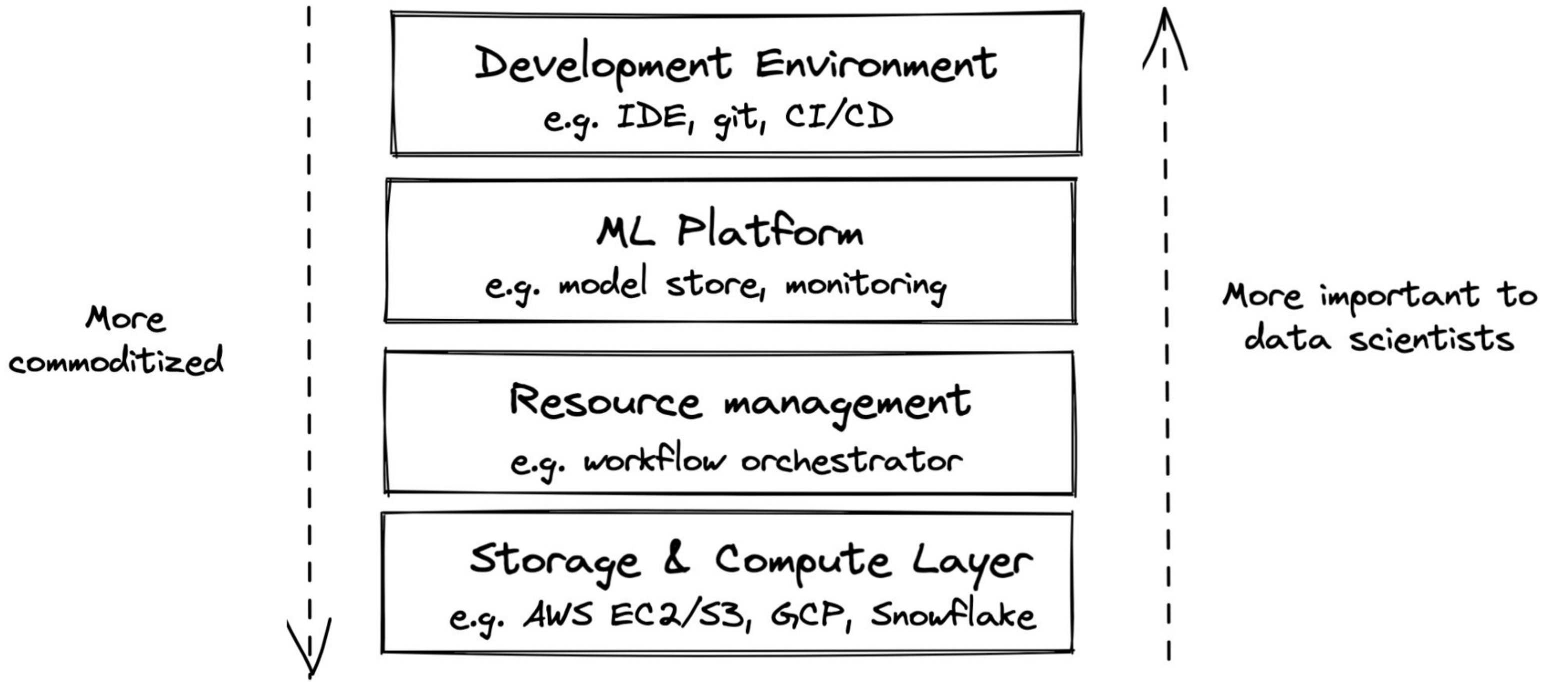
Multiple common apps

Serving millions requests/hr

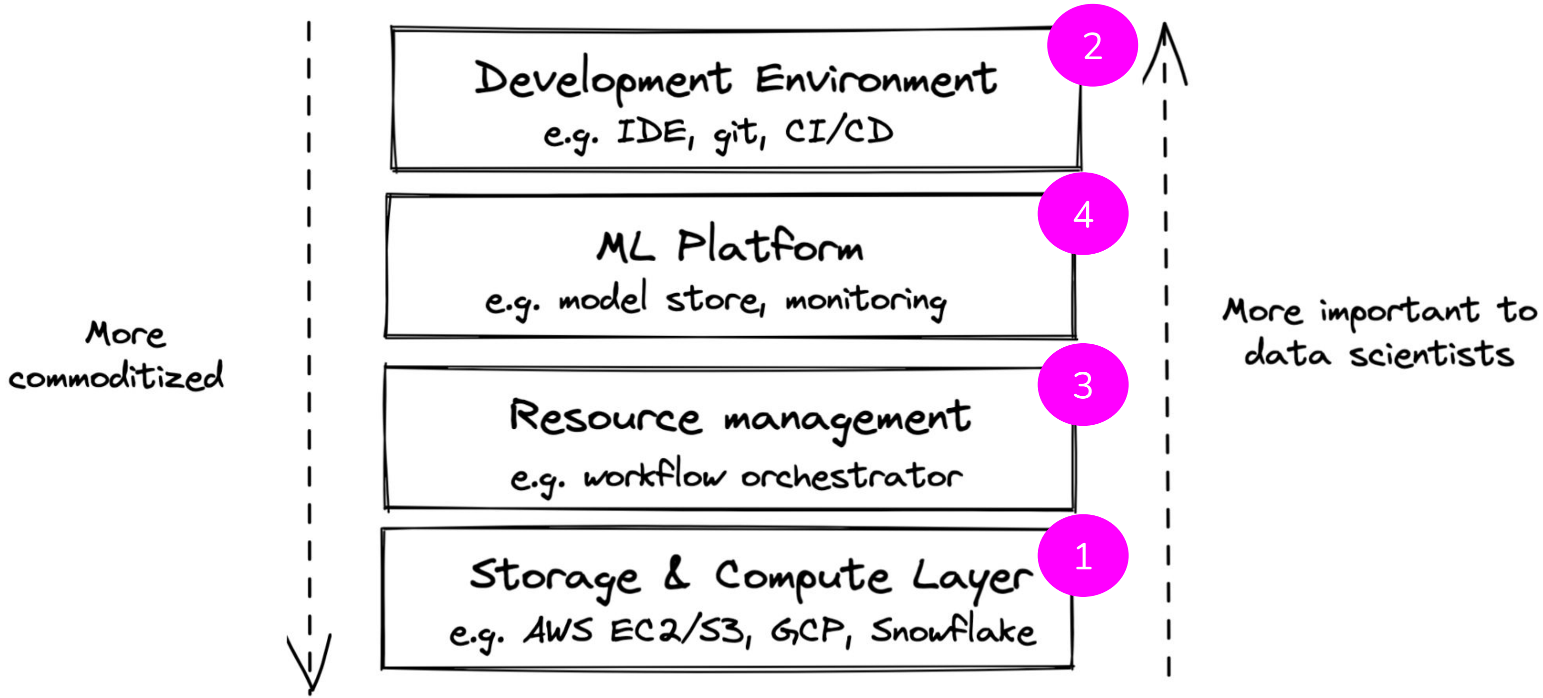
Production Scale


Amount of data the largest ML model handles
>10TBs
4.5%
500GBs - 10TBs
13.6%
<1GB
21.2%
1GB - 500GBs
60.6%
n=66 | claypot.ai

Infrastructure Layers



Infrastructure Layers



Storage

- Where data is collected and stored
- Simplest form: HDD, SSD
- More complex forms: data lake, data warehouse
- Examples: S3, Redshift, Snowflake, BigQuery

See Data Lecture slides

Storage: heavily commoditized

- Most companies use storage provided by other companies (e.g. cloud)
- Storage has become so cheap that most companies just store everything

Compute layer: engine to execute your jobs

- Compute resources a company has access to
- Mechanism to determine how these resources can be used

Compute layer: engine to execute jobs

- Simplest form: a single CPU/GPU core
- Most common form: cloud compute

Compute unit

- Compute layer can be sliced into smaller compute units **to be used concurrently**
    - A CPU core might support 2 concurrent threads, **each thread** is used as a compute unit to execute its own **job**
    - Multiple CPUs can be joined to form a **large compute unit** to execute a large **job**

Compute unit

- Compute layer can be sliced into smaller compute units **to be used concurrently**
  - A CPU core might support 2 concurrent threads, **each thread** is used as a compute unit to execute its own **job**
  - Multiple CPUs can be joined to form a **large compute unit** to execute a large **job**

Unit: job

DASK

APACHE Spark

RAY

Unit: pod

Wrapper around container

kubernetes

Compute layer: how to execute jobs

1. Load data into memory
2. Perform operations on that data
   a. Operations: add, subtract, multiply, convolution, etc.

To add arrays A and B

1. Load A & B into memory
2. Perform addition on A and B

Compute layer: how to execute jobs

1. Load data into memory
2. Perform operations on that data
   a. Operations: add, subtract, multiply, convolution, etc.

If A & B don't fit into memory, it'll be possible to do the ops without out-of-memory algorithms

To add arrays A and B
1. Load A & B into memory
2. Perform addition on A and B

Compute layer: how to execute jobs

1. Load data into memory
2. Perform operations on that data
    a. Operations: add, subtract, multiply, convolution, etc.

To add arrays A and B
1. Load A & B into memory
2. Perform addition on A and B

Important metrics of compute layer:
1. Memory
2. Speed of computing ops

Compute layer: memory

- Amount of memory
  - Straightforward
  - An instance with 8GB of memory is more expensive than an instance with 2GB of memory

Compute layer: memory

- Amount of memory
- I/O bandwidth: speed at which data can be loaded into memory

Compute layer: speed of ops

- Most common metric: FLOPS
    - **F**loating **P**oint **O**perations **P**er **S**econd

“A Cloud TPU v2 can perform up to 180 teraflops, and the TPU v3 up to 420 teraflops.”

\- Google, 2021

Compute layer: speed of ops

- Most common metric: FLOPS
- Contentious
  - What exactly is an ops?
    - If 2 ops are fused together, is it 1 or 2 ops?
  - Peak perf at 1 teraFLOPS doesn't mean your app will run at 1 teraFLOPS

Compute layer: utilization

- Utilization = actual FLOPS / peak FLOPS

If peak 1 trillion FLOPS but job runs 300
billion FLOPS
-> utilization = 0.3

Compute layer: utilization

- Utilization = actual FLOPS / peak FLOPS
- Dependent on how fast data can be loaded into memory

The higher, the better

Tensor Cores are very fast. So fast … that they are idle most of the time as ***they are waiting for memory to arrive from global memory.***

For example, during BERT Large training, which uses huge matrices — the larger, the better for Tensor Cores — ***we have utilization of about 30%****.*

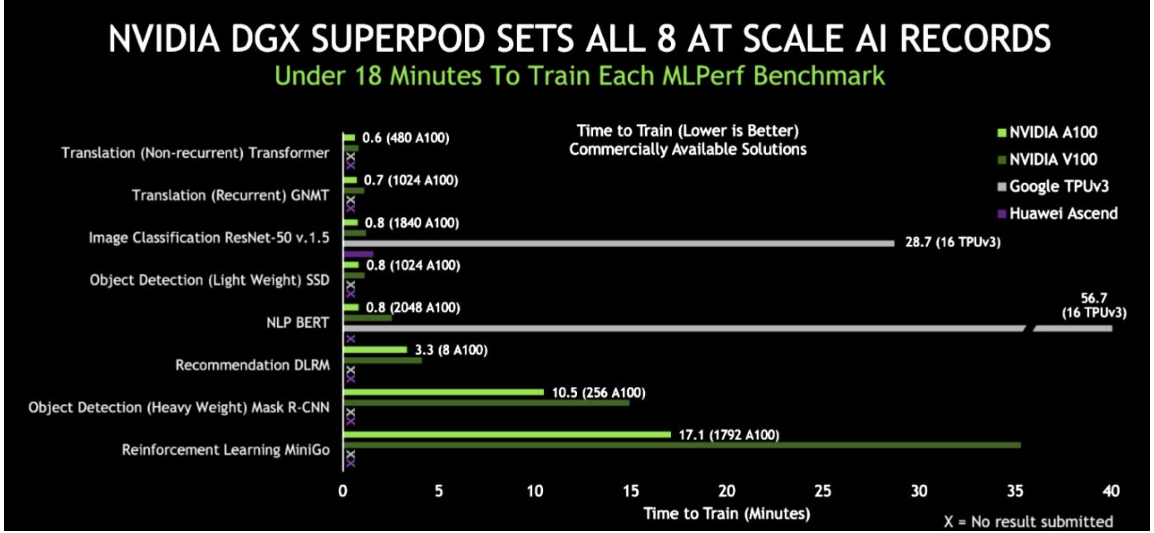
- Tim Dettmers, 2020

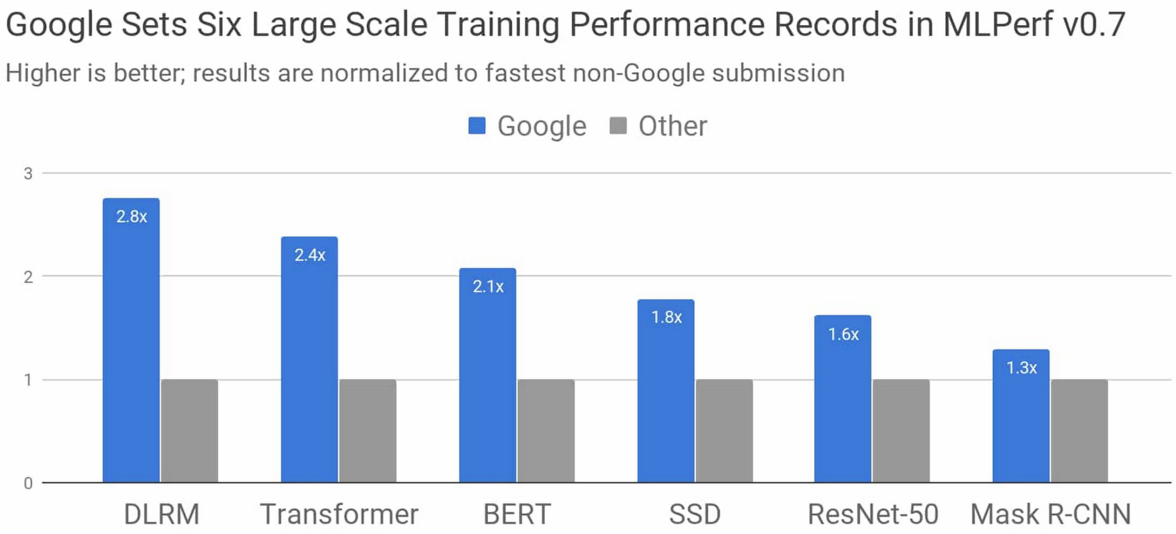
Compute layer: if not FLOPS, then what?

Compute layer: if not FLOPS, then what?

- How long it will take this compute unit to do common workloads
- MLPerf measure hardware on common ML tasks e.g.
  - Train a ResNet-50 model on the ImageNet dataset
  - Use a BERT-large model to generate predictions for the SQuAD dataset

MLPerf is also contentious





Compute layer: evaluation

- Memory
- Cores
- I/O bandwidth
- Cost

Some GPU instances on AWS

Instance	GPUs	vCPU	Mem (GiB)	GPU Mem (GiB)
p3.2xlarge	1	8	61	16
p3.8xlarge	4	32	244	64
p3.16xlarge	8	64	488	128
p3dn.24xlarge	8	96	768	256

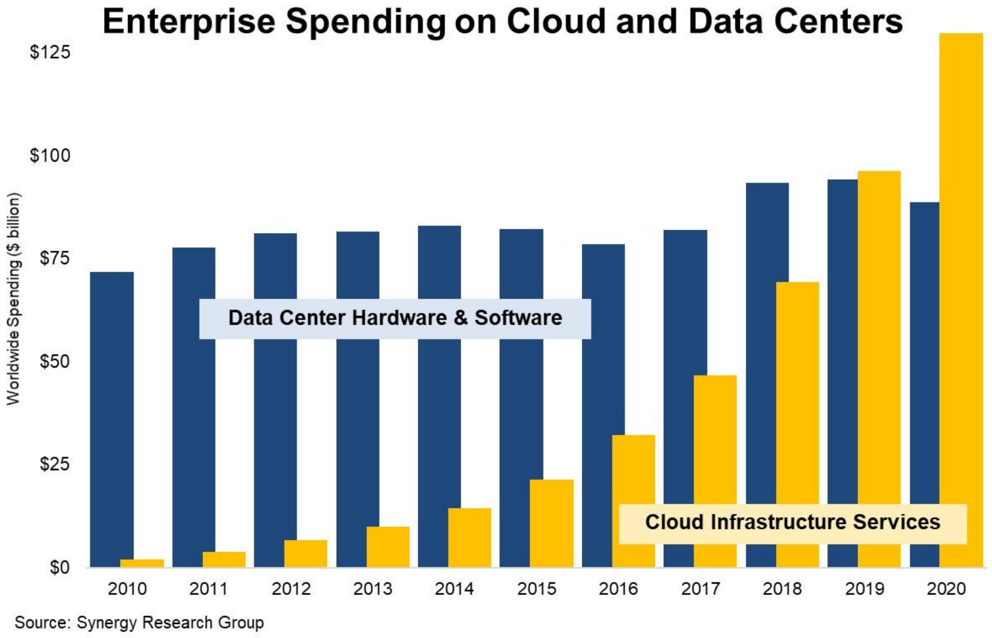
Some TPU instances on GCP

TPU type (v2)	v2 cores	Total memory
v2-8	8	64 GiB

TPU type (v3)	v3 cores	Total memory
v3-8	8	128 GiB

Public Cloud vs. Private Data Centers

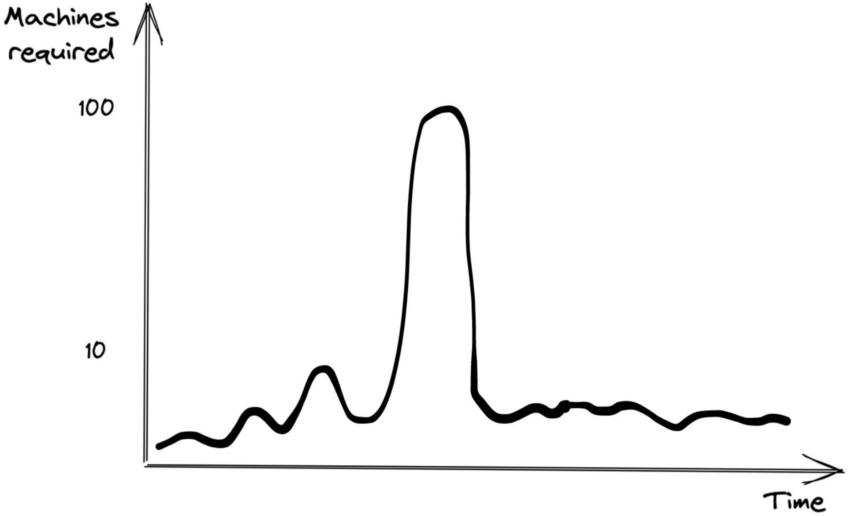
- Like storage, compute is largely commoditized



[2020 – The Year that Cloud Service Revenues Finally Dwarfed Enterprise Spending on Data Centers | Synergy Research Group]

Benefits of cloud

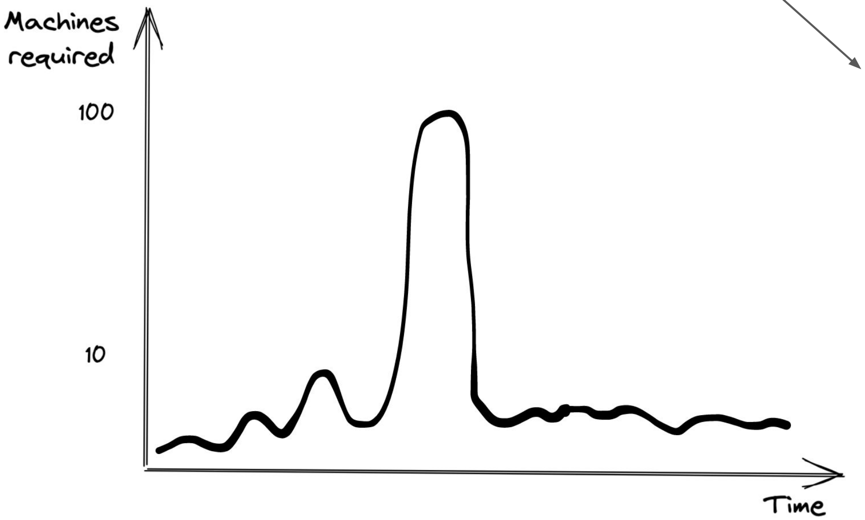
- Easy to get started
- Appealing to variable-sized workloads
  - Private: would need 100 machines upfront, most will be idle most of the time
  - Cloud: pay for 100 machines only when needed



Benefits of cloud

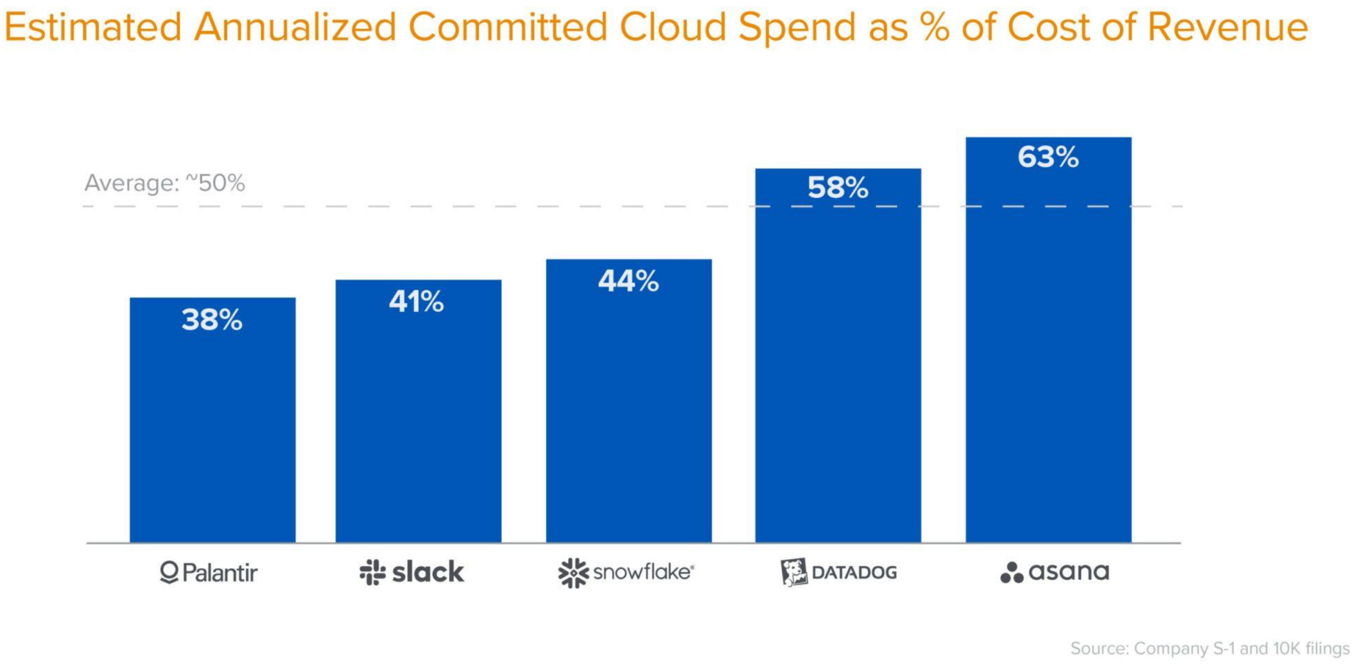
- Easy to get started
- Appealing to variable-sized workloads
  - Private: would need 100 machines upfront, most will be idle most of the time
  - Cloud: pay for 100 machines only when needed

Autoscaling!



Drawbacks of cloud: cost

- Cloud spending: ~50% cost of revenue



[The Cost of Cloud, a Trillion Dollar Paradox | Andreessen Horowitz](#) (2021)

Drawbacks of cloud: cost

*"Across 50 of the top public software companies currently utilizing cloud infrastructure, an **estimated $100B of market value is being lost … due to cloud impact on margins** — relative to running the infrastructure themselves."*

The Cost of Cloud, a Trillion Dollar Paradox | Andreessen Horowitz (2021)

Cloud repatriation

- Process of moving workloads from cloud to private data centers

Dropbox Infrastructure Optimization Initiative Impact

Dropbox Historical Financials

	2015	2016	2017
Revenue	**$604**	**$845**	**$1,107**
Annual Growth Rate		*40%*	*31%*
Infrastructure Optimization Cumulative Net Savings	N/A	40	75
Cost of Revenue	407	391	369
Gross Profit	**$196**	**$454**	**$738**
Gross Margin	*33%*	*54%*	*67%*
Free Cash Flow	**($64)**	**$137**	**$305**
Incremental Margin vs. 2015 (% Pt)		**+21%**	**+34%**

Source: Dropbox S-1, a16z analysis

A large chunk due to cloud repatriation

Multicloud strategy

- To optimize cost
- To avoid cloud vendor lock-in

“81% of respondents said they are working with two or more providers”

- Gartner (2019)

5. Resource Management

Resource management

	Pre-cloud	Cloud
Resources	Finite	Practically infinite
Implication	More resources for an app = less resources for other apps	More resources for an app don't have to affect other apps
Goal	Utilization	Utilization + cost efficiency

Resource management

	Pre-cloud	Cloud
Resources	Finite	Practically infinite
Implication	More resources for an app = less resources for other apps	More resources for an app don't have to affect other apps
Goal	Utilization	Utilization + cost efficiency

Simplify the allocation challenge

Resource management

	Pre-cloud	Cloud
Resources	Finite	Practically infinite
Implication	More resources for an app = less resources for other apps	More resources for an app don't have to affect other apps
Goal	Utilization	Utilization + cost efficiency

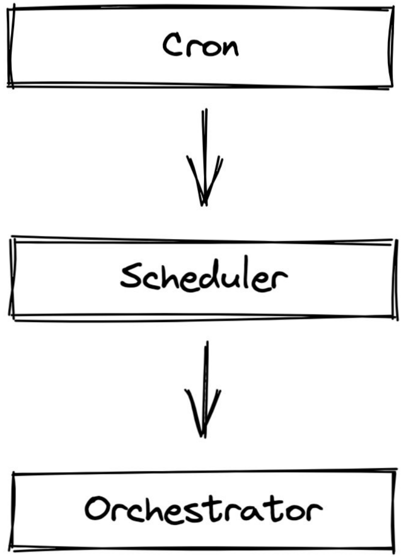
OK to use more resources if help engineers to be more productive

ML workloads

- Repetitive
  - Batch prediction
  - Periodical retraining
  - Periodical analytics
- Dependencies
  - E.g. train depends on featurize

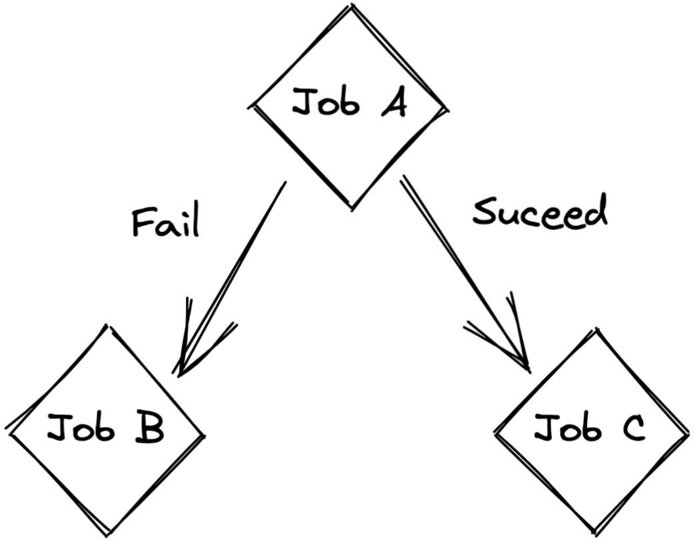
ML workloads

- Repetitive
  - Batch prediction
  - Periodical retraining
  - Periodical analytics
- Dependencies
  - E.g. train depends on featurize



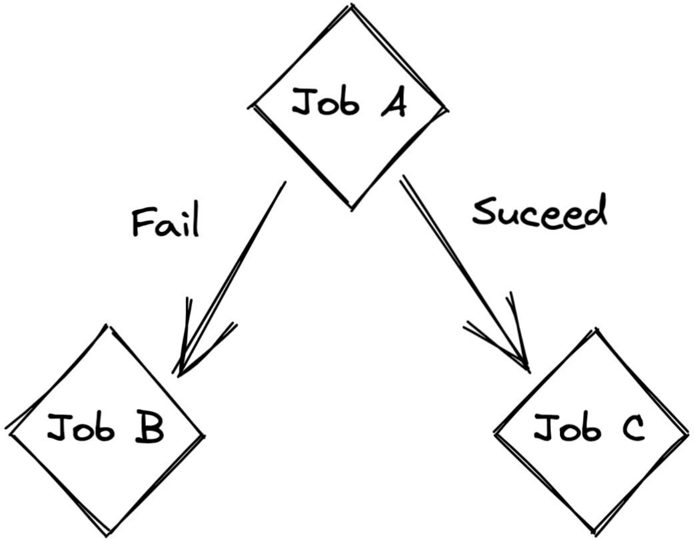
Cron: extremely simple

- Schedule jobs to run at fixed time intervals
- Report the results



Cron: extremely simple

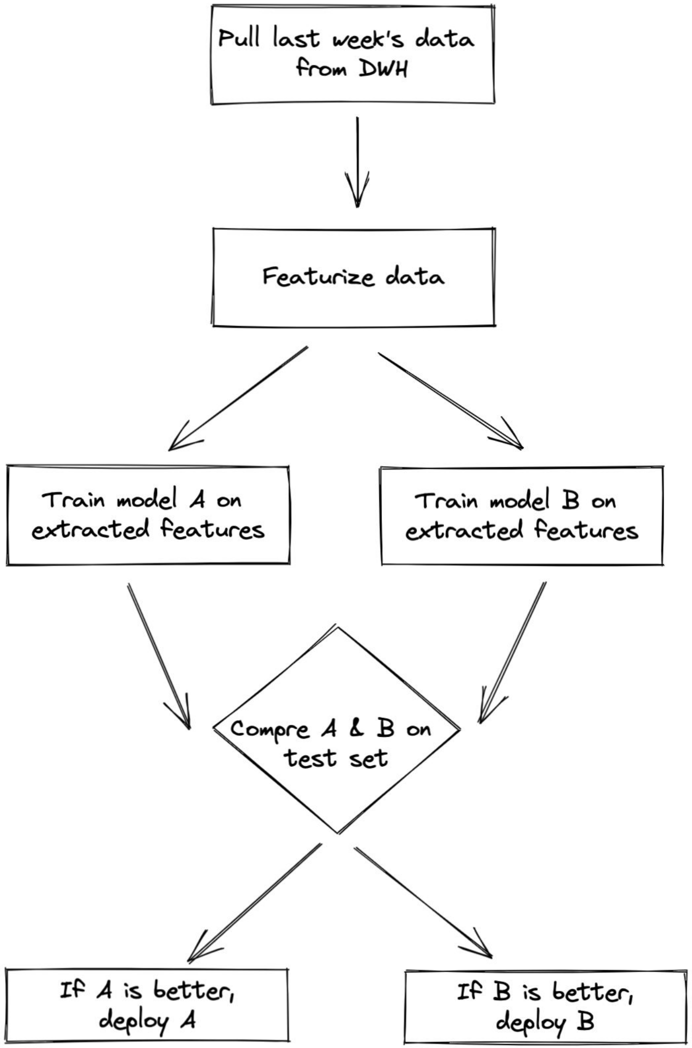
- Schedule jobs to run at fixed time intervals
- Report the results



Cron can't handle this

Scheduler

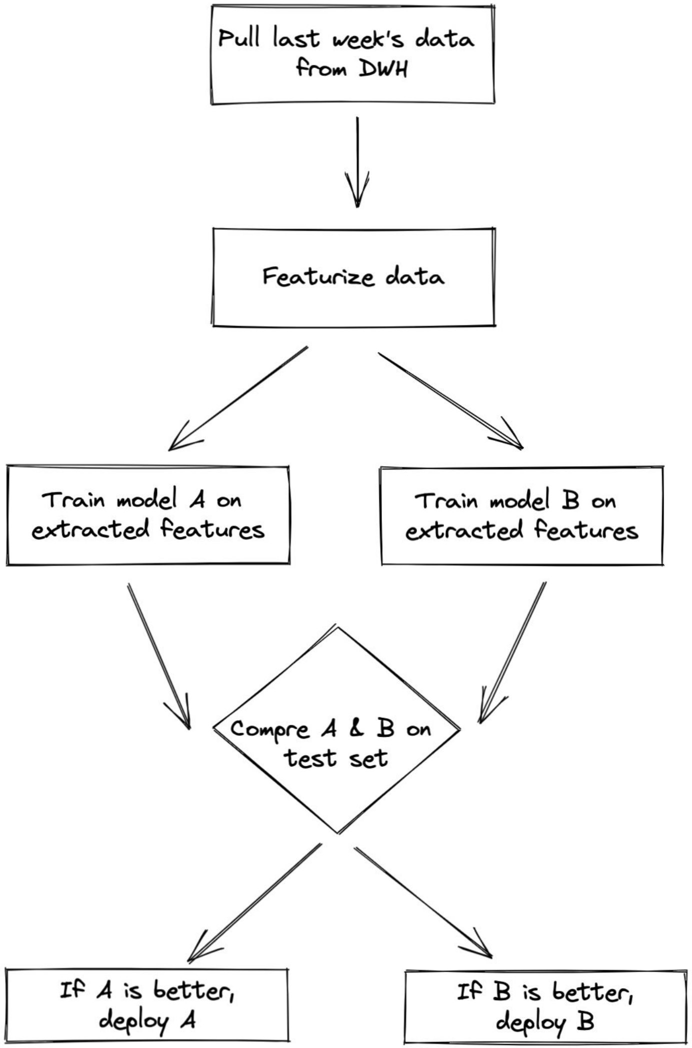
- Schedulers are cron programs that can handle dependencies



Scheduler

- Most schedulers require you to specify your workloads as DAGs

This is a DAG
- Directed
- Acyclic
- Graph



Scheduler

- Can handle event-based & time based triggers
  - Run job A whenever X happens
- If a job fails, specify how many times to retry before giving up
- Jobs can be queued, prioritized, and allocated resources
  - If a job requires 8GB of memory and 2 CPUs, scheduler needs to find an instance with 8GB of memory and 2 CPUs

Scheduler: SLURM example

```
#!/bin/bash
#SBATCH -J JobName
#SBATCH --time=11:00:00        # When to start the job
#SBATCH --mem-per-cpu=4096     # Memory, in MB, to be allocated per CPU
#SBATCH --cpus-per-task=4      # Number of cores per task
```

Scheduler: optimize utilization

- Schedulers aware of:
    - resources available
    - resources needed for each job
- Sophisticated schedulers (e.g. Google Borg) can reclaim unused resources
    - If I estimate that my job needs 8GB and it only uses 4GB, reclaim 4GB for other jobs

Scheduler challenge

- General purpose schedulers are extremely hard to design
- Need to handle any workload with any number of concurrent machines
- If scheduler is down, every workflow this scheduler touches will also be down

Scheduler to Orchestrator

- Scheduler: **when** to run jobs
- Orchestrator: **where** to run jobs

Scheduler to Orchestrator

- Scheduler: **when** to run jobs
    - Handle jobs, queues, user-level quotas, etc.
- Orchestrator: **where** to run jobs
    - Handle containers, instances, clusters, replication, etc.
    - Provision: allocate more instances to the instance pool as needed

Scheduler to Orchestrator

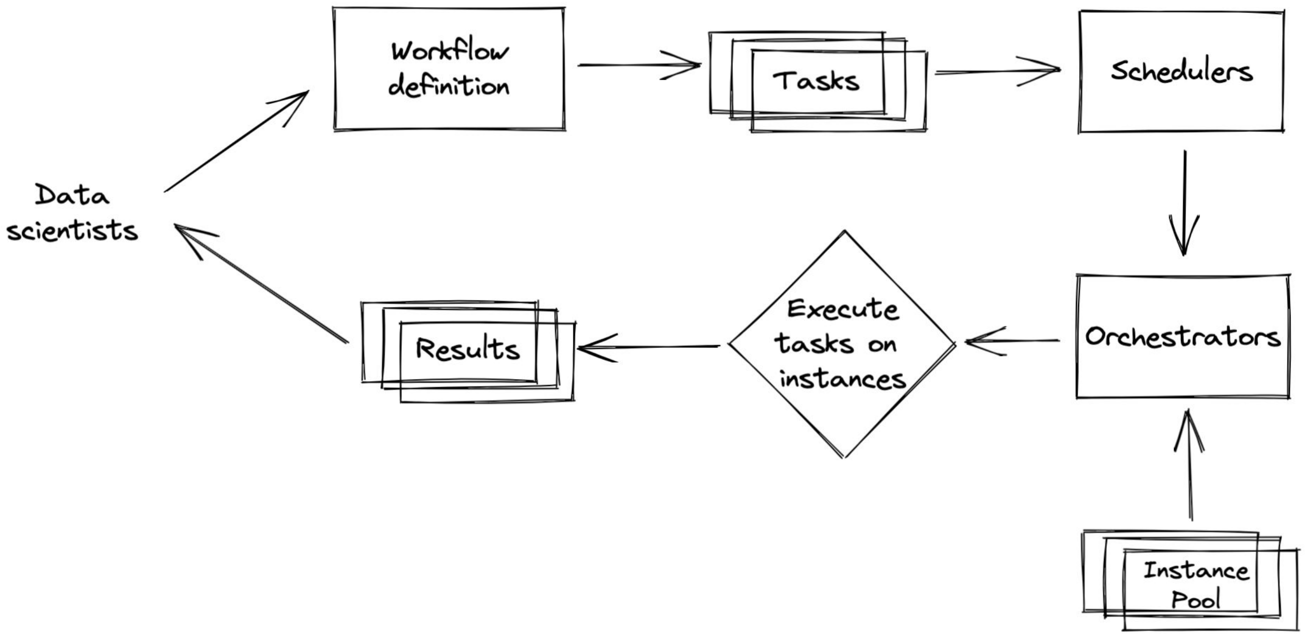
- Scheduler: **when** to run jobs
  - Handle jobs, queues, user-level quotas, etc.
  - Typically used for periodical jobs like batch jobs
- Orchestrator: **where** to run jobs
  - Handle containers, instances, clusters, replication, etc.
  - Provision: allocate more instances to the instance pool as needed
  - Typically used for long-running jobs like services

kubernetes

Scheduler & orchestrator

- Schedulers usually have some orchestrating capacity and vice versa
  - Schedulers like SLURM and Google's Borg have some orchestrating capacity
  - Orchestrators like HashiCorp Nomad and K8s come with some scheduling capacity
- Often, schedulers are run on top of orchestrators
  - Run Spark's job scheduler on top of K8s
  - Run AWS Batch scheduler on top of EKS

Data science workflow management



Data science workflow

- Can be defined using:
  - Code (Python)
  - Configuration files (YAML)
- Examples: Airflow, Argo, KubeFlow, Metaflow

Airflow

- 1st gen data science workflow management
- Champion of "configuration-as-code"
- Wide range of operators to expand capabilities

```
dag = DAG(
    'docker_sample',
    default_args={
        'owner': 'airflow',
        'depends_on_past': False,
        'email': ['airflow@example.com'],
        'email_on_failure': False,
        'email_on_retry': False,
        'retries': 1,
        'retry_delay': timedelta(minutes=5),
    },
    schedule_interval=timedelta(minutes=10),
    start_date=days_ago(2),
)

t1 = BashOperator(task_id='print_date', bash_command='date', dag=dag)

t2 = BashOperator(task_id='sleep', bash_command='sleep 5', retries=3, dag=dag)

t3 = DockerOperator(
    api_version='1.19',
    docker_url='tcp://localhost:2375',  # Set your docker URL
    command='/bin/sleep 30',
    image='centos:latest',
    network_mode='bridge',
    task_id='docker_op_tester',
    dag=dag,
)


t4 = BashOperator(task_id='print_hello', bash_command='echo "hello world!!!"', dag=dag)


t1 >> t2
t1 >> t3
t3 >> t4
```

Airflow: cons

- Monolithic
  - The entire workflow as a container
- Non-parameterized
  - E.g. need to define another workflow if you want to change learning rate
- Static DAG
  - Can't handle workloads with unknown number of records

```python
dag = DAG(
    'docker_sample',
    default_args={
        'owner': 'airflow',
        'depends_on_past': False,
        'email': ['airflow@example.com'],
        'email_on_failure': False,
        'email_on_retry': False,
        'retries': 1,
        'retry_delay': timedelta(minutes=5),
    },
    schedule_interval=timedelta(minutes=10),
    start_date=days_ago(2),
)

t1 = BashOperator(task_id='print_date', bash_command='date', dag=dag)

t2 = BashOperator(task_id='sleep', bash_command='sleep 5', retries=3, dag=dag)

t3 = DockerOperator(
    api_version='1.19',
    docker_url='tcp://localhost:2375',  # Set your docker URL
    command='/bin/sleep 30',
    image='centos:latest',
    network_mode='bridge',
    task_id='docker_op_tester',
    dag=dag,
)


t4 = BashOperator(task_id='print_hello', bash_command='echo "hello world!!!"', dag=dag)


t1 >> t2
t1 >> t3
t3 >> t4
```

Argo: next gen

- Created to address Airflow's problems
  - Containerized
  - Fully parameterized
  - Dynamic DAG

Argo: cons

- YAML-based configs
  - Can get very messy
- Only run on K8s clusters
  - Can’t easily test in dev environment

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: coinflip-
  annotations:
    workflows.argoproj.io/description: |
      This is an example of coin flip defined as a sequence of conditional steps.
      You can also run it in Python: https://couler-proj.github.io/couler/examples/#coin-flip
spec:
  entrypoint: coinflip
  templates:
  - name: coinflip
    steps:
    - - name: flip-coin
        template: flip-coin
    - - name: heads
        template: heads
        when: "{{steps.flip-coin.outputs.result}} == heads"
      - name: tails
        template: tails
        when: "{{steps.flip-coin.outputs.result}} == tails"

  - name: flip-coin
    script:
      image: python:alpine3.6
      command: [python]
      source: |
        import random
        result = "heads" if random.randint(0,1) == 0 else "tails"
        print(result)

  - name: heads
    container:
      image: alpine:3.6
      command: [sh, -c]
      args: ["echo \"it was heads\""]

  - name: tails
    container:
      image: alpine:3.6
      command: [sh, -c]
      args: ["echo \"it was tails\""]
```

Kubeflow & Metaflow: same code in dev & prod

- Allows data scientists to use the same code in both dev and prod environments

Kubeflow: more mature but more boilerplate

Dockerfile for the component train

```
ARG BASE_IMAGE_TAG=1.12.0-py3
FROM tensorflow/tensorflow:$BASE_IMAGE_TAG
RUN python3 -m pip install keras
COPY ./src /pipelines/component/src
```

Load specs of different components

```
preprocess_operation = kfp.components.load_component_from_url(
    'https://storage.googleapis.com/{}/components/preprocess/component.yaml'.format(BUCKET))
help(preprocess_operation)

train_operation = kfp.components.load_component_from_url(
    'https://storage.googleapis.com/{}/components/train/component.yaml'.format(BUCKET))
help(train_operation)

ai_platform_deploy_operation = comp.load_component_from_url(
    "https://storage.googleapis.com/{}/components/deploy/component.yaml".format(BUCKET))
help(ai_platform_deploy_operation)
```

Spec for the component train

```
name: train
description: Trains the NER Bi-LSTM.
inputs:
- {name: Input x URI, type: GCSPath}
- {name: Input y URI, type: GCSPath}
- {name: Input job dir URI, type: GCSPath}
- {name: Input tags, type: Integer}
- {name: Input words, type: Integer}
- {name: Input dropout }
- {name: Output model URI template, type: GCSPath}
outputs:
  - name: Output model URI
    type: GCSPath
implementation:
  container:
    image: gcr.io/<PROJECT-ID>/kubeflow/ner/train:latest
    command: [
      python3, /pipelines/component/src/train.py,
      --input-x-path,           {inputValue: Input x URI},
      --input-job-dir,          {inputValue: Input job dir URI},
      --input-y-path,           {inputValue: Input y URI},
      --input-tags,             {inputValue: Input tags},
      --input-words,            {inputValue: Input words},
      --input-dropout,          {inputValue: Input dropout},
      --output-model-path,      {inputValue: Output model URI template},

      --output-model-path-file, {outputPath: Output model URI},

    ]
```

Create the workflow in Python

```
@dsl.pipeline(
  name='Named Entity Recognition Pipeline',
  description='Performs preprocessing, training and deployment.'
)
def pipeline():

    preprocess_task = preprocess_operation(
        input_1_uri='gs://kubeflow-examples-data/named_entity_recognition_dataset/ner.csv,
        output_y_uri_template="gs://{}/{{workflow.uid}}/preprocess/y/data".format(BUCKET),
        output_x_uri_template="gs://{}/{{workflow.uid}}/preprocess/x/data".format(BUCKET),
        output_preprocessing_state_uri_template="gs://{}/{{workflow.uid}}/model".format(BUCKET)
    ).apply(kfp.gcp.use_gcp_secret('user-gcp-sa'))


    train_task = train_operation(
        input_x_uri=preprocess_task.outputs['output-x-uri'],
        input_y_uri=preprocess_task.outputs['output-y-uri'],
        input_job_dir_uri="gs://{}/{{workflow.uid}}/job".format(BUCKET),
        input_tags=preprocess_task.outputs['output-tags'],
        input_words=preprocess_task.outputs['output-words'],
        input_dropout=0.1,
        output_model_uri_template="gs://{}/{{workflow.uid}}/model".format(BUCKET)
    ).apply(kfp.gcp.use_gcp_secret('user-gcp-sa'))


    deploy_task = ai_platform_deploy_operation(
        model_path= train_task.output,
        model_name="named_entity_recognition_kubeflow",
        model_region="us-central1",
        model_version="version1",
        model_runtime_version="1.13",
        model_prediction_class="model_prediction.CustomModelPrediction",
        model_python_version="3.5",
        model_package_uris="gs://{}/routine/custom_prediction_routine-0.2.tar.gz".format(BUCKET)
    ).apply(kfp.gcp.use_gcp_secret('user-gcp-sa'))
```

Metaflow: less mature but cleaner API

- Run notebook code in cloud with a line of code (@batch)
  - Run experiments locally
  - Once ready, run code on AWS Batch
- Can run different steps of the same workflow in different envs

```
class RecSysFlow(FlowSpec):
    @step
    def start(self):
        self.data = load_data()
        self.next(self.fitA, self.fitB)

    # fitA requires a different version of NumPy compared to fitB
    @conda(libraries={"scikit-learn":"0.21.1", "numpy":"1.13.0"})
    @step
    def fitA(self):
        self.model = fit(self.data, model="A")
        self.next(self.ensemble)

    @conda(libraries={"numpy":"0.9.8"})
    # Requires 2 GPU of 16GB memory
    @batch(gpu=2, memory=16000)
    @step
    def fitB(self):
        self.model = fit(self.data, model="B")
        self.next(self.ensemble)

    @step
    def ensemble(self, inputs):
        self.outputs = (
                    (inputs.fitA.model.predict(self.data) +
                     inputs.fitB.model.predict(self.data)) / 2
                    for input in inputs
        )
        self.next(self.end)

    def end(self):
        print(self.outputs)
```

6. ML Platform

Model platform: story time

1. Anna started working on recsys at company X
2. To deploy recsys, Anna's team need to build tool like model deployment, model store, feature store, etc.
3. Other teams at X started deploying models and needed to build the same tools
4. X decided to have a $\underbrace{\text{centralized platform}}_{\text{ML Platform}}$ to serve multiple ML use cases

ML platform: key components

- Model deployment
- Model store
- Feature store

Deployment: online | batch prediction

See previous lectures

- Deployment service:
    - Package your model & dependencies
    - Push the package to production
    - Expose an endpoint for prediction

Deployment: online | batch prediction

- Deployment service:
  - Package your model & dependencies
  - Push the package to production
  - Expose an endpoint for prediction
- The most common MLOps tool
  - Cloud providers: SageMaker (AWS), Vertex AI (GCP), AzureML (Azure), etc.
  - Independent: MLflow Models, Seldon, Cortex, Ray Serve, etc.

Deployment: online | batch prediction

- Deployment service:
  - Package your model & dependencies
  - Push the package to production
  - Expose an endpoint for prediction
- The most common MLOps tool
- Not all can do batch + online prediction well
  - e.g. some companies use Seldon for online prediction, but Databricks for batch

Deployment service: model quality challenge

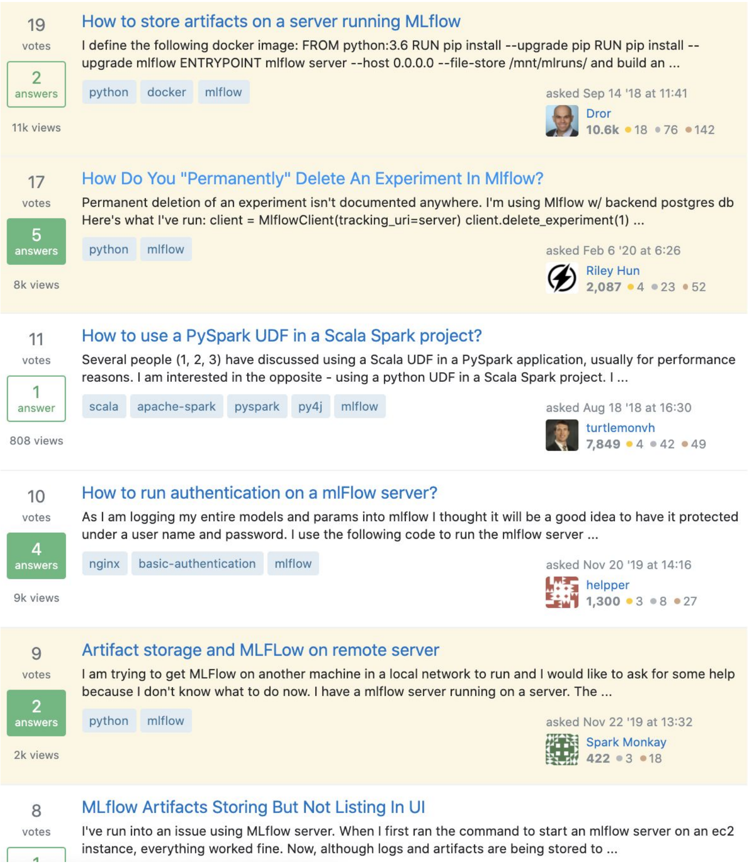
- How to ensure a model's quality pre- and during deployment?
  - Traditional code: CI/CD, PR review
  - ML: ???, ???

Model store

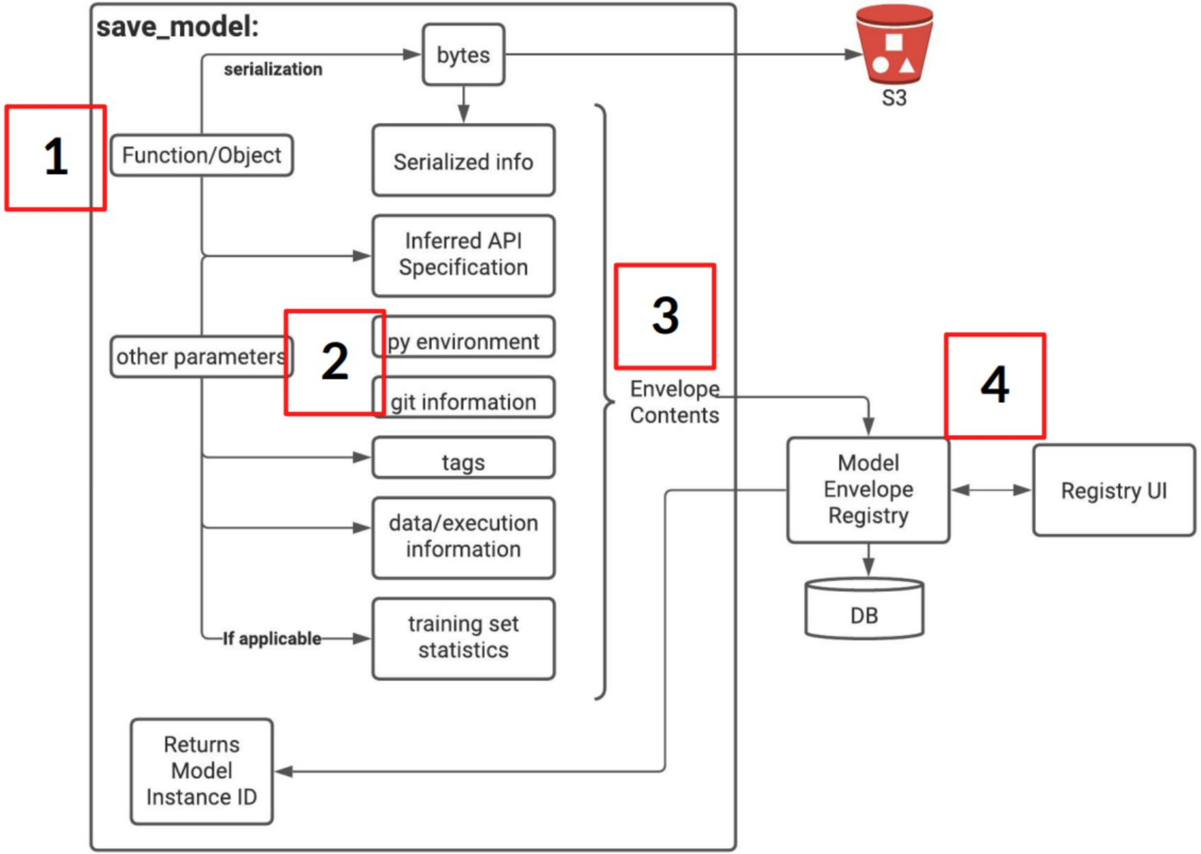
- Simplest form: store all models in blob storage like S3
- Problem:
  - When something happens, how to figure out:
    - Who/which team is responsible for this model?
    - If the correct model binary was deployed?
    - If the features used are correct?
    - If the code is up-to-date?
    - If something happened with the data pipeline?

Model store: artifact tracking

- Track all metadata necessary to debug a model later
- Severely underestimated



Model store: artifact tracking at Stitch Fix



Feature store: key challenges

1. Feature management
   a. Multiple models might share features, e.g. churn prediction & conversion prediction
   b. How to allow different teams to find & use high-value features discovered by other teams?

Feature store: key challenges

1. Feature management
2. Feature consistency
   a. During training, features might be written in Python
   b. During deployment, features might be written in Java
   c. How to ensure consistency between different feature pipelines?

Feature store: key challenges

1. Feature management
2. Feature consistency
3. Feature computation
    a. It might be expensive to compute the same feature multiple times for different models
    b. How to store computed features so that other models can use?

Feature store: key challenges

1. Feature management ← Feature catalog
2. Feature consistency
3. Feature computation ← Data warehouse

Other ML platform components

- Monitoring (ML & ops metrics)
- Experimentation platform
- Measurement (business metrics)

Evaluate MLOps tools

1. Does it work with your cloud provider?
2. Open-source or managed service?
3. Data security requirements

Machine Learning Systems Design

Deployment and Monitoring

Next Lecture: Model Maintenance

CE 40959 Spring 2023
Ali Zarezade
SharifMLSD.github.io