

# WAYS TO PRODUCTIONIZE H2O

Tom Kraljevic

[tomk@h2o.ai](mailto:tomk@h2o.ai)

H2O Open Tour 2016, New York City

July 19, 2016

# WHERE TO FIND THE SLIDES

They're in GitHub...

[h2oai/h2o-meetups/tree/master/  
2016\\_07\\_19\\_H2O\\_Open\\_Tour\\_NYC\\_Prod](https://github.com/h2oai/h2o-meetups/tree/master/2016_07_19_H2O_Open_Tour_NYC_Prod)

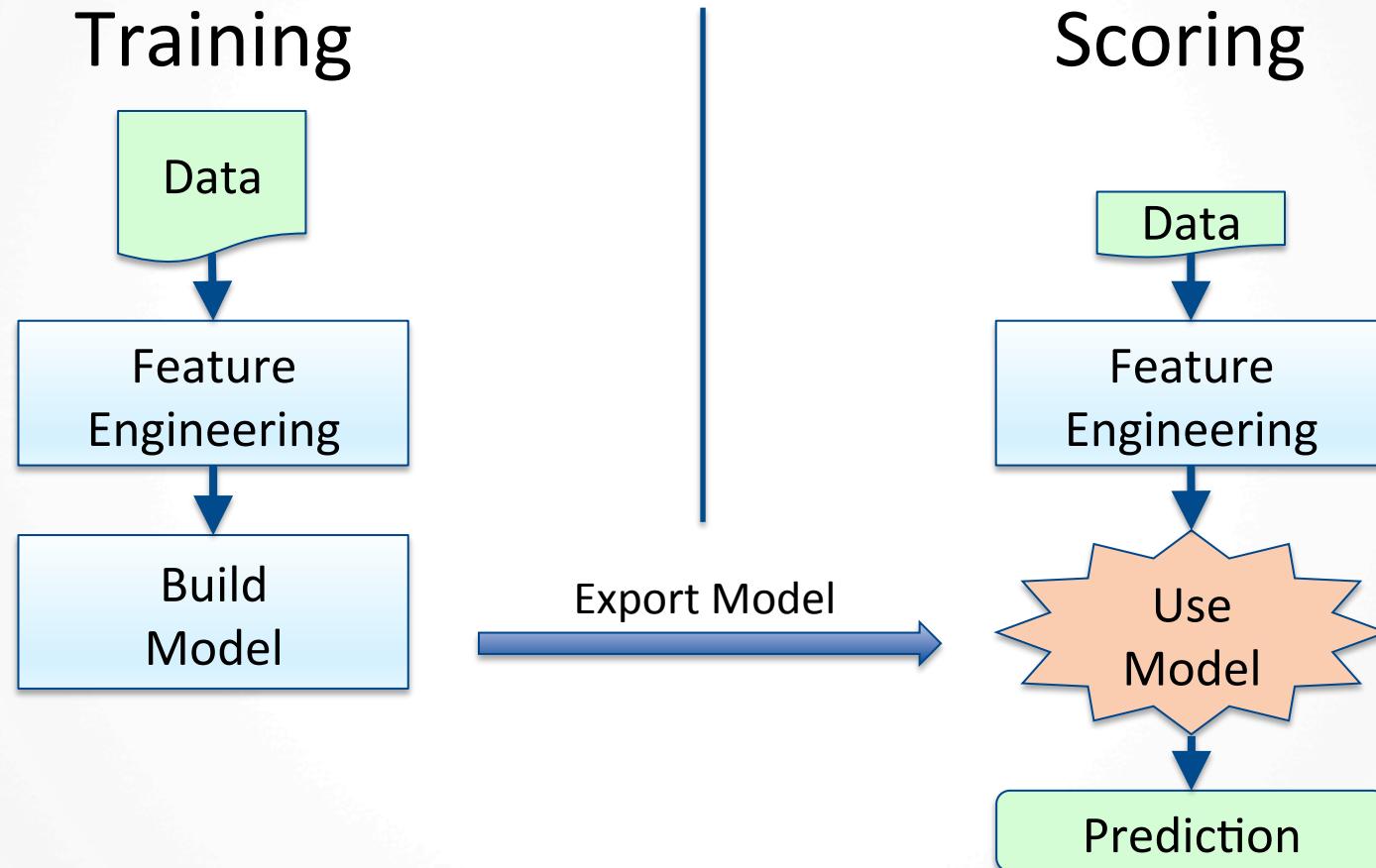
# DEFINING “IN PRODUCTION”?

- Removal would impact value to the business
  - Dollars
  - Time

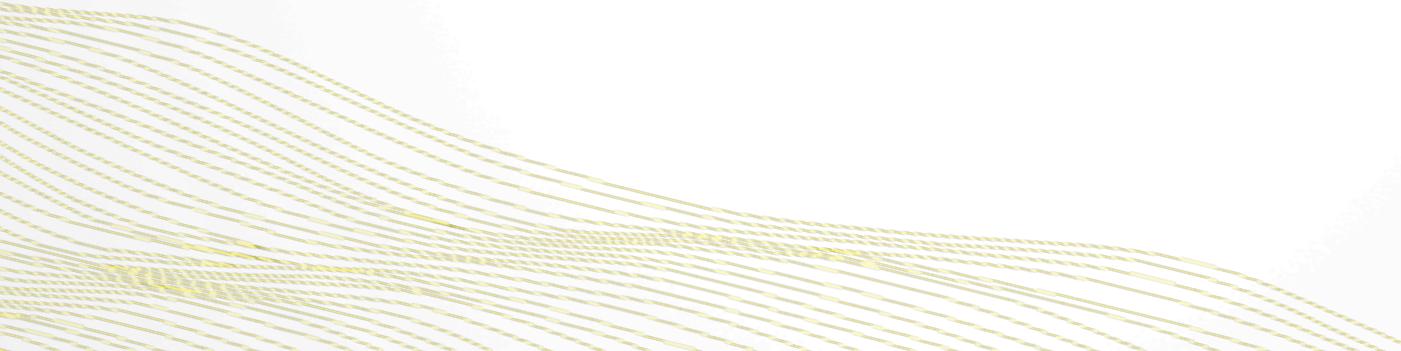
# OUTLINE

- Motivation
- Productionizing Model Building
  - Training
- Productionizing Model Use
  - Scoring aka Predicting
- Q & A

# TYPICAL DEPLOYMENT APPROACH



# **PRODUCTIONIZING MODEL BUILDING (TRAINING)**



# H2O TRAINING DIMENSIONS

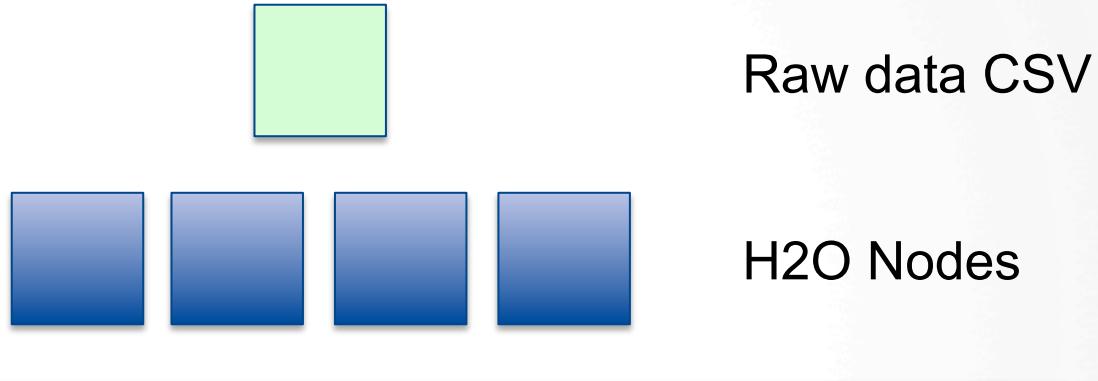
- Where does the data live?
  - Hadoop
    - HDFS, Hive
  - S3 / AWS
  - Database
  - Local drive (laptop, desktop, server)
- How big is the data?
  - Fits in memory of one host? (Or not)
  - “Too small”?
- How often should models be re-trained?
  - Daily? Weekly? Monthly? Quarterly?
- Where will model training happen?
  - Hadoop YARN job
  - EC2 instances
  - Bare metal cluster
  - A really big server
  - Laptop
- Resource requirements (how many nodes?)
  - CPUs
  - Memory
  - Time to build a model

# H2O TRAINING SIZING BEST PRACTICE

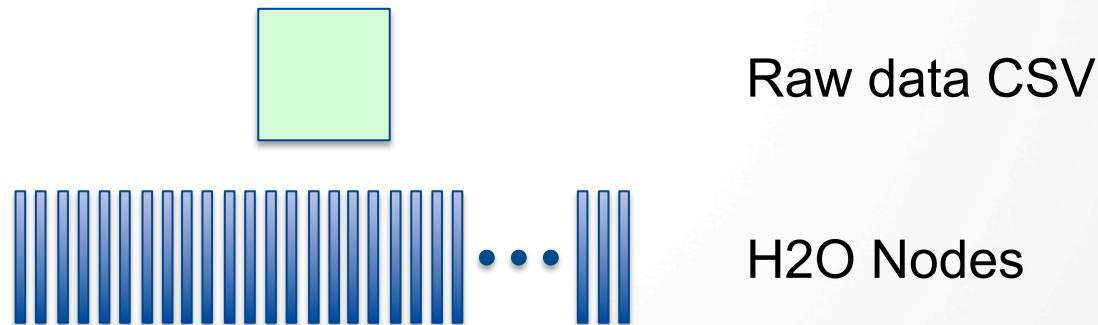
- For a novice getting started with a decent sized dataset:
  - Size total cluster memory to 4x on-disk data size (data for model training, not “total data”)
  - Use nodes of at least 10 GB, preferably larger
  - Use a small number of big nodes
    - e.g. 8 nodes x 30 GB each
    - For a YARN environment, you will have to adjust default settings to start big containers
  - **Definitely DO NOT use 200 nodes x 1 GB each**
    - Caution: Hadoop MapReduce veterans often start this way!
    - 1 GB is way too small for *any* workload!

# H2O TRAINING SIZING BEST PRACTICE

Start  
With This



For the  
Experts

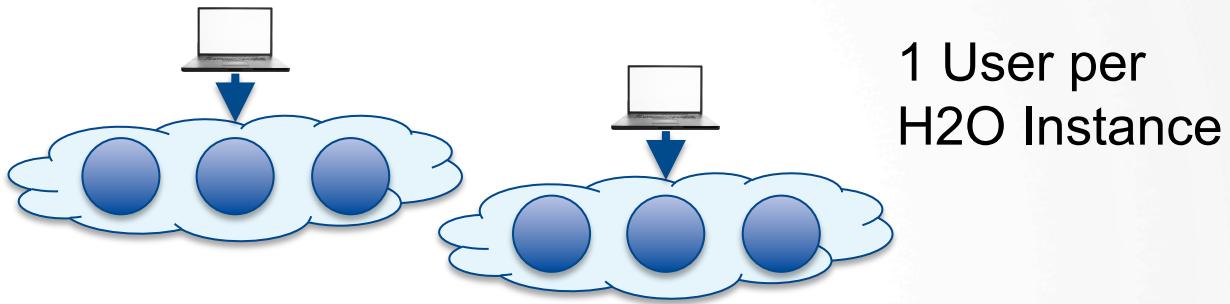


# H2O USER PROVISIONING BEST PRACTICE

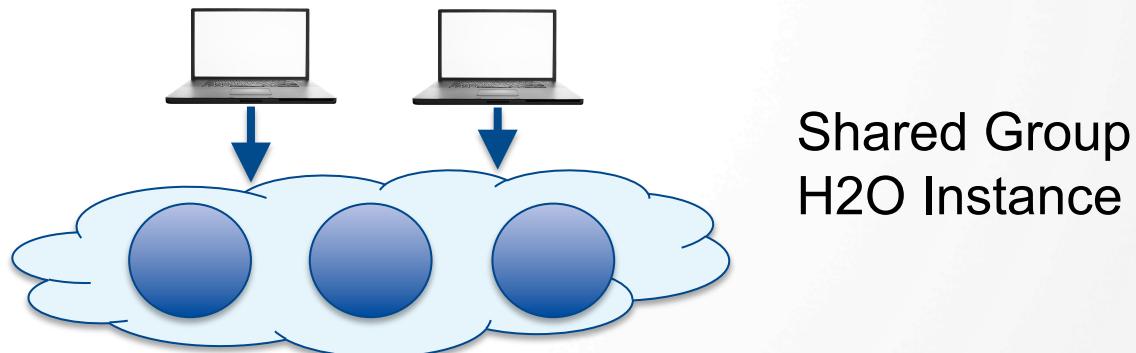
- Individual H2O jobs (aka application instances, aka clusters) are not multi-tenant
  - Think of YARN as providing multi-tenancy and H2O as a single job
    - H2O instance per interactive user, or per batch job
  - Steam product from H2O.ai simplifies management of H2O clusters

# H2O USER PROVISIONING BEST PRACTICE

Recommended



Not Recommended



1 User per  
H2O Instance

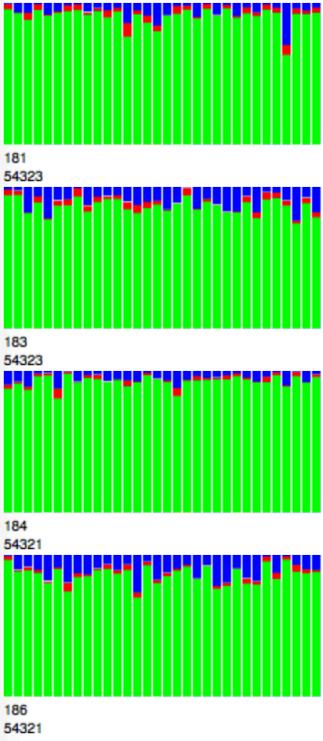
Shared Group  
H2O Instance

# H2O MONITORING BEST PRACTICE

- Monitor your H2O instance's resource utilization
  - Watch the Water Meter in the Flow Web UI to see:
    - If your hardware cluster is already being slammed before H2O even starts doing anything
    - If your H2O instance is using CPU when you expect it to
  - Look at H2O log files for detailed memory utilization information (advanced)

# H2O MONITORING BEST PRACTICE

## CPU Monitoring



## Memory Monitoring (advanced)

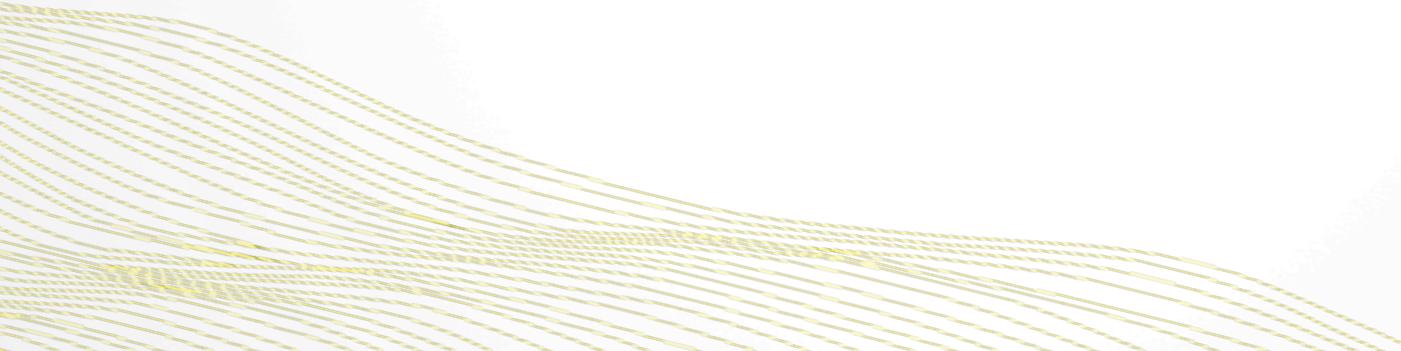
yarn logs -applicationId application\_1457562501251\_2398

```
07-18 20:18:03.364 172.16.2.184:54321 31423 main INFO: ----- H2O started -----
07-18 20:18:03.430 172.16.2.184:54321 31423 main INFO: Build git branch: rel-turchin
07-18 20:18:03.430 172.16.2.184:54321 31423 main INFO: Build git hash: c09d9da94e7c61eb7ef4573f9f5029e0ca501469
07-18 20:18:03.430 172.16.2.184:54321 31423 main INFO: Build git describe: jenkins-rel-turchin-3
07-18 20:18:03.431 172.16.2.184:54321 31423 main INFO: Build project version: 3.8.2.3
07-18 20:18:03.431 172.16.2.184:54321 31423 main INFO: Build time: 2016-04-25 13:07:53
07-18 20:18:03.431 172.16.2.184:54321 31423 main INFO: Built on: 2016-04-25 13:07:53'
07-18 20:18:03.431 172.16.2.184:54321 31423 main INFO: Java availableProcessors: 32
07-18 20:18:03.431 172.16.2.184:54321 31423 main INFO: Java heap totalMemory: 39.45 GB
07-18 20:18:03.431 172.16.2.184:54321 31423 main INFO: Java heap maxMemory: 39.45 GB
07-18 20:18:03.431 172.16.2.184:54321 31423 main INFO: Java version: Java 1.7.0_67 (from Oracle Corporation)
07-18 20:18:03.431 172.16.2.184:54321 31423 main INFO: JVM launch parameters: [-XX:NewRatio=8, -Djava.net.preferIPv4Stack=true, -Dhdp.version=2.2.6.3-1, -Xms40g,
-Xmx40g, -XX:PermSize=256m, -verbose:gc, -XX:+PrintGCDetails, -XX:+PrintGCTimeStamps, -Dlog4j.configurationFile=/opt2/hdp/yarn/log/application_1457562501251_2398/container_e13_1457562501251_2398/_log4j2.xml, -Dlog4j.configurationFile=/opt2/hdp/yarn/log/application_1457562501251_2398/container_e13_1457562501251_2398/_log4j2.xml, -Dyarn.app.container.log.dir=/opt2/hdp/yarn/log/application_1457562501251_2398/container_e13_1457562501251_2398/_log4j2.xml, -Dyarn.app.container.log.filesize=0, -Dlog4j.root.logger=INFO, CLA]
07-18 20:18:03.431 172.16.2.184:54321 31423 main INFO: OS version: Linux 3.13.0-46-generic (amd64)
07-18 20:18:03.431 172.16.2.184:54321 31423 main INFO: Physical memory: 251.89 GB
07-18 20:18:03.431 172.16.2.184:54321 31423 main INFO: Maximum physical memory: 251.89 GB
07-18 20:18:03.431 172.16.2.184:54321 31423 main INFO: User name: 'tomk'
07-18 20:18:03.432 172.16.2.184:54321 31423 main INFO: Possible IP Address: br2 (br2), 172.16.2.184
07-18 20:18:03.432 172.16.2.184:54321 31423 main INFO: Possible IP Address: lo (lo), 127.0.0.1
07-18 20:18:03.432 172.16.2.184:54321 31423 main INFO: Internal communication uses port: 54322
07-18 20:18:03.432 172.16.2.184:54321 31423 main INFO: Listening for HTTP and REST traffic on http://172.16.2.184:54321/
07-18 20:29:58.542 172.16.2.184:54321 31423 F1-->1 INFO: Frame distribution summary:
07-18 20:29:58.542 172.16.2.184:54321 31423 F1-->1 INFO: Size Number of Rows Number of Chunks per Column Number of Chunks
07-18 20:29:58.542 172.16.2.184:54321 31423 F1-->1 INFO: 172.16.2.181:54323 918.6 MB 29811120.000000 131.000000 4061.000000
07-18 20:29:58.542 172.16.2.184:54321 31423 F1-->1 INFO: 172.16.2.183:54323 883.7 MB 290223380.000000 127.000000 3937.000000
07-18 20:29:58.542 172.16.2.184:54321 31423 F1-->1 INFO: 172.16.2.184:54323 919.5 MB 29811120.000000 127.000000 3937.000000
07-18 20:29:58.542 172.16.2.184:54321 31423 F1-->1 INFO: 172.16.2.186:54321 921.5 MB 28989760.000000 127.000000 3937.000000
07-18 20:29:58.542 172.16.2.184:54321 31423 F1-->1 INFO: mean 911.8 MB 29173914.000000 128.000000 3968.000000
07-18 20:29:58.542 172.16.2.184:54321 31423 F1-->1 INFO: min 883.7 MB 28989760.000000 127.000000 3937.000000
07-18 20:29:58.542 172.16.2.184:54321 31423 F1-->1 INFO: max 925.5 MB 29811120.000000 131.000000 4061.000000
07-18 20:29:58.542 172.16.2.184:54321 31423 F1-->1 INFO: stdDev 16.5 MB 370495.062500 1.732051 53.693577
07-18 20:29:58.542 172.16.2.184:54321 31423 F1-->1 INFO: total 3.56 GB 116695256.000000 512.000000 15872.000000
792, 586: [GC [PSYoungGen: 2353508K->-15533946K (3107328K)] 6945343K->-6408429K (40390144K), 0.1256170 secs] [Times: user=2.19 sys=0.53, real=0.12 secs]
793, 632: [GC [PSYoungGen: 3107328K->-1553392K (3107328K)] 7962139K (40390144K), 0.1929720 secs] [Times: user=2.61 sys=1.67, real=0.20 secs]
794, 524: [GC [PSYoungGen: 3106455K->-1221115K (3107328K)] 8903127K->-8517453K (40390144K), 0.1658980 secs] [Times: user=2.92 sys=0.77, real=0.17 secs]
795, 684: [GC [PSYoungGen: 160125206K->-9965740K (40390144K), 0.1161300 secs] [Times: user=2.81 sys=0.82, real=0.16 secs]
796, 684: [GC [PSYoungGen: 3050216K->-9865516K (3107328K)] 11519362K->-10546313K (40390144K), 0.1470070 secs] [Times: user=2.03 sys=1.27, real=0.15 secs]
797, 983: [GC [PSYoungGen: 2540021K->-781681K (3107328K)] 12099782K->-110068/9K (40390144K), 0.1154810 secs] [Times: user=1.98 sys=1.21, real=0.12 secs]
```

# ADDITIONAL YARN BEST PRACTICES

- Limiting memory occurs “naturally”
- No CPU usage limit by default, but you can use:
  - YARN cgroups
  - H2O –nthreads startup option
- Use node labels to direct container placement
- Capacity and Fair schedulers are both fine
- Many people provision YARN queues to manage resource use for teams
- You *must* disable preemption for H2O jobs

# **PRODUCTIONIZING MODEL USE (SCORING AKA PREDICTING)**



# HIGH-LEVEL H2O DIMENSIONS

## In-H2O

- Batch
- Many rows at once
  - think Frames of data
- R / Python / Scala
  - script driving H2O
- Script

## POJO

- Real-time
- One row at a time
- Java / Scala
  - embedded or as a service
- Application

# OTHER INDEPENDENT DIMENSIONS

## Choice A

- On-premise
- Data Science
- Micro-service with REST API
- Model only
- Train and predict together
- Pre-calculated

## Choice B

- Cloud
- Computer Science
- Embedded
- Model + Feature engineering
- Train and predict separately
- Calculate live

# DATA ENVIRONMENTS

- Hive
- Databases
  - Oracle, MS SQL, SQLite, etc.
- Cloud databases  
(Amazon RDS)
- Teradata
- Bare flat files
- Hadoop HDFS
- Amazon S3
- Storm spout
- Spark stream
- Apache Apex

# MODEL ENVIRONMENTS

- Laptop / Desktop
- Hadoop
- Standalone cluster
- Steam
- Cloud (e.g. EC2)
- Cloud endpoint (e.g. AWS Lambda)
- Storm bolt
- Spark stream
- Apache Apex
- Hive UDF
- POJO called from R

# KEY PARAMETERS

- Model training
  - Frequency
  - Cost (memory, CPU, time)
- Prediction SLA
  - Throughput
  - Latency
  - Availability
- Security

# EXAMPLE DESIGN PATTERNS

## In-H2O

- Driven from Flow/R/  
Python/Scala
  - Often launching on Hadoop
- Embedding H2O (droplet)

## POJO

- JUnit test
- Craigslist app (Sparkling Water)
- App-consumer-loan (Jetty servlet)
- App-malicious-domain (AWS lambda)
- POJO from R (jar file)
- Journals.ai desktop app (SQLite DB)
- Storm bolt
- Hive UDF
- Steam from H2O.ai (REST API)

# WALKTHROUGH FOR 5 DESIGN PATTERNS

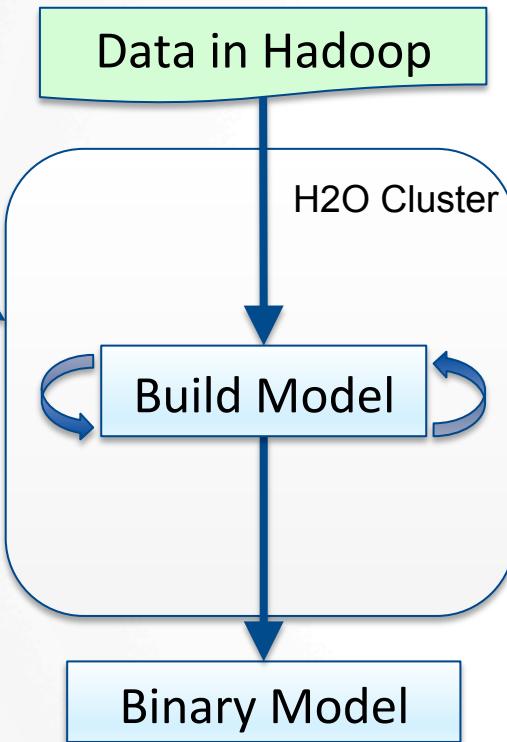
- Batch In-H2O
- Steam
- AWS Lambda
- Hive UDF
- Pre-calculated DB

# PURE BATCH H2O DESIGN PATTERN

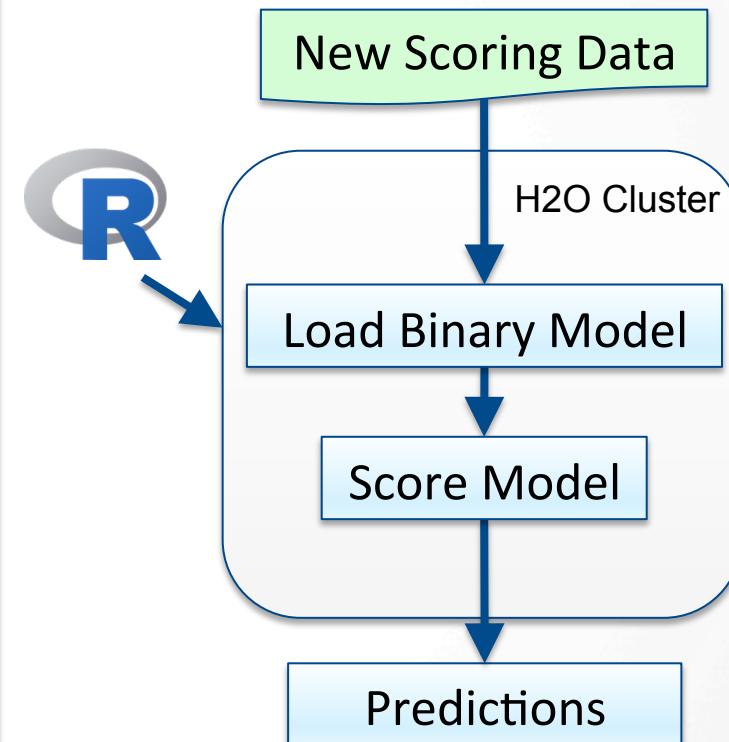
- Training and scoring data lives in data lake
- Entire process driven by R
- Training occurs periodically
- Scoring occurs frequently

# PURE BATCH H2O DESIGN PATTERN

## Training



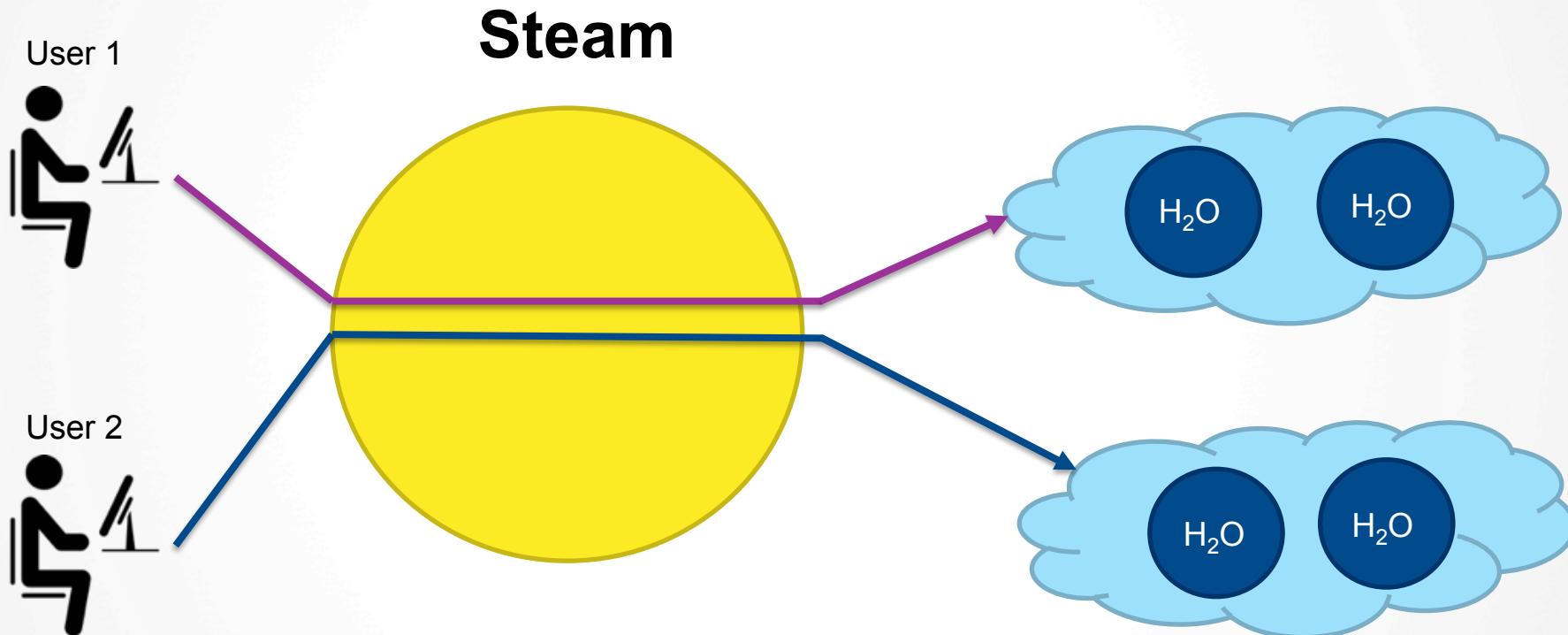
## Scoring



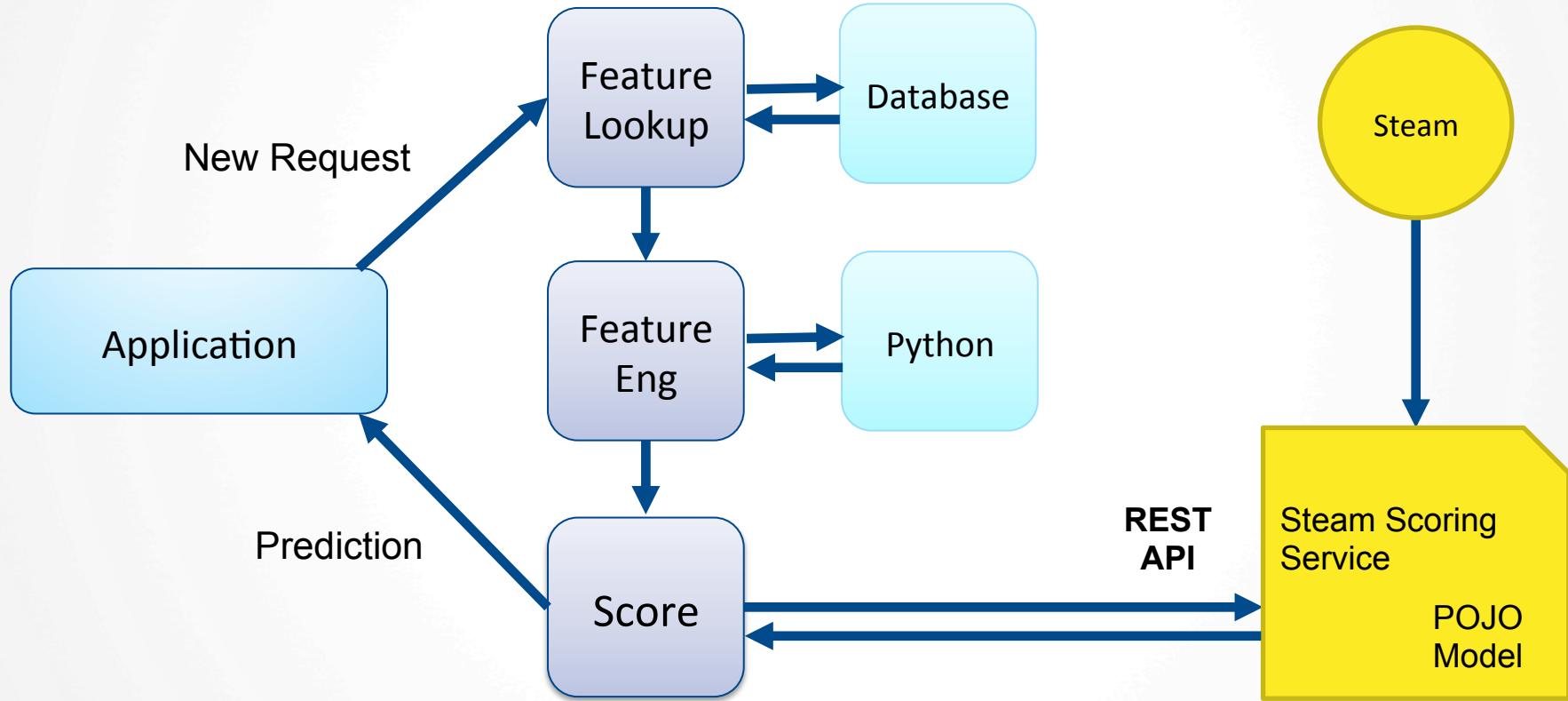
# STEAM DESIGN PATTERN

- Training data lives in Hadoop
- POJO models are deployed as a service with a REST API

# STEAM DESIGN PATTERN (TRAINING)



# STEAM DESIGN PATTERN (SCORING)



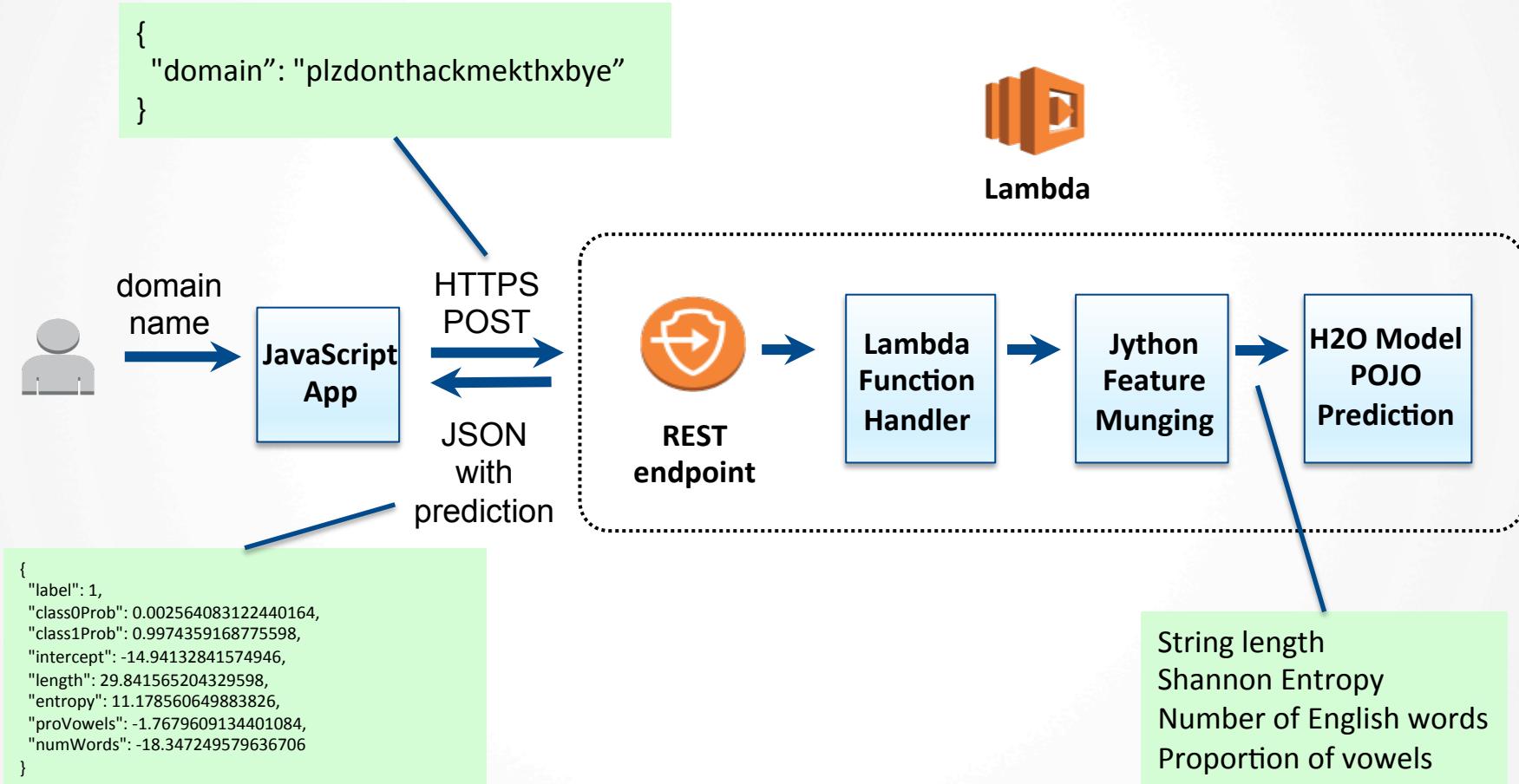
# AWS LAMBDA DESIGN PATTERN

- Training data lives in data lake
- Scoring data arrives via API call
- Real-time POJO scoring in AWS Lambda

# MOTIVATION FOR AWS LAMDA

USE CASE	REQUIREMENTS	TECHNOLOGY
Feature engineering	<ul style="list-style-type: none"><li>- Use of Python for DataSci</li><li>- Code re-use for training and production</li><li>- Speed</li></ul>	Jython
Machine Learning Prediction	<ul style="list-style-type: none"><li>- High model accuracy</li><li>- Real-time environment</li></ul>	H2O Generated POJO Model
Deployment to Production	<ul style="list-style-type: none"><li>- Easy handoff to Ops</li><li>- Speed to deployment</li><li>- Built-in scalability</li><li>- No infrastructure management</li></ul>	AWS Lambda

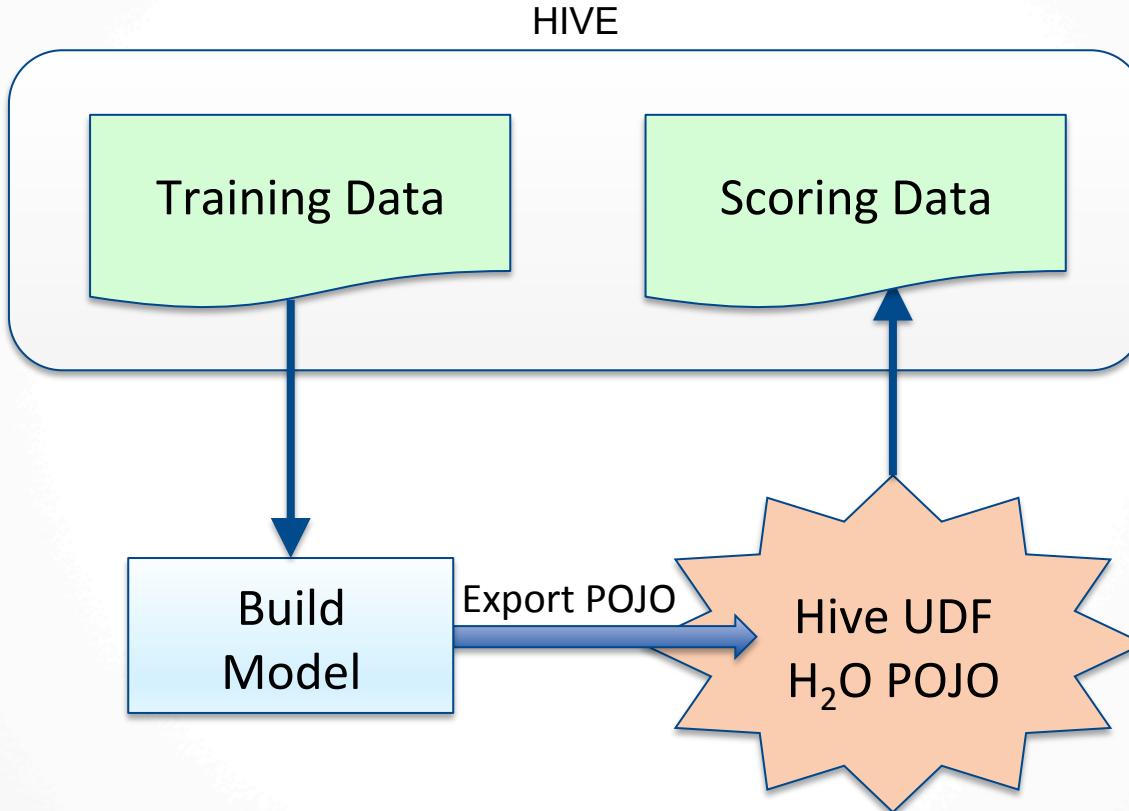
# AWS LAMBDA DESIGN PATTERN (SCORING)



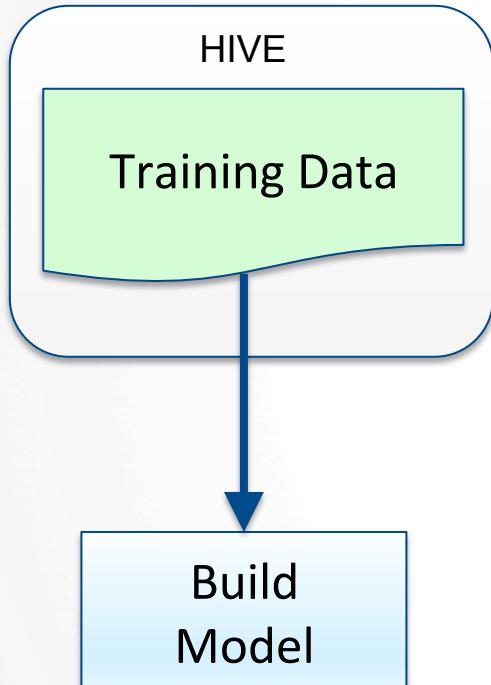
# HIVE UDF DESIGN PATTERN

- Training data lives in Hive
- Scoring data lives in Hive
- Batch POJO scoring
- Models built infrequently
- Scoring occurs daily
  - Strict SLA to complete the batch job

# HIVE UDF DEPLOYMENT

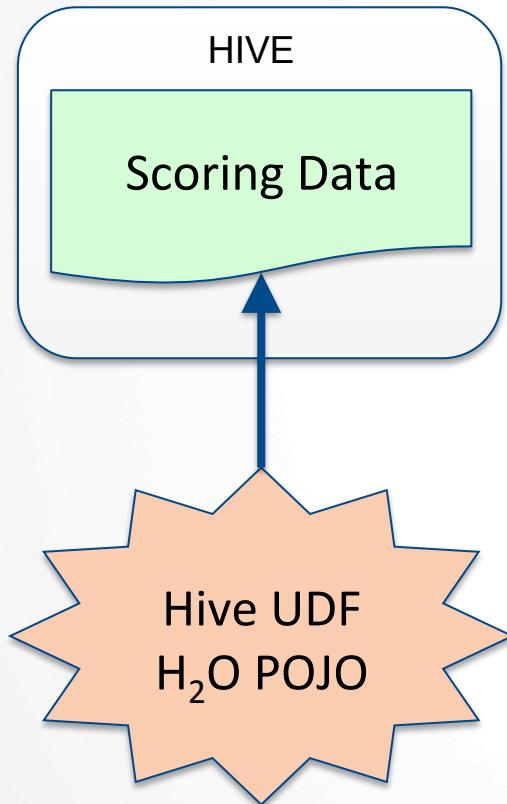


# HIVE UDF (TRAINING)



- Prepare training data
  - Data may be stored in unreadable format!
  - Export to read \_part files for  $H_2O$
- Launch  $H_2O$
- Build models
- Export models as POJOs

# HIVE UDF (SCORING)



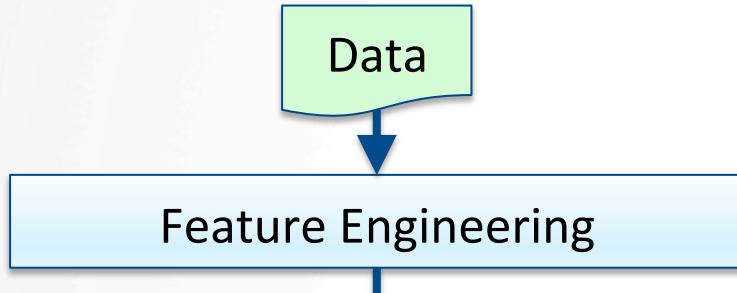
- Create Hive UDF
  - Extend GenerUDF class
  - initialize(): initialize H<sub>2</sub>O models
  - evaluate(): loop through models & score
  - Build JAR file
- Hive
  - Load JARs, define UDF, set columns
    - ADD JAR localjars/h2o-genmodel.jar;
    - ADD JAR target/ScoreData-1.0-SNAPSHOT.jar;
    - CREATE TEMPORARY FUNCTION fn AS ...
    - set hivevar:scoredatacolnames=C1,C2...
  - Score!
    - select fn(\${scoredatacolnames}) from TABLE

# JOURNALS.AI DESIGN PATTERN

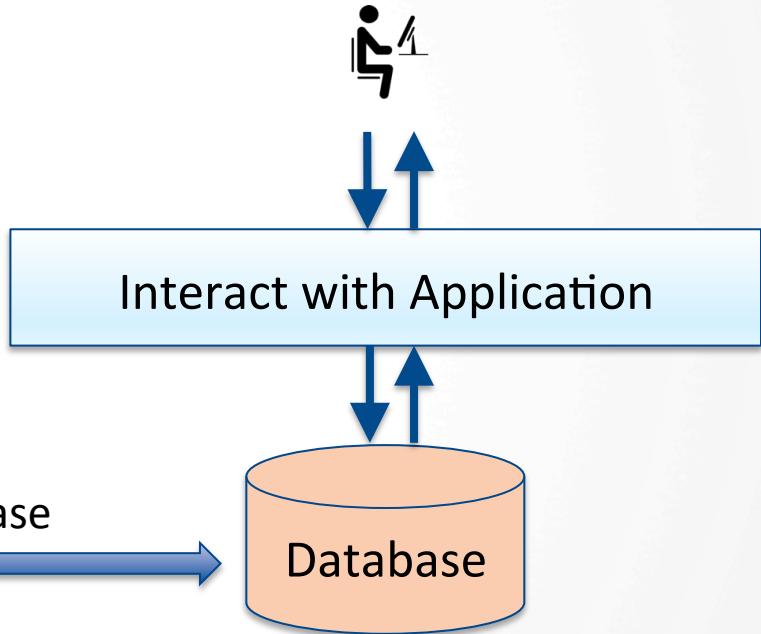
- Driven by R
- Training & scoring on bare metal servers
- Scores pre-calculated and stored in a database  
(e.g. MS SQL, SQLite)
- Application with domain-specific UI built on top  
of the database
- Flexible deployment (laptop, server, cloud, etc.)

# JOURNALS.AI PRODUCTIONIZED

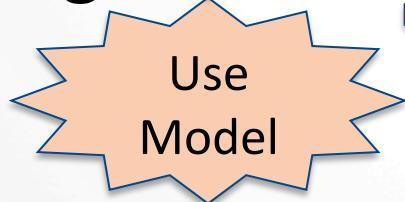
## Training



## App Deployment



## Scoring

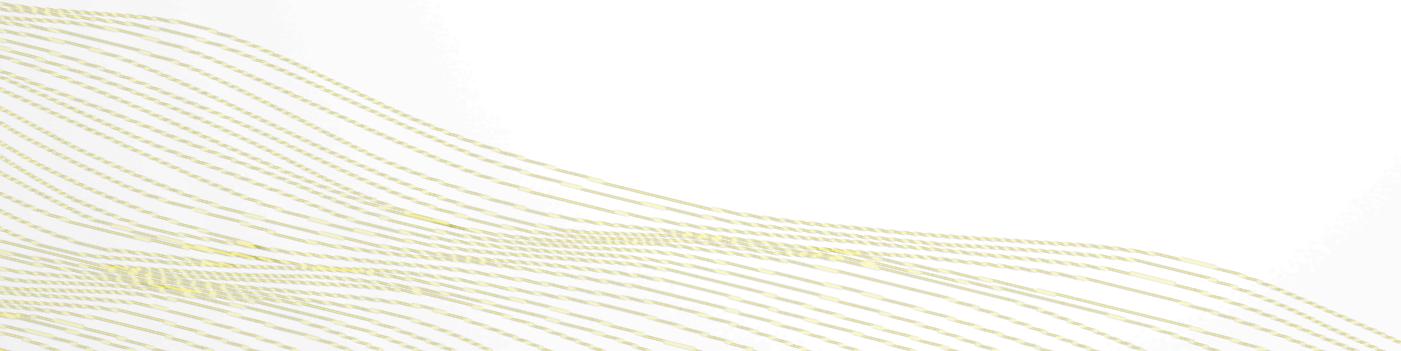


# Q & A

- Thanks for attending!
- Slides in github:
  - [h2oai/h2o-meetups/tree/master/  
2016\\_07\\_19\\_H2O\\_Open\\_Tour\\_NYC\\_Prod](https://github.com/h2oai/h2o-meetups/tree/master/2016_07_19_H2O_Open_Tour_NYC_Prod)
- Send follow up questions to:

Tom Kraljevic  
[tomk@h2o.ai](mailto:tomk@h2o.ai)

# **FEATURE ENGINEERING**



# FEATURE ENGINEERING

- For a model in production to make predictions, it needs access to the same features that were trained on
- Often there is a need to build the same features in both training and production environments

# FEATURE ENGINEERING DIMENSIONS

- Language: R / Python / Java / Scala
- Code re-use for train and prod: Yes / No
- Prod is batch / interactive
- Latency SLA
- Throughput SLA
- Live lookups: Yes / No

# FEATURE ENGINEERING RECIPIES

- Think like a pipeline
- Function call to transform a row
  - Column Name/Value map in -> map out
- Transformation microservice
- Storm feature engineering bolt
- Jython
- Live Python interpreter

# FEATURE ENGINEERING STEPS

- Database lookup
- Imputation
- Sanity checks (e.g. age is positive)
- Unknown categorical level check
- Creating new features

# Production Pipeline

