

**BΞΔM**  
S U M M I T

# Real-Time Predictive Modeling with MLServer, MLFlow, and Apache Beam



# Agenda



- Background
  - Who are we?
  - The problem that we're solving
  - A quick architecture overview
- Training
  - Getting the data (Clickhouse)
  - Training our models (SKLearn)
  - Managing our models (MLFlow)
- Deploying
  - Syncing MFlow, MLServer, and Dataflow via GCS
- Scoring
  - Getting the data (PubSub)
  - Tensor Forming (Beam)
  - Scoring (MLServer)
  - Embedded vs External Inference
- After v1
  - Cost Optimization
  - Shared Resources
- Conclusions
- Q&A

# Background

How we got here

Devon Peticolas

Principal Engineer at Oden

One of Oden's first engineers  
(responsible for many bad engineering  
decisions but Beam is not one of them)

I wrote my first beam job in 2018

This is my fourth Beam Summit!

Jeswanth Yadagani

Senior ML Engineer at Oden

Not one of Oden's first engineers  
(still feeling the pain for those other bad  
engineering decisions)

I wrote my first beam job in 2020

This is my third Beam Summit!



# Who is Oden Technologies?



## Oden Technologies

- Think “New Relic but for manufacturing”
- Real-time and historical analytics for manufacturing
- Customers in plastics, chemical, paper
- We have lots of time-series data
- Productized machine learning on AI



Now Dashboards Explore Discover Labs Go To Legacy Explore

### Explore Oden Analytics

GROUP BY Line State Category State Reason

TIME Current week SPLIT BY TIME Select SET WEEKDAY All week + 12am...

FILTERS All factories All lines All products

Search

Back Showing Sunday, February 12, 2023 12:00 AM - Monday, February 13, 2023 1:12 PM

Top Downtime Reasons by Line Product Performance by Runtime Top Scrap by Product

Timeseries	Line	State Category	State Reason	Factory	Product	R	When	Duration
View...	B4	Unplanned Downtime	Machine Breakdown	Factory B	948BB635	V	2/12/2023 12:00am	2/13/2023 1:12pm
View...	Compounding 1	Downtime	-	Factory A	6D5009BZ	U	2/12/2023 12:00am	2/13/2023 1:12pm
View...	B1	Unplanned Downtime	Reel Change	Factory B	AAA9B11B	V	2/12/2023 12:00am	2/13/2023 1:12pm
View...	Extruder 3	Downtime	-	Factory A	+5	+	2/12/2023 12:00am	2/13/2023 1:12pm
View...	Extruder 3	Unplanned Downtime	Machine Breakdown	Factory A	+6	+	2/12/2023 12:00am	2/13/2023 1:12pm
View...	B4	Downtime	-	Factory B	41F9238D	V	2/12/2023 12:00am	2/13/2023 1:12pm
View...	Extruder 3	Unplanned Downtime	Machine Jam	Factory A	+6	+	2/12/2023 12:00am	2/13/2023 1:12pm
View...	Extruder 3	Unplanned Downtime	No Operator	Factory A	+7	+	2/12/2023 12:00am	2/13/2023 1:12pm
View...	Extruder 1	Unplanned Downtime	Quality Issue	Factory A	109	+	2/12/2023 12:00am	2/13/2023 1:12pm

Now Dashboards Explore Discover Labs Process AI

Process AI Extrusion Line BA7BEC4

Stable periods (10/4/2023 - 4/4/2024) Select a point below to view its process settings

EXTRUSION LINE ACTUAL LINE SPEED (PV)  
CURRENT 190.00 > RECOMMENDED 227.24

EXTRUDER 1 RATIO (PV)  
CURRENT 85.00 > RECOMMENDED 90.00

EXTRUDER 2 RATIO (PV)  
CURRENT 39.50 > RECOMMENDED 43.41

EXTRUDER 1 ZONE 1 TEMP (SP)  
CURRENT 275.00 > RECOMMENDED 275.00

EXTRUDER 1 ZONE 2 TEMP (SP)  
CURRENT 280.00 > RECOMMENDED 295.00

EXTRUDER 1 ZONE 3 TEMP (SP)  
CURRENT 290.00 > RECOMMENDED 305.00

EXTRUDER 1 ZONE 4 TEMP (SP)  
CURRENT 300.00 > RECOMMENDED 320.00

EXTRUDER 1 ZONE 5 TEMP (SP)

BARE OD  
ACTUAL 491.617 NOMINAL 498

HOT OD  
ACTUAL 690.997 NOMINAL 661

COLD OD  
ACTUAL 667.148 NOMINAL 658

WALL THICKNESS  
ACTUAL 87.765 NOMINAL 80

CURRENT EXCESS WALL THICKNESS (IN) 9.707 PROJECTED EXCESS WALL THICKNESS (IN) -2.717 PROJECTED MATERIAL SAVINGS (IN) 11.315



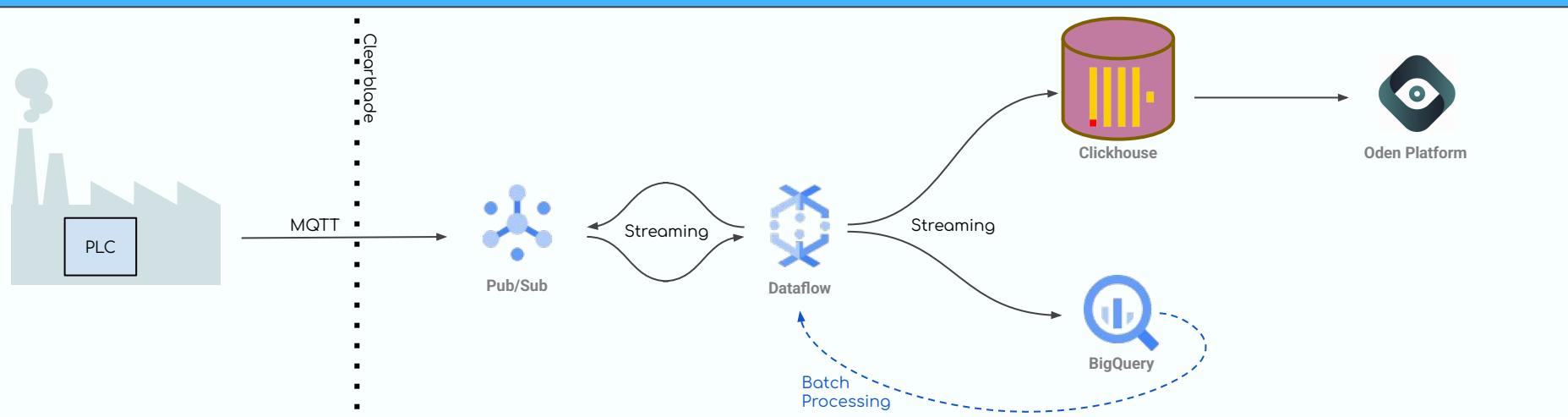
# How Does Oden Use Beam

## Streaming

- Processing of “raw” manufacturing data via MQTT (Clearblade to PubSub)
- Stateful and windowed transformation and state change detection
- Delivery into Clickhouse and BigQuery

## Batch

- The same jobs we run in streaming but in a special “batch mode” for:
  - Backfills
  - Late Data Processing
  - Outage Recovery



# Example Job: Calculated Metrics

User Has

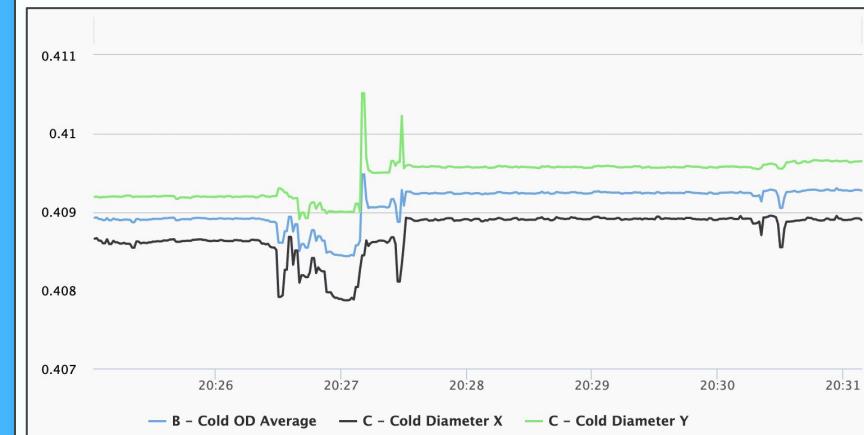
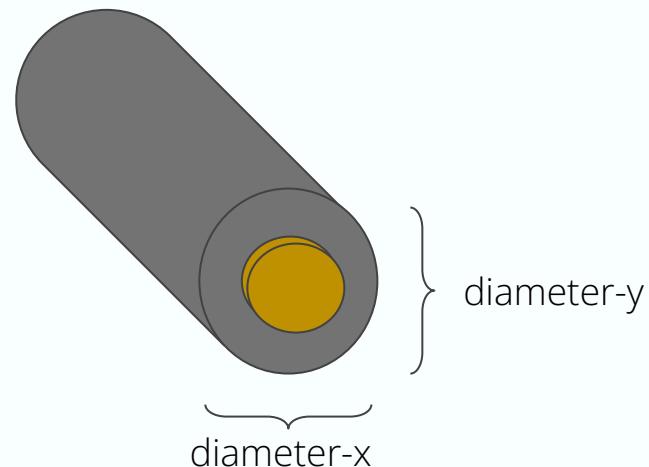
diameter-x and diameter-y

User Wants

avg-diameter = (diameter-x + diameter-y) / 2

Calculated Metrics

- New “calculated” metrics need to be computed in real-time
- Components come from different sensors
- User-defined formulas stored in Postgres
- New calculated metrics are treated just like “real” metrics



# Predictive Metrics

Customers perform “off line” tests of their product to determine quality

- Paper
  - Stretch until tear testing
  - Folding and crushing
- Ink
  - Color spectrum testing
  - Particulate distribution testing
- Wire
  - Tensile strength
  - Wall thickness

These measures come 40m to 2d after production.

## Hypothesis

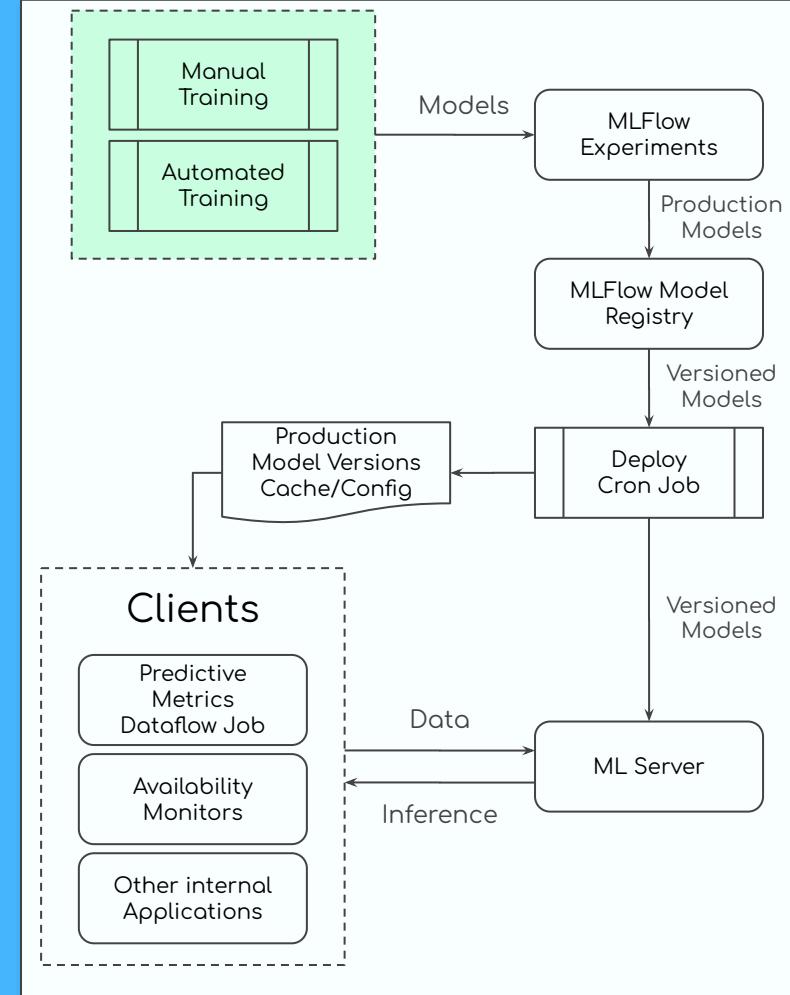
In-line measurement can predict off-line quality



# Architecture Overview

# Architecture Overview

- Experiments are conducted locally to fit the appropriate model, features and pipeline.
- Automated Training is orchestrated via Airflow using Cloud Run Jobs.



# Architecture Overview

- MLFlow Experiments: Stores information about ML model training and experiments along with metrics, model objects, and supporting artifacts.
- MLFlow Model Registry: Allows versioning of production models.

**mlflow**



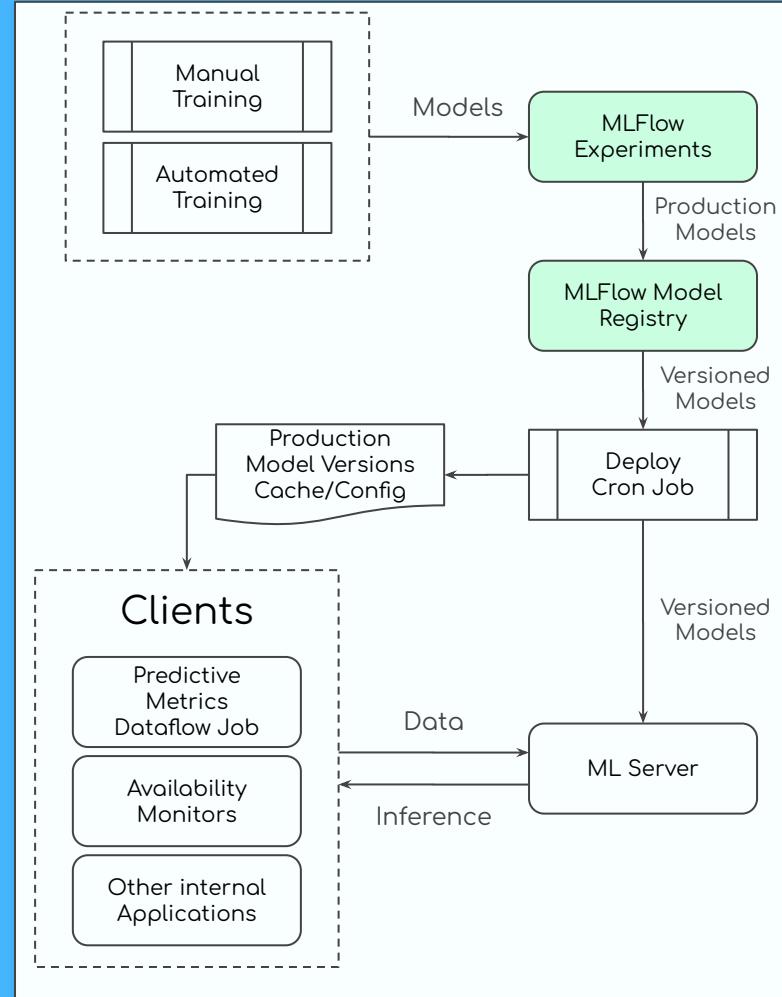
Cloud Storage



Cloud Run



Cloud SQL



# Architecture Overview

- MLFlow Experiments: Stores information about ML model training and experiments along with metrics, model objects, and supporting artifacts.
- MLFlow Model Registry: Allows versioning of production models.

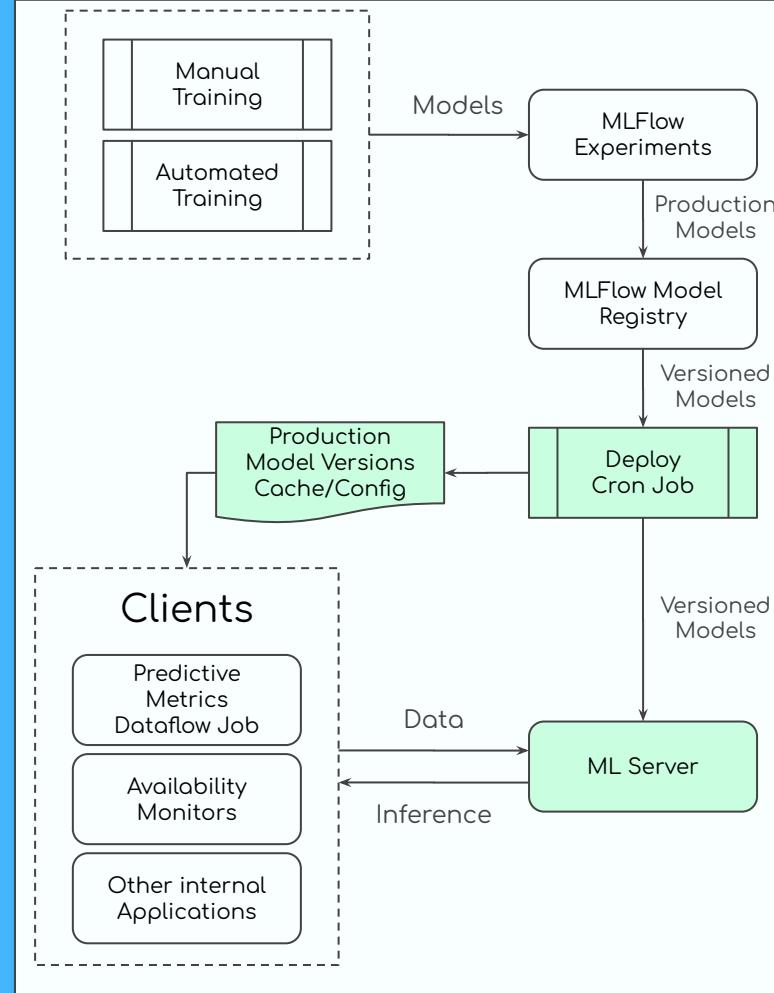


Run Name	Created	Duration	Metrics
Residual Model Experiments - Residual Model All Features were inputs into Residual model	12 days ago	0.725762...	16.962935... 0.0741904... 0.4651808... 0.7426245...
Residual Model Experiments - Residual Model Not Strongly Regularized	12 days ago	0.6934766...	19.30471... 0.0841287... 0.4065453... 0.7037975...
Residual Model Experiments - Residual Model Super Regularized XGBoost (Sequential Split) - RecSys ...	12 days ago	0.7958151...	16.568932... 0.0720921... 0.5878223... 0.9183157...
Residual Model Experiments - Residual Model Super Regularized XGBoost (Sequential Split) - RecSys ...	12 days ago	0.7785546...	16.191871... 0.0730292... 0.5679953... 0.9184996...
Residual Model Experiments - Residual Model Super Regularized XGBoost (Sequential Split) - RecSys ...	12 days ago	0.6003485...	16.322396... 0.0705374... 0.5876392... 0.9188329...
Residual Model Experiments - Residual Model Super Regularized XGBoost (Sequential Split) - RecSys ...	14 days ago	0.710045...	17.277768... 0.0758028... 0.5266647... 0.7303197...
Residual Model Experiments - Residual Model Electic Net (Sequential Split) - RecSys Maag Pump_2023...	14 days ago	0.7643636...	16.042668... 0.071894... 0.4919275... 0.4169054...
Residual Model Experiments - Base Model with Oil Data (Only 2.5 months of data) - RecSys Maag Pump_2023...	14 days ago	0.6485594...	6.1581401... 0.0211587... -0.143076... 0.9036251...
Residual Model Experiments - Base Model (Sequential Split) - RecSys Maag Pump_2023-06-18 16:11:1...	14 days ago	0.7878269...	16.614176... 0.0720582... 0.5784213... 0.9178839...
Residual Model Experiments - Base Model (Sequential Split) - RecSys Maag Pump_2023-06-18 16:11:1...	14 days ago	-	- - - -
RecSys Maag Pump - Second stage XGBoost - individual Feature Feeders 2023-06-17 13:03:00	15 days ago	0.3197856...	17.491757... 0.0713392... -0.257479... 0.6667676...
RecSys Maag Pump - Smoothed With Oil Injector Interactions (Second stage XGBoost) - Random Split...	15 days ago	0.9017908...	9.6894083... 0.04050608... 0.7629988... 0.7516454...
RecSys Maag Pump - Smoothed With Oil Injector Interactions (Second stage XGBoost) - Random Split...	15 days ago	0.7701109...	13.304828... 0.0529093... 0.2419956... 0.8394351...
RecSys Maag Pump - Smoothed With Oil Injector Interactions (Second stage XGBoost) - 93% Train 20...	19 days ago	0.721496...	18.102869... 0.0768616... 0.5264083... 0.8524433...
RecSys Maag Pump - Smoothed With Oil Injector Interactions (Second stage XGBoost) - 2025-06-13 1...	19 days ago	0.7354972...	17.084797... 0.0720070... 0.4104948... 0.7256048...
RecSys Maag Pump - Smoothed With Oil Injector Interactions (Second stage XGBoost) - 2025-06-12 20:37:24	20 days ago	0.7644659...	15.292053... 0.0656511... 0.5494613... 0.9426673...
RecSys Maag Pump - Smoothed With Oil Injector With Flow 3 + 4 combined 2025-06-12 20:37:24	20 days ago	0.7386580...	15.400884... 0.0663984... 0.6050124... 0.9414796...
RecSys Maag Pump - Smoothed Without Oil Injector Reduced Lambda / Alpha Ranges AND Product N...	20 days ago	0.7272505...	15.504847... 0.0680123... 0.4891130... 0.8607370...
RecSys Maag Pump - Smoothed Without Oil Injector Reduced Lambda / Alpha Ranges AND Product N...	20 days ago	0.6175207...	11.581705... 5.0208329... -0.199724... 0.4271943...
RecSys Maag Pump - Smoothed Without Oil Injector Reduced Lambda / Alpha Ranges 2025-06-12 16:...	20 days ago	0.8040572...	13.242027... 0.0578847... 0.6217379... 0.8620314...
RecSys Maag Pump - Smoothed Without Oil Injector 2025-06-12 15:58:17	20 days ago	0.7690639...	14.875175... 0.0637761... 0.5633183... 0.9364719...
RecSys Directly on Viscosity - With Product Normalization 2025-06-04 19:36:25	28 days ago	0.2243850...	17.701099... 0.2150057... -0.998654... 0.7877930...
RecSys Directly on Viscosity - Train with same inputs as maag pump model 2025-06-03 16:40:01	29 days ago	0.9221443...	20.210768... 0.0772170... 0.7251845... 0.9423257...
RecSys Directly on Viscosity - Vanilla 2025-06-03 16:18:29	29 days ago	0.9060886...	15.986028... 0.0574579... 0.8332340... 0.9374779...
RecSys Directly on Viscosity - Including Oil Injection Locations - With Product Normalization 2025-06...	29 days ago	0.3534808...	19.507934... 1.6930235... -3.160008... 0.9460469...
RecSys Directly on Viscosity - Including Oil Injection Locations - Not Normalized by Product 2025-06...	29 days ago	0.4308196...	20.033238... 0.0715433... -3.001281... 0.9999516...
RecSys MAAG PUMP - Product Normalized without Feeder Mass Flows / Feed Factors 2025-05-27 15...	1 month ago	0.1317333...	16.313020... 3.1897048... -0.100727... 0.6620067...
RecSys MAAG PUMP - Random Split (Maag Pump Not Normalized by Product) 2025-05-23 18:39:37	1 month ago	0.4737003...	11.596941... 0.0261569... 0.3090825... 0.9384493...
RecSys MAAG PUMP - Normalized by Product (with Feeder Recipes) 2025-05-23 18:30:30	1 month ago	0.2398134...	9.5811807... 4.4464978... -0.218724... 0.9162473...
RecSys MAAG PUMP - Normalized by Product 2025-05-23 18:25:55	1 month ago	-	- - - -
RecSys MAAG PUMP - Normalized by Product 2025-05-22 19:09:25 2025-05-22 19:10:01	1 month ago	0.1908226...	17.461881... 3.7325110... -0.160212... 0.9777575...



# Architecture Overview

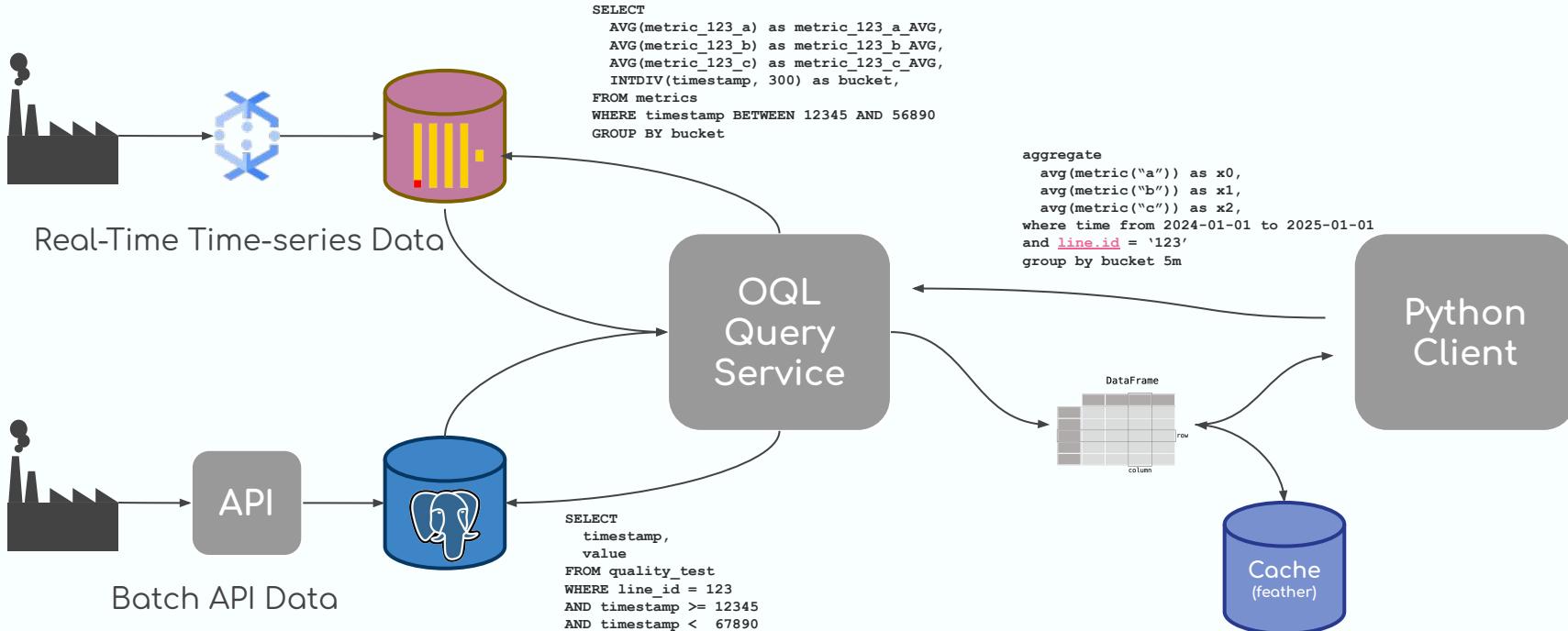
- Deploy Cron Job is scheduled using Airflow
- Models from MLFlow Model Registry are deployed onto MLServer
- Production Model Version information is stored in GCS



# Training our Models

With SKLearn and MLFlow

# Getting Data for Training



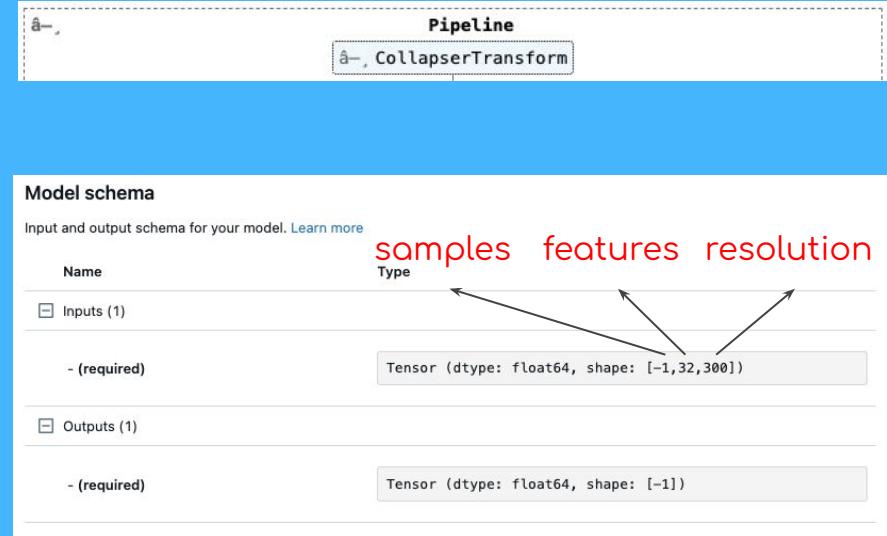
# Training the Model

- Data Scientists conduct EDA, feature engineering and builds an sklearn pipeline.



# Training the Model

- Data Scientists conduct EDA, feature engineering and builds an sklearn pipeline.
- The first layer of the pipeline is designed to support time shifted features from the low resolution data.



# Training the Model

- Data Scientists conduct EDA, feature engineering and builds an sklearn pipeline.
- The first layer of the pipeline is designed to support time shifted features from the low resolution data.
- All the model experiments are logged to MLFlow along with test statistics and supporting artifacts for peer review.

Overview Model metrics System metrics Traces Artifacts

model\_outputs/residual\_plot.png 81.45KB  
Path: gs://oden-production\_miflow\_artifact\_store/27/59cea15094fc4a0aa63cd51e72723edfa

Residuals vs. Predicted Values

Residuals

Predicted Value

In-Sample

Out-of-Sample

Distribution

Artifacts

- data\_split
- model
- model\_outputs
  - performance\_metrics
    - Test\_performance\_metrics.csv
    - Test\_product\_performance\_metrics.csv
    - Train\_performance\_metrics.csv
    - Train\_product\_performance\_metrics.csv
    - baseline\_performance\_metrics.csv
    - baseline\_product\_performance\_metrics.csv
    - pn\_Test\_product\_performance\_metrics.csv
    - pn\_Train\_product\_performance\_metrics.csv
    - pn\_baseline\_product\_performance\_metrics.csv
  - ale\_plots.png
  - all\_features\_ale\_plots.png
  - feature\_importances.png
  - in\_range\_table.csv
  - model\_predictions.csv
  - prediction\_plot\_insample.png
  - prediction\_plot\_outsample.png
  - residual\_plot.png
  - model\_params
    - cross\_val\_splits.json
    - train\_val\_test\_splits.json
    - dataflow\_config.json
    - inference\_metadata.json
    - pipeline\_vis.html
    - pq\_training.log

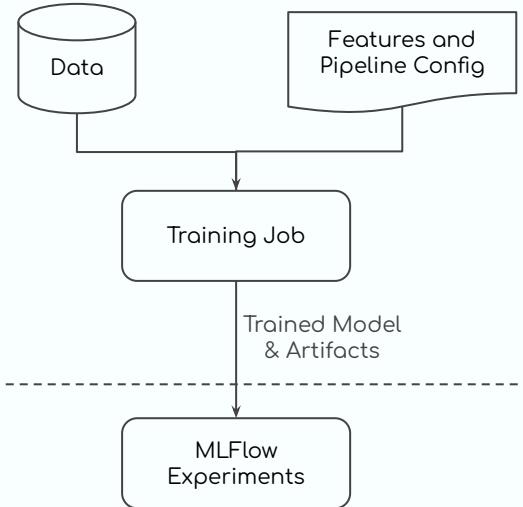
Metric	Value
train_corr	0.9185993286254339
test_corr	0.949061721727258
train_r2	0.8408351971582717
test_r2	0.8948399662709521



# Training the Model

- Data Scientists conduct EDA, feature engineering and builds an sklearn pipeline.
- The first layer of the pipeline is designed to support time shifted features from the low resolution data.
- All the model experiments are logged to MLFlow along with test statistics and supporting artifacts for peer review.
- Optionally, if the model needs to be retrained on a schedule with latest time series data, automated training is orchestrated via Airflow.

## Automated Training DAG (scheduled 1st of every month)



# Deploying our Models

w/ MLFlow, MLServer, and a GCS config

# Deploying our Models

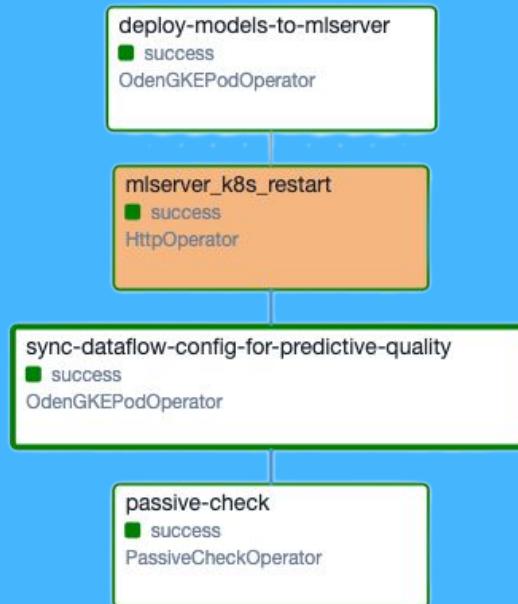
- Models in MLFlow Experiments are registered to MLFlow Model Registry along with versioning after review.

Version	Registered at
Version 17	05/20/2025, 08:42:22 AM
Version 16	05/19/2025, 03:08:29 PM
Version 15	04/22/2025, 02:09:37 PM
Version 14	02/14/2025, 11:52:12 AM
Version 13	12/06/2024, 02:09:22 PM
Version 12	12/06/2024, 11:48:07 AM
Version 11	10/19/2024, 12:54:02 PM
Version 10	10/16/2024, 11:27:55 AM
Version 9	09/25/2024, 09:55:41 AM
Version 8	09/09/2024, 02:51:15 PM



# Deploying our Models

- Models in MLFlow Experiments are registered to MLFlow Model Registry along with versioning after review.
- Deploy Job is runs every hour to ensure
  - All the versioned models in Model Registry are deployed to MLServer.
  - Production model version cache/config points to the latest version



# Deploying our Models

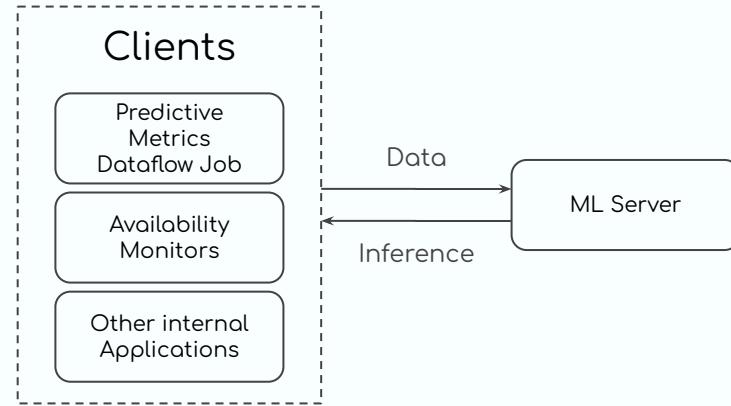
- Models in MLFlow Experiments are registered to MLFlow Model Registry along with versioning after review.
- Deploy Job is runs every hour to ensure
  - All the versioned models in Model Registry are deployed to MLServer.
  - Production model version cache/config points to the latest version
- The config contains three things
  - order of input features
  - resolution of the data expected by the model
  - Inference metric metadata

```
{  
    "input_metric_ids": [  
        "3ea26934-8464-54d1-86a6-70a5c7c9a5f3",  
        "99baf78a-2bdf-534f-90e5-b58fd852c2c5"  
    ],  
    "window_size_s": 300,  
    "step_size_s": 10,  
    "line_id": "613cbd00-1279-420e-b0c5-dc310b9978b9",  
    "model_identifier": "Monitoring-Monitoring-Factory-Clearblade-SLO-",  
    "model_version": null,  
    "output_metric_id": "1caeef70-43b1-4813-9698-c9462e2d1de3",  
    "output_device_id": "b9be0864-8a25-4260-bd84-0c90aac0c38a",  
    "output_machine_id": "602edb42-9d61-42c0-869c-3897b9d040c7",  
    "output_metric_name": "synthetic_predicted_metric"  
}
```



# Deploying our Models

- Models in MLFlow Experiments are registered to MLFlow Model Registry along with versioning after review.
- Deploy Job is runs every hour to ensure
  - All the versioned models in Model Registry are deployed to MLServer.
  - Production model version cache/config points to the latest version
- The config contains three things
  - order of input features
  - resolution of the data expected by the model
  - Inference metric metadata
- MLServer loads all the models into memory and serves inference requests from clients via GRPC

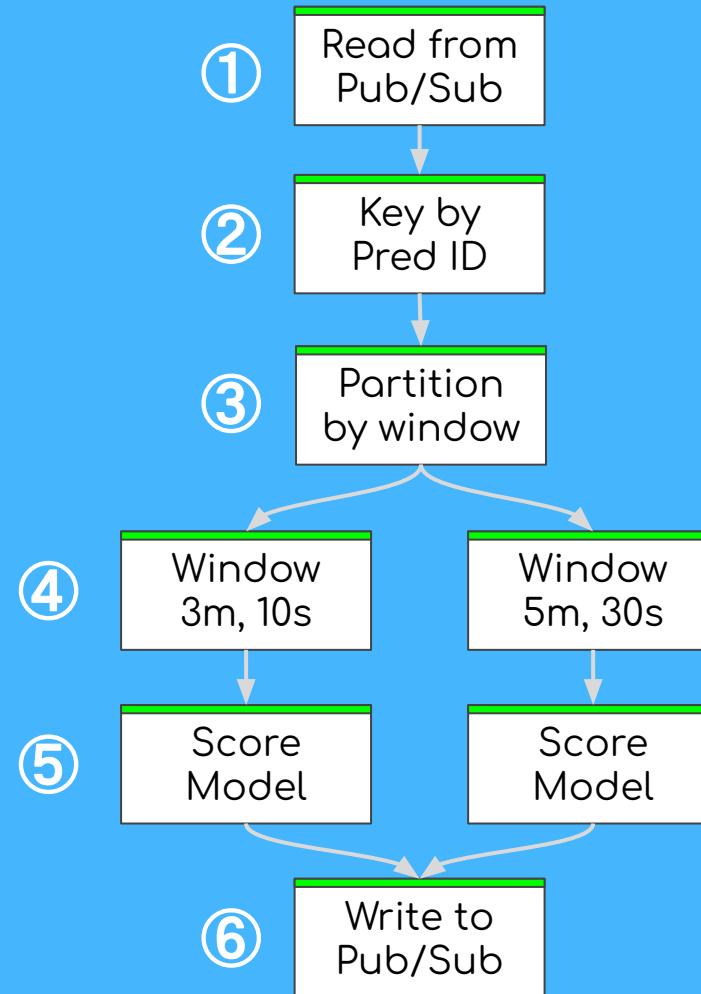


# Scoring out models

With Apache Beam and MLServer

# Predictive Metrics Beam Job

1. Read Metrics from Pub/Sub  
(using custom multi-source-reader)
2. Key metrics to Predictive Metric ID(s)  
they're components to
3. Partition key'd metrics into PCollections  
by windowed size+slide
4. Window by window size and slide
5. Form tensors, score model, and form new  
Metric object from score
6. Write new metrics to Pub/Sub  
(using custom multi-sink-writer)



# Predictive Metrics Beam Job

- Reading and writing to Pub/Sub is done using a multi-source reader and writer.
- This allows us to deploy this job in “batch mode” via Options.

```
public static class Read<OutputT>
  extends PTransform<PBegin, PCollection<OutputT>> {
  ...
  public Read(ReadOptions options, Class<OutputT> outputClass) {...}

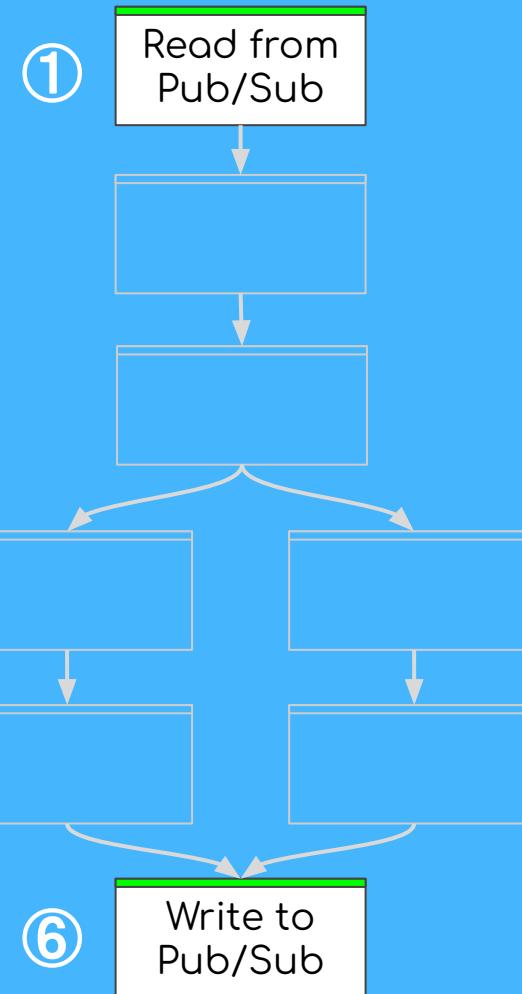
  public String getName() {
    return "Read " + outputClass.getSimpleName() + " from " + options.getReadMode();
  }
  ...

  public PCollection<OutputT> expand(PBegin input) {
    return switch (options.getReadMode()) {
      case "PUBSUB" -> expandPubsub(input);
      case "FILE" -> expandfile(input);
      case "BIGQUERY" -> expandBigQuery(input);
      default -> {
        throw new RuntimeException("Unknown mode: " + options.getReadMode());
      }
    };
  }
  ...

  public static class Write<InputT>
    extends PTransform<PCollection<InputT>, PDone> {
    ...
    public Write(WriteOptions options, Class<InputT> inputClass) {...}

    public String getName() {
      return "Write" + inputClass.getSimpleName() + " to " + options.getWriterMode();
    }
    ...

    public PDone expand(PCollection<AvroT> input) {
      return switch (options.getWriterMode()) {
        case "PUBSUB" -> expandPubsub(input);
        case "FILE" -> expandfile(input);
        case "FILE_WINDOWED" -> expandfileWindowed(input);
        case "LOG" -> expandLog(input);
        default -> {
          throw new RuntimeException("Unknown option: " + options.getWriterMode());
        }
      };
    }
  }
}
```



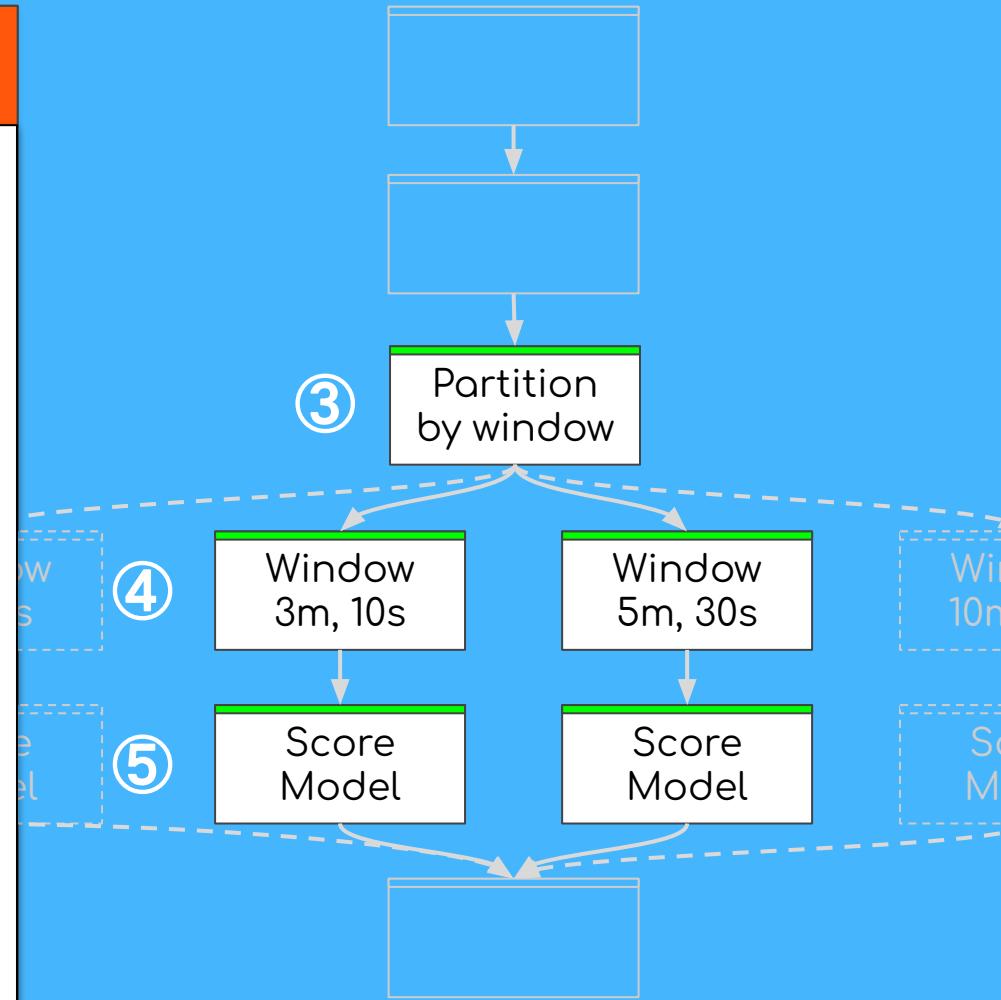
# Predictive Metrics Beam Job

- Because different models are built using different sized windows, we split the pipeline by window size.
- This means window size **must be known at DAG read time** (deploy time).
- Recombining the collections with different windows is a PITA so we run just as many scoring PTransforms.
- We just build the DAG in a for-loop

```
List<PCollection<Metric>> predictiveMetricsCollections = new ArrayList<>();
int counter = 0;
for (WindowSettings window : windows) {

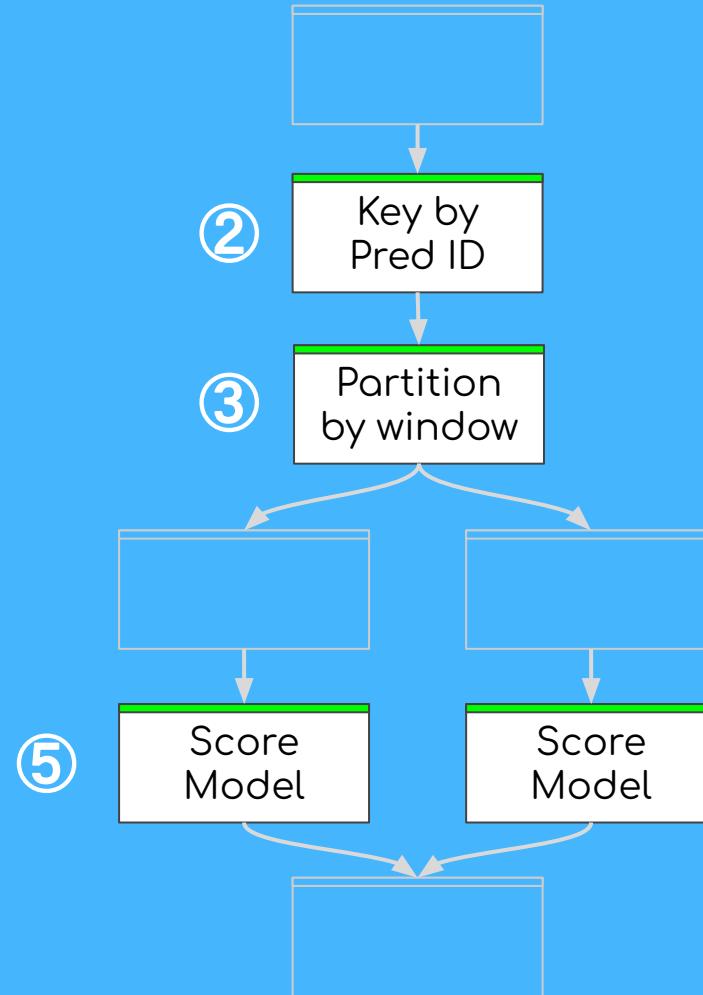
    TupleTag<KV<MetricKey, List<Metriquita>>> windowTag =
        allMatchedWindowTagsAsList.get(counter);
    PCollection<KV<MetricKey, List<Metriquita>>> windowedStream = splitStreams.get(windowTag);

    PCollection<Metric> predictiveMetrics =
```



# Predictive Metrics Beam Job

- Some steps require reading the deployed model config which is stored in GCS.
- In the past, we would:
  - Read the config on an interval using a GenerateSequence.
  - Collapse into a PCollectionView
  - Load into PTransforms as side input
- But this came with problems:
  - Cold-start issues
  - Strange PCollectionView errors
- Now our PTransforms:
  - Fetch the config from GCS when needed.
  - Cache in the PTransform w/ TTL



# Predictive Metrics Beam Job

- Some steps require reading the deployed model config which is stored in GCS.
- In the past, we would:
  - Read the config on an interval using a GenerateSequence.
  - Collapse into a PCollectionView
  - Load into PTransforms as side input
- But this came with problems:
  - Cold-start issues
  - Strange PCollectionView errors
- Now our PTransforms:
  - Fetch the config from GCS when needed.
  - Cache in the PTransform w/ TTL

```
/*
 * A base DoFn that encapsulates the logic for fetching configuration from GCS and refreshing
 * definitions.
 */
4 inheritors
public abstract static class ConfigDoFn<InputT, OutputT> extends DoFn<InputT, OutputT> {

    // The static Storage handle is shared among all subclasses
    private static final Storage storage = StorageOptions.getDefaultInstance().getService();

    protected final String bucketName;
    protected final String objectName;

    protected TimedGCSFetcher fetcher;
    protected PredictiveMetricDefinitions definitions;

    public ConfigDoFn(String bucketName, String objectName) {
        this.bucketName = bucketName;
        this.objectName = objectName;
    }

    public static void refreshDefinitions(
        TimedGCSFetcher fetcher, PredictiveMetricDefinitions definitions)
        throws MissingConfigurationException {
        fetcher.refresh();
        if (definitions.neverSucceeded()) {
            LOG.error("No predictive metric definitions present");
            throw new MissingConfigurationException("No predictive metric configuration present");
        }
    }

    @Override
    @Setup
    public void setup()
        throws MissingConfigurationException, TextFormat.ParseException, InvalidFormatException {
        this.definitions = new PredictiveMetricDefinitions();
        this.fetcher = new TimedGCSFetcher(storage, bucketName, objectName, this.definitions);
        refreshDefinitions(fetcher, definitions);
    }

    @StartBundle
    public void startBundle(StartBundleContext context) throws MissingConfigurationException {
        refreshDefinitions(fetcher, definitions);
    }
}
```

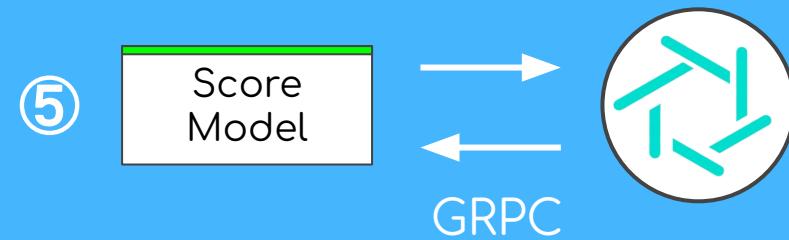


# Scoring w/ MLServer

- MLServer is an application for serving standard inference runtimes via REST and GRPC
- Serves models over the Open Inference Protocol standard for scoring
- Lets users serve multiple models at once (multi-modal serving)

## Why not Vertex?

- Vertex requires packaging each model in its own container meaning more isolation but more resources per model.
- At the time we chose MLServer, Vertex required one vCPU per model.



## Embedded Model Scoring

- Pro: Low latency, no external calls, easy to parallelize.
- Pro: Data Scientists will touch Python.
- Pro: Built-in RunInference transform.
- Con: In our experience, Python Beam streaming is less performant at windowing.
- Con: We have lots of homegrown code for writing Java Beam jobs.
- Considered: Multi-Language pipelines but we have no operational experience in these.

## External Scoring Service

- Pro: We get to use Java.
- Pro: It's easy to test and scale scoring our models from non-beam (APIs).
- Pro: We've decoupled model scoring dependencies from pybeam dependencies.
- Pro: All model scoring exists in only one place.
- Con: We risk being IO-bound.
- Con: Error tracking is more difficult.

# Scoring against MLServer

Scoring w/ MLServer is easy:

1. Sort our input values by their ID.

```
// Sort the metrics by their metricID
HashMap<String, List<Metriquita>> metricsByMetricId = new HashMap<>();
for (Metriquita metric : metrics) {
    if (!metricsByMetricId.containsKey(metric.getMetricId())) {
        metricsByMetricId.put(metric.getMetricId(), new ArrayList<>());
    }
    metricsByMetricId.get(metric.getMetricId()).add(metric);
}
```

2. Form our (2d) tensor:

```
// Create a tensor from the metrics (num_metrics, window_size)
double[][] tensor = new double[numExpectedInputMetrics][definition.getWindowSizeS()];
for (int i = 0; i < numExpectedInputMetrics; i++) {
    String metricId = definition.getInputMetricIds()[i];
    List<Metriquita> metricList = metricsByMetricId.get(metricId);
    for (int j = 0; j < definition.getWindowSizeS(); j++) {
        tensor[i][j] = metricList.get(j).getValue();
    }
}
```

3. And score via GRPC

```
ModelInferRequest request =
    ModelInferRequest.newBuilder()
        .setmodelName(definition.getModelIdentifier())
        .setModelVersion(Integer.toString(definition.getModelVersion()))
        .addInputs(
            ModelInferRequest.InferInputTensor.newBuilder()
                .setName("input-0")
                .setdatatype("FP64")
                .addAllShape(List.of(-1L, (long) tensor.length, (long) tensor[0].length))
                .setContents(
                    InferTensorContents.newBuilder()
                        .addAllFp64Contents(
                            Arrays.stream(tensor) Stream<double[]>
                                .flatMapToDouble(Arrays::stream) DoubleStream
                                .boxed() Stream<Double>
                                .collect(Collectors.toList())))
                .build())
            .build())
        .build();

ModelInferResponse resp;
int retries = 3;
StatusRuntimeException lastException = null;
while (retries > 0) {
    try {
        resp = this.client.modelInfer(request);
        return resp.getOutputs().index(0).getContents().getFp64Contents(index: 0);
    } catch (StatusRuntimeException e) {
        // If it's an issue with the request, don't retry.
        if (!GRPCErrorHandler.shouldRetryOnError(e)) return null;
        LOG.warn("Failed to score tensor, try {} of {}", RETRIES - retries + 1, RETRIES, e);
        lastException = e;
        retries--;
        try {
            Thread.sleep((long) (SLEEP_MS * Math.pow(RETRIES - retries, 2)));
        } catch (InterruptedException ex) {
            Thread.currentThread().interrupt();
        }
    }
}
throw lastException;
```



# After v1

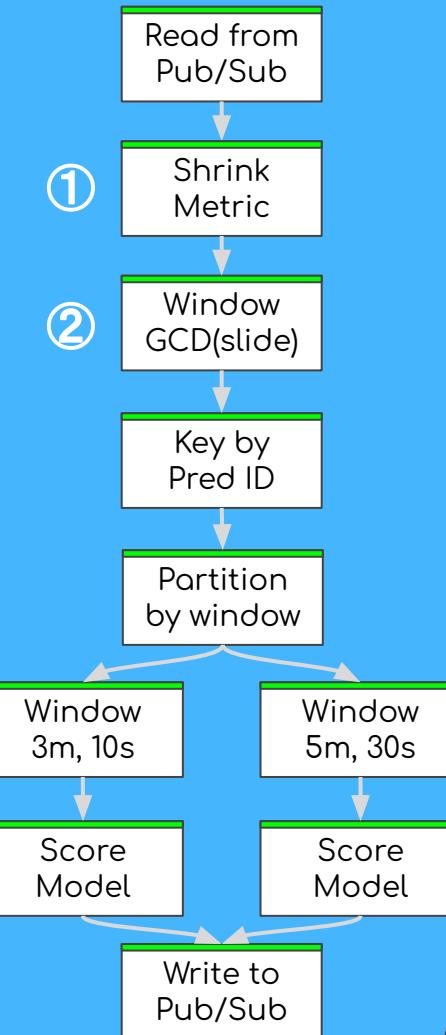
Some interesting challenges along the way.

# Streaming Engine Optimization

Due to the high dimensionality of the windowed join ( $\text{num\_inputs} * \text{window\_size} / \text{window\_slide}$ )  
Streaming Engine was the largest cost driver of our models making them unprofitable for contracts (\$1,300 to 1,800 per model per year).

To solve this, we added two steps:

1. Shrink the (serialized) metric as much as possible.
2. Window in two-stages.



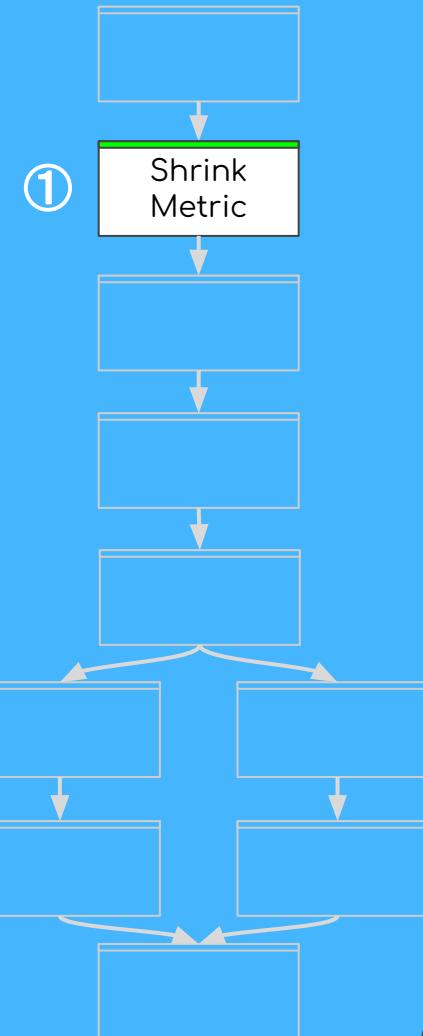
# Streaming Engine Optimization

Previously, our Metric class:

- Irrelevant UUIDs that were stored as 36-char strings.
- A legacy “name” identifier.
- Serialized using SchemaCoder (which is way better than Serializable!)

We introduced a new smaller Metric class (Metriquita) which:

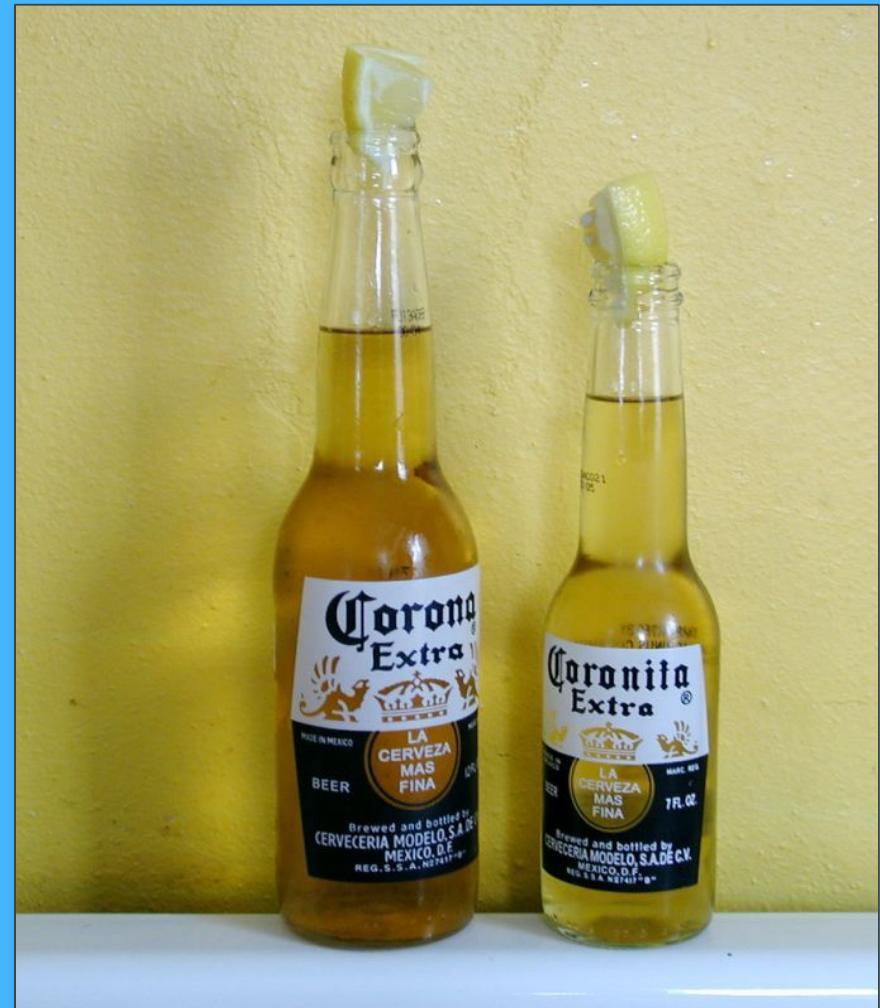
- Dropped everything but the metric UUID, value, and timestamp.
- Used a CustomCoder to tightly pack the UUIDs as two longs.



# Streaming Engine Optimization

```
public class MetriquitaCoder extends CustomCoder<Metriquita> {  
    private static final MetriquitaCoder INSTANCE = new MetriquitaCoder();  
    private MetriquitaCoder() {}  
    public static MetriquitaCoder of() { return INSTANCE; }  
  
    @Override  
    public void encode(Metriquita value, OutputStream outStream) throws IOException {  
        DataOutputStream dataOut = new DataOutputStream(outStream);  
        dataOut.writeLong(value.metricIdMostSigBits);  
        dataOut.writeLong(value.metricIdLeastSigBits);  
        dataOut.writeLong(value.timestampMs);  
        dataOut.writeDouble(value.value);  
    }  
  
    @Override  
    public Metriquita decode(InputStream inStream) throws IOException {  
        DataInputStream dataIn = new DataInputStream(inStream);  
        Metriquita record = new Metriquita();  
        record.metricIdMostSigBits = dataIn.readLong();  
        record.metricIdLeastSigBits = dataIn.readLong();  
        record.timestampMs = dataIn.readLong();  
        record.value = dataIn.readDouble();  
        return record;  
    }  
}
```

SchemaCoder is: 138  
SerializableCoder is: 334  
SnappyCoder is: 144  
MetriquitaCoder is: 32



# Streaming Engine Optimization

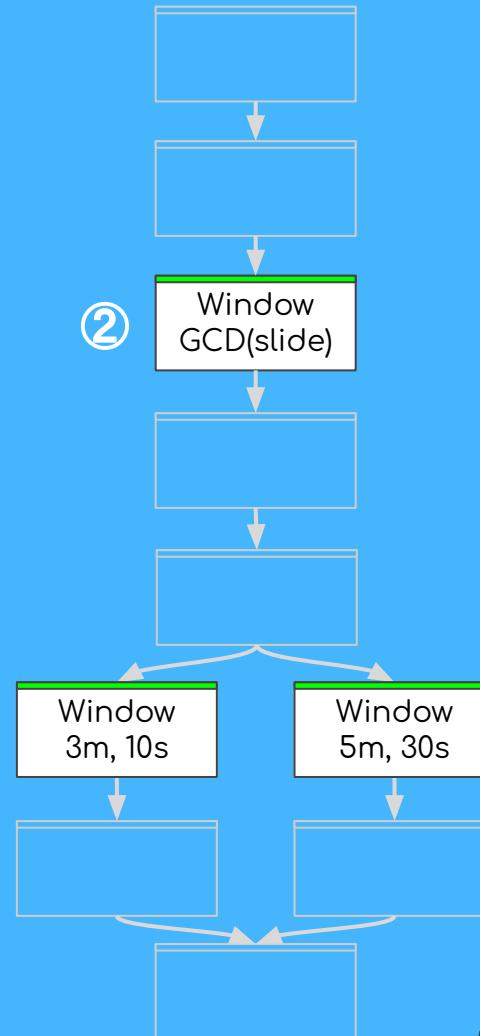
Metriquita reduced each element by a constant 106b. But we still had to account for:

- Pairing a key to each element (+38b)
- Adding the window to each pair (+300b)

The clear target was to reduce the number of elements joined in a window once and, unintuitively, this was accomplished by windowing twice.

Window 1: Pre-aggregate each metric into a list of metrics (GCD of all possible window slides)

Window 2: Normal windowing but now with small list-chunks of input metrics.



```
seconds_in_a_day / window_step_size_s * num_input_metrics * window_size_s * 384b
```

$$\text{seconds\_in\_a\_day} / \text{window\_step\_size\_s} * \text{num\_input\_metrics} * \text{window\_size\_s} * 384b$$

The formula is broken down into three main components:

- Joins per day**:  $\text{seconds\_in\_a\_day} / \text{window\_step\_size\_s}$
- Elements to join**:  $\text{num\_input\_metrics} * \text{window\_size\_s}$
- Element size**:  $384b$

```
seconds_in_a_day / window_step_size_s * num_input_metrics * 384b  
+  
seconds_in_a_day / window_step_size_s * window_size_s / gcd_slides * num_input_metrics * 658b
```

$$\text{seconds\_in\_a\_day} / \text{window\_step\_size\_s} * \text{num\_input\_metrics} * 384b$$

Joins per day (first)

$$+ \text{seconds\_in\_a\_day} / \text{window\_step\_size\_s} * \text{window\_size\_s} / \text{gcd\_slides} * \text{num\_input\_metrics} * 658b$$

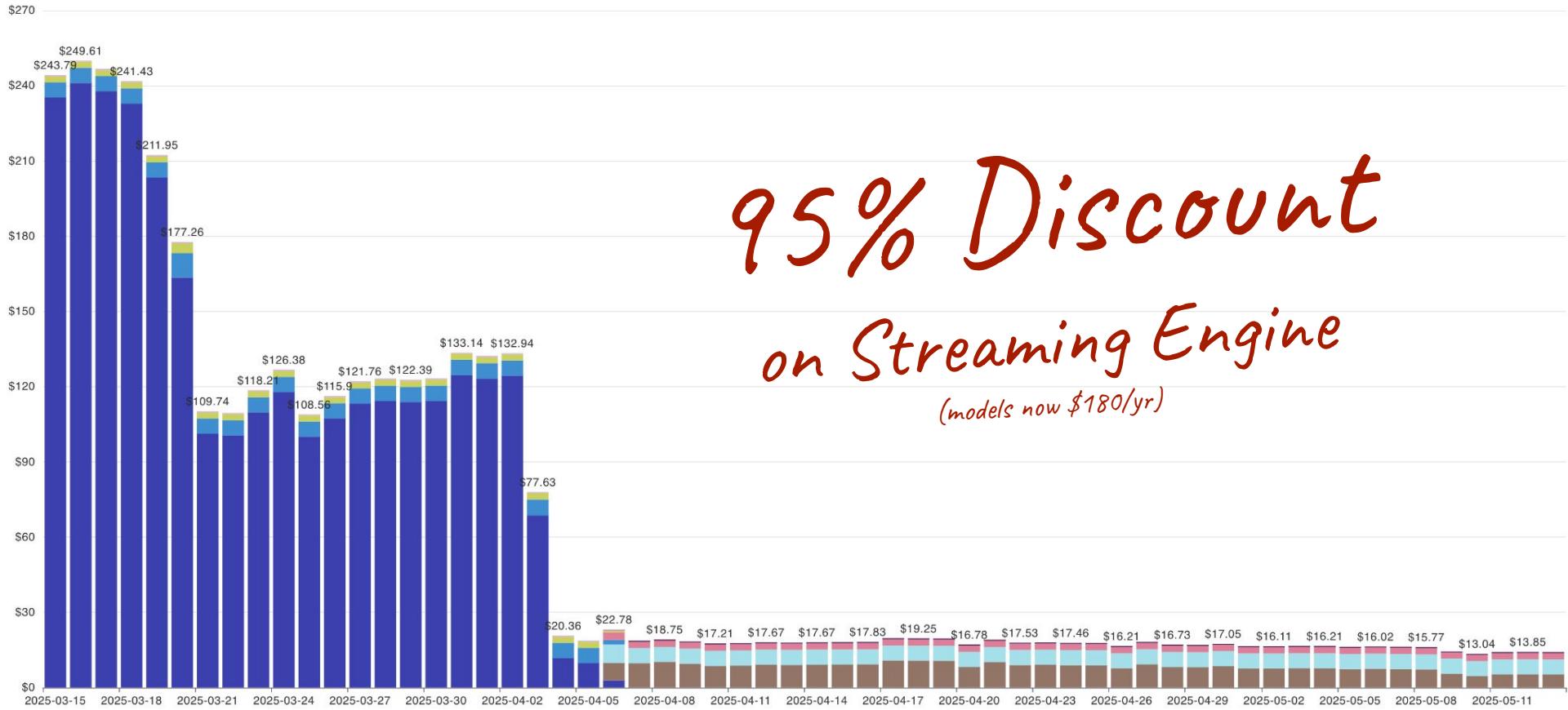
Joins per day (second)

Elements to join (first)

Element Size (first)

Lists to join (second)

List Size (second)



# 95% Discount on Streaming Engine (models now \$180/yr)

(models now \$180/yr)

- Cloud Dataflow;Streaming data processed for Iowa Cloud Dataflow;vCPU Time Streaming South Carolina Cloud Dataflow;RAM Time Streaming South Carolina Cloud Dataflow;Local Disk Time PD Standard South Carolina  
Cloud Dataflow;Streaming data processed for South Carolina Cloud Dataflow;vCPU Time Streaming Iowa Cloud Dataflow;RAM Time Streaming Iowa Cloud Dataflow;Local Disk Time PD Standard Iowa

## Shared Inference Resources

- At Oden, we have a large number of tiny models. Resource sharing by models is crucial for cost scaling reasons.
  - 4 cpu cores and 4 gigs of memory
  - 150+ production sklearn pipelines
  - 350 req/minute with <200ms latency



## Shared Inference Resources

- At Oden, we have a large number of tiny models. Resource sharing by models is crucial for cost scaling reasons.
  - 4 cpu cores and 4 gigs of memory
  - 150+ production sklearn pipelines
  - 350 req/minute with <200ms latency
- But we are restricted to a single python runtime!!



# Shared Inference Resources

- At Oden, we have a large number of tiny models. Resource sharing by models is crucial for cost scaling reasons.
  - 4 cpu cores and 4 gigs of memory
  - 150+ production sklearn pipelines
  - 350 req/minute with <200ms latency
- But we are restricted to a single python runtime!!
  - Changes in code for new models may break already existing models
  - Python upgrade needed careful planning and gymnastics
    - Perform a surgery OR
    - Retrain models in new runtime

## Python 3.12.9 Upgrade Postmortem

Owned by Devon Petricolas ...  
Last updated: May 05, 2025 • 6 min read • 15 people viewed

### High-Level Summary

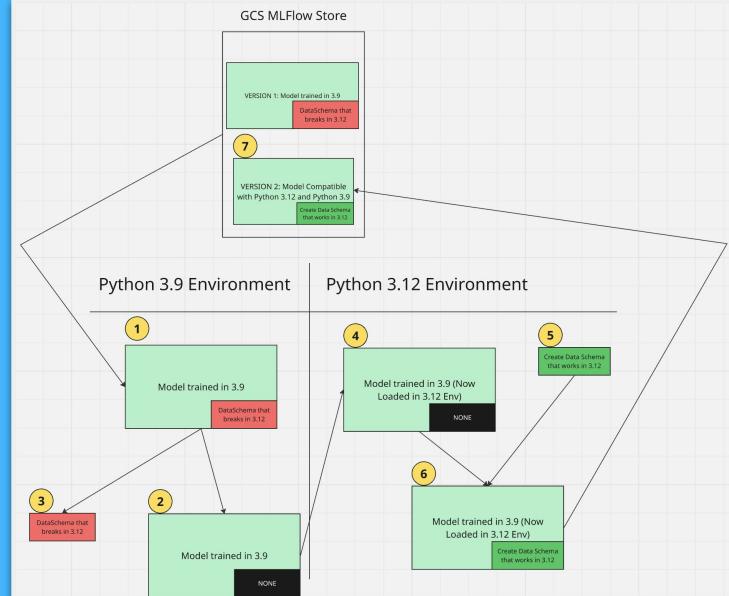
On April 22nd, the Data Science team upgraded all science-repo Python services from 3.9.21 to 3.12.9 to unblock the Copilot Squad's MCP Server work. Due to the way Predictive Quality and Recsys models are deployed, this resulted in a 35-minute total outage in Predictive Quality and a 26-minute partial outage in Recsys. As of April 22nd, no other issues have been identified.

### Timeline

#### Lead-Up

1. There is a large amount of unresolved work on deploying major dependency changes to existing models hosted in MLServer. This issue has been identified as the cause of [two Incidents](#) and was one of the driving, but ultimately unresolved, issues identified in our [2024 Q4 Code Yellow](#). As of Q2 2025, we believe that the [beginnings of a solution](#) are evident, but the work has not been prioritized.

2. The Data Science team worked on updating Python commands to build and run MLServer to handle



# In Conclusion

# Takeaways

- Streaming Beam works well forming and scoring windows of data!
  - We needed to pay close attention to Streaming Engine costs on Dataflow.
  - It's worth testing your encoders!
  - Syncing MLServer and Dataflow via a simple JSON config has been easier than anticipated!
- Using an external service for scoring was a good call!
  - IO was never an issue.
  - Opened up non-beam inference capabilities.
- We're still struggling to balance cost vs runtime.
  - A single inference server and runtime has saved us money.
  - Shared dependencies makes model deployment stressful.
- MLFlow and MLServer have allowed easy experimentation and deployment by Data Scientists.



# Where are we going?

- Many models perform well in real-time!  
95% correlation, >60% R<sup>2</sup>
  - Some offline quality tests are harder to model than others.
  - Large inconsistencies in the models between products being manufactured.
- 
- We are seeing success embedding our quality predictions in bigger systems.
  - Now that we've built this infrastructure we can explore other predictive models such as predicting future in-line metrics.



Devon and Jeswanth

# QUESTIONS?

## Subtitle 1

Some text

## Subtitle 2

Other text

# Text and vertical image

The image shown is just a sample, replace with your own.



Lorem ipsum dolor sit amet, ignota utamur hendrerit ne his, modo signiferumque nam id. Eos ne choro virtute posidonium. Eu everti philosophia per. Legere theophrastus eu vim, sit decore omittam definiebas ne. Id diam dolorum definiebas vix, dignissim pertinacia adolescens eu vim.

# Text with horizontal image



NAMES

# QUESTIONS?

Maybe some contact info here?

Twitter

Linkedin

Github

Or whatever you want