Mímir: Building and Deploying an ML Framework for Industrial IoT

Devon Peticolas
Oden Technologies
NY, USA
devon@oden.io

Russell Kirmayer
Oden Technologies
NY, USA
russell.kirmayer@oden.io

Deepak S. Turaga
Oden Technologies
NY, USA
deepak.turaga@oden.io

***Abstract*—In this paper we describe Mímir, a production grade cloud and edge spanning ML framework for Industrial IoT applications. We first describe our infrastructure for optimized capture, streaming and multi-resolution storage of manufacturing data and its context. We then describe our workflow for scalable ML model training, validation, and deployment that leverages a manufacturing taxonomy and parameterized ML pipelines to determine the best metrics, hyper-parameters and models to use for a given task. We also discuss our design decisions on model deployment for real-time and batch data in the cloud and at the edge. Finally, we describe the use of the framework in building and deploying an application for Predictive Quality monitoring during a Plastics Extrusion manufacturing process.**

***Index Terms*—Industrial IoT, ML Framework, Predictive Quality, Process Control, Plastics Extrusion**

I. Introduction

There have been three major industrial revolutions so far, starting with the adoption of mechanization, setting up of the assembly line, and introduction of electronics with Programmable Logic Controllers (PLCs) into the manufacturing environment. Each of these has led to significant improvements in manufacturing productivity, but has also come with major socio-economic-environmental upheavals and challenges. We are now approaching the fourth industrial revolution, often referred to as Industry 4.0, due to a confluence of several different technologies, including the Internet of Things (IoT), cloud computing, and AI. IoT has brought us cheap sensing and connectivity, allowed us to collect data reliably and in real-time from these environments. Cloud computing has allowed us to scalably and robustly process and store such data. Finally, advances in machine learning and other fields of AI, are now allowing us to learn about and analyze the data (structured, unstructured, and other), reason about its context and plan sequences of actions to further optimize the manufacturing processes.

Manufacturing involves orchestrating fairly complex physical and chemical processes over multiple steps using various types of equipment and material. Consider as one example an Extrusion process that is used within plastics based manufacturing to create a range of products such as pipe/tubing, fencing, window frames, plastic film, wire insulation etc. During this process plastic pellets are ingested, melted through a combination of friction and heaters, forced through an appropriately shaped die, and then extruded as a part, or as a coating on other material. There are several different factors that contribute to the success of this process. These include *Extruder Control Settings* including the rate of ingestion, the RPMs of the rotating screw, the temperature bands of the heaters, the shape of the die, and the rate at which material is pulled through, *Material Properties*, *Environmental Aspects* such as ambient humidity, temperature, and finally *Human Factors* driven by operators, process engineers, and plant managers.

In order to optimize and control this process, we need to be able to continuously observe these different factors, understand their interactions given the physical context of the process, identify their impact on the eventual quality and throughput, and then automatically adapt them as required. This is complicated by several factors in the manufacturing environment, including the lack of observability of certain parts of the process, fragmented systems that collect different subsets of the data, unreliable connectivity, and the different time granularities and delays associated with gathering information from equipment, people, environment, and material. There is also an associated problem of the lack of wide adoption of standards and conventions for naming, data formats and transport protocols, all of which need to be overcome to extract meaningful insights. In this setting, we need to tackle at least three different classes of problems, *Diagnostic*, *Predictive* and *Prescriptive* or *Control*, which require a combination of long-term historical analysis, real-time online prediction, and optimization strategies. In order to address these requirements, we have built **Mímir**, a machine learning and data science infrastructure that spans the cloud and the edge, is scalable, extensible and robust, handles all kinds of data and its variations in structure, format and naming, includes state of the art ML algorithms tailored for manufacturing problems, and allows both experimentation, as well as rapid productionization of the results of the experimentation. In this paper we describe this framework in detail, including its core components (services and open source tools), our manufacturing specific extensions, and our design choices that allow us to deploy data science and ML applications in the industrial setting. We also highlight the use of the framework to realize an application for predictive quality monitoring of a plastics extrusion process.

The rest of this paper is organized as follows. We start with some related work in Section II, and describe the Mímir architecture and individual components in III. In Section IV we describe an application for predictive quality built on Mímir. We provide experimental results on scaling and robustness in Section V, and conclude in Section VI.

II. Related Work

While several different IoT systems are beginning to leverage ML and advanced analytics, there is little published literature on the ML frameworks being developed. A good survey of ML for sensors and IoT applications is included in [13]. In addition there are several infrastructure, communication, and computation platforms that allow the design, development and deployment of IoT applications. These include commercial systems such as those provided by the different cloud vendors [6], [3], [4], [7], [8] and some of these also discuss applications to manufacturing [4], [7]. There are also systems built by academic and open source communities [11], [14]. However, these systems describe their capability as a collection of discrete services that can be used to construct IoT applications, and do not include technical details of how an end to end IoT ML based solution can be realized, the appropriate design choices, and an actual instantiation. Many of these are also not specific to manufacturing and industrial use cases. There is related work on constructing large-scale ML frameworks, e.g. the one from Facebook [9], but these are targeted more to social media, and unstructured data analysis rather than the typical timeseries process and context data available in the manufacturing environment. Finally, there have recently been publications on the use of ML for IoT both at the edge [12] as well as in the cloud [10], but these need to be connected and combined to build an end to end IoT system. In this paper, we describe our framework for an end to end Industrial IoT solution.

III. System Overview

We show the overall system overview in Figure 1. Our system consists of edge devices and a cloud infrastructure for end to end data capture and analysis. At the edge, we use custom ARM based devices that communicate directly with manufacturing systems, machines and other sensors using multiple interfaces including PLCs, OPC UA, serial and ethernet-based communication, or custom communication protocols. These devices periodically capture the metrics and then clean, package, and format them into messages, and stream these messages to the cloud. The devices also support additional computation, such as ML model scoring on them. These devices also capture and stream asynchronous control events that provide context (e.g. state change, product change, downtime, maintenance) for the metrics.

Data is streamed continuously using MQTT (a low-weight communication protocol) to the cloud, where it is both analyzed as well as stored. In the cloud, data is distributed to the different analysis services using PubSub - a highly scalable message queue. Live data is processed with real-time analysis tools that perform various statistical and aggregation operations, and also score pre-trained ML models to generate

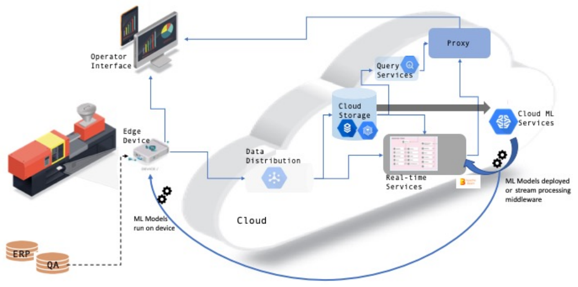

Fig. 1. End to End Data Collection Infrastructure

new insights that are displayed live on an operator interface. Additionally, all data is stored in large-scale storage that is optimized for efficient access to any desired interval of time. The stored data is used for batch and offline analysis, including generation of performance reports (how was performance in the last week/month), extraction of patterns of behavior leading to good and bad performance, as well as to train ML models. Within the Mímir framework we build tools that allow easy experimentation and exploration of such time-series manufacturing data with various ML workflows.

We build the Mímir framework on top of multiple Google Cloud Platform (GCP) [5] managed services as well as open source tools that we extend and optimize for edge for our IoT manufacturing use case. The main services we use to build our framework include *Cloud PubSub*, *Google Cloud Storage (GCS)*, *BigQuery*, *PostgreSQL*, and *Google Cloud ML*. In addition, we use *Apache Beam* [2] for real-time stream processing and *Apache Airflow* [1] for workflow orchestration. We now describe the Mímir framework in more detail. We start with a description of our storage layer and how we optimize it for multi-resolution time-series analysis. We then describe how we realize our ML workflows using a set of composable atomic steps, and our training and validation workflow that allows us to create, update and manage ML models. Finally, we describe how we operationalize and deploy ML models against streaming data either in the cloud or at the edge.

A. *Streaming Data Handling*

Our data stream processing infrastructure leverages the Apache Beam stream processing framework[1]. This framework includes a programming model for writing stream processing applications using multiple languages (Java, Python, Scala), a set of tools that allow us to perform common stream relational processing, and a set of distributed runtimes that allow us to deploy these jobs and scale them over a distributed compute infrastructure for production deployments. The set of tools include:

- Data connectors: including connectors to message queues (e.g. MQTT), databases (e.g. BigQuery), filesystems and object stores (e.g. GCS), as well as REST interfaces.

[1]Beam also supports batch processing, discussed in Section III-C

- Windowing: ability to set event-time, wall-clock-time, and interval based windows that handle out of order data arrival
- Relational Processing: ability to Join, GroupBy, Aggregate, Filter, Transform
- Extensible Processing: ability to support custom user code for additional stream processing

Beam is supported by several different runtime middlewares that allow us to deploy a beam job and manage it, scale it and provide fault tolerance. We use the Google Dataflow Runner for running Beam jobs on the cloud, and use the Apache Flink Runner to run Beam jobs at the edge. The same Beam code can then be run across cloud and edge through only configuration changes. These jobs handle our streaming metrics and perform several tasks, including ingesting them and formatting them for storage, performing additional calculations on them, windowing them to extract different types of features, and applying trained ML models for real-time predictions.

B. Data Storage Optimization

Our storage layer is built on top of Google Cloud Storage, BigQuery, and PostgreSQL. The core of the Mímir metrics data infrastructure is structured columnar data that we store in files on Google Cloud Storage or within the BigQuery database. Since GCS buckets can be viewed within BigQuery using External Tables, this provides us an interchangeable efficient store and query layer, that allows us to access large volumes of data using an SQL like query language. Additionally, we store asynchronous events and customer configuration within a PostgreSQL instance. This non-metric data is made available in BigQuery by regularly copying snapshots of the PostgreSQL tables into GCS and creating an external table BigQuery table that points to the file's bucket. This allow efficient joins to provide contextualized snapshots for analysis.

Streaming metric data is pushed into BigQuery using a Beam job (Section III-A) continuously. We employ table-level time-partitioning in BigQuery, i.e. splitting our data into separate tables by day based on ingestion time to provide efficient range queries. Our day-partitioned set of tables are named `metrics_it` (metrics ingestion time). A common characteristic of IoT manufacturing is the lack of reliable connectivity between the factory and the cloud, and this causes data to arrive late, in a bursty manner or potentially out of order. Additionally, powering down machines can mean that data remains buffered at the edge before it can be transmitted to the cloud for long periods of time[2]. Ingesting metrics into an ingestion time partitioned table allows us to reliably measure the true delay between the actual event time and the ingestion time. For analysis tasks, we rely on the event timestamps, hence, we also create a day partitioned set of tables named `metrics_et` that are partitioned based on the event time. We run an ETL job in BigQuery that copies over metrics from `metrics_it` to the appropriate `metrics_et` table, and during this process, we enforce SLAs on data lateness, discarding data that has arrived too late after the event. The `metrics_et` table allows us to efficiently select windows of data needed for analysis.

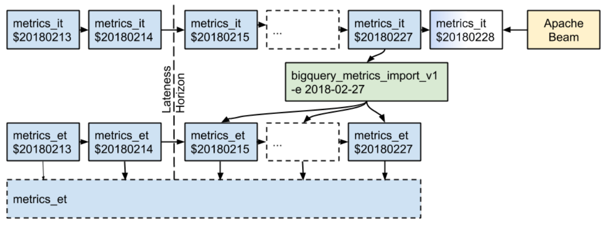

Fig. 2. `metrics-bq` BigQuery Metrics Tables

Stored metric data includes a key (the machine UUID and metric name), an event timestamp, and a value. In addition, we separately store a mapping of metric keys to labels that describe the role of each metric within the process, its relative importance, and additional units and conversion formulae (e.g. Celsius to Fahrenheit). The labels are derived from a specialized taxonomy for manufacturing process metrics that we have developed through interactions with domain experts, and allows us to utilize metrics consistently for analysis across lines and potentially factories even if the metric keys or measurement units are different. Context for metrics includes metadata as several types of intervals such as runs (type and batch of product being produced), states (whether the line is actively running, or idling) shifts, work orders etc. Each of these context intervals have a time-and-location, such as the work order number on a particular production line with a given start and end timestamp. These intervals are also available to BigQuery as external tables exported from our PostgreSQL metadata store. In order to contextualize our metrics with these intervals, we perform several window joins in BigQuery. For example, to know which `work_orders` had the most variation in their `line_speed` (a metric measuring rate of production), we need to write an SQL window join between `workorder` and `metric` tables. Window joins are cross-joins, therefore the runtime complexity is $O(m \times w)$ where m is the number of metric values and w is the number of intervals of type work order. Metrics are measured every second, while events occur at the order of hours, i.e. $m << w$. Often, metric variation within a minute is not relevant for analysis, and hence we create rollup views of the metrics as associative aggregates[3] and align both the metrics and the intervals to minute boundaries. The rollup values we calculate are sum, count, max, min, and sum2 ($\sum x^2$). These aggregates can further be incrementally aggregated to a lower resolution (e.g. 10 minutes) as needed. This allows us to provide multi-time-resolution views on our data, and through effectively pre-computed caches, we can tradeoff this resolution with the join and compute cost. This can improve the performance of our windowed join by a constant factor of 60x ($\hat{m} = m/60$).

[2]in the worst case, we have seen data arrive over 14 days late, since some machines were turned off over the winter break.

[3]There exists some aggregate function f in SQL that satisfies the associative law for the values in those rollups.

These window joins are performed for each of the k different types of context intervals ($O(m \times w_1 \times ... \times w_k)$). In order to further reduce join complexity, we pre-join all of our context intervals together to create sub-intervals as intersections of the start and end times for each of these intervals, i.e. each sub-interval is the longest continuous period in time where every context value is the same. We also quantize these sub-intervals into fixed-width time-slices, e.g.1 minute, 5 minutes, 1 hour, 1 day, for another multi-resolution view, and also effective joins against the metric rollups views. The resulting views can be thought of as a rollup-of-rollups. It is very important to realize that these views are materialized when needed, for instance by a particular analysis task, and hence do not require additional storage. In summary, to optimize our metric and context storage, we perform:

- Creating metric aggregates at multiple resolutions quantized to 60s boundaries
- Joining all types of intervals into a normalized sub-intervals view.
- Quantizing sub-intervals into 60s boundaries.
- Creating multiple views that joining rollups against sub-intervals at different resolutions.

Complexity gains achieved by this type of multi-resolution time-series storage optimization are discussed further in Section V.

C. Composable ML Workflows

In order to facilitate the creation and orchestration of our machine learning workflows, we decompose them into a set of atomic steps each of which can be realized differently based on whether we are running the step on streaming data or batch data, and whether we are deploying the step at the edge or in the cloud. Within the Mímir framework, we consider the following six steps:

- Metric Packing: Collecting metrics and context for a given problem into metric packs
- Labeling: Assigning a label to each metric pack
- Feature Extraction: Extracting features from a metric pack to create training data
- Model Training: Building a model on the training data
- Model Validation: Validating model against held out data
- Model Scoring: Applying model to new data

An ML workflow will then include a sequence of one or more of these steps. Consider the problem of training a model for quality prediction, as described later in Section IV. Our workflow in this case comprises of Metric Packing $\rightarrow$ Labeling $\rightarrow$ Feature Extraction $\rightarrow$ Model Training $\rightarrow$ Model Validation. Here, during Metric Packing we need to collect all metrics for an extrusion line for a 5 minute interval. For Labeling, we need to use the value of one metric from 5 minutes after the end of the window as the label. During Feature Extraction, we extract a set of features from the metric packs (e.g. using statistical aggregations such as moments etc.). During Model Training, we then train a model using the available set of parameterized models. This entire workflow is orchestrated as described in Section III-E.

Note that some of these steps, such as Metric Packing, can be realized using both stream processing or batch processing whereas others, such as Labeling, are typically realized using batch processing. Similarly, if we want to score a trained model for quality prediction on new data, we have a workflow that includes Metric Packing $\rightarrow$ Feature Extraction $\rightarrow$ Model Scoring, and in order to realize the prediction in real-time all of these steps need to be deployed as stream processing jobs.

We also use this high level abstraction to include both jobs that run in the cloud and jobs that run at the edge. This means that a workflow that requires model scoring on streaming data at the edge will deploy to our edge stream processing system the necessary steps. We orchestrate our workflow instances appropriately to make sure that we instantiate each step using the appropriate runtime (streaming or batch) and at the appropriate location (edge versus cloud)

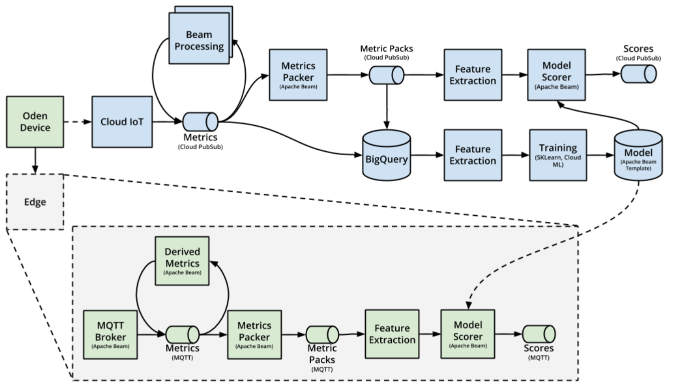
Beam Processing
Metrics Packer (Apache Beam)
Metric Packs (Cloud PubSub)
Feature Extraction
Model Scorer (Apache Beam)
Scores (Cloud PubSub)
Oden Device
Cloud IoT
Metrics (Cloud PubSub)
BigQuery
Feature Extraction
Training (SKLearn, Cloud ML)
Model (Apache Beam Template)
Edge
Derived Metrics (Apache Beam)
MQTT Broker (Apache Beam)
Metrics (MQTT)
Metrics Packer (Apache Beam)
Metric Packs (MQTT)
Feature Extraction
Model Scorer (Apache Beam)
Scores (MQTT)

Fig. 3. `pipeline` Cloud and Edge Workflows

An example with the model training and model scoring workflows is shown in Figure 3. Streaming data that is received from the factory floor is processed by the Metric Packer, Labeler and Feature Extractor in the cloud to create stored training data. The Model Training then uses this stored data to train a model that is validated against unseen data using the Model Validator. Once the model is validated and ready for deployment, we instantiate a Metric Packer, Feature Extractor, and Model Scorer at the edge to score the model against new streaming data.

D. Model Training and Validation

Once labeled training examples are available in cloud storage, we trigger model training using an Airflow job. The trigger is currently set to run periodically, based on our estimate of the variability of the underlying temporal data characteristics. We use two different types of models for our training - traditional ML models from `scikit-learn` and deep learning models using `Keras and Tensorflow`. We perform our training steps using Google Cloud ML. For the `scikit-learn` models, we create templates for each model, data transformation as well as feature selection algorithm with a list of possible ranges for the different tunable parameters. We show one example of a template below:

```
def get_regression_model_parameters(model_tag):
  if model_tag=='lasso':
    a = np.linspace(start=0.00001, stop=0.001, num=50)
    res = {model_tag+'__': a}
    reg = Lasso()
    return (reg,res)}
```

Each template is associated with a tag - a text string that acts as a label. We then compose instances of these templates into `scikit-learn pipelines` during the training process – where we use the tags to identify instances. As one example, we use the tags `lasso` and `chi2` to compose the regression model and the Chi-squared feature selector into a pipeline as shown below.

```
(fs,fs_params) = get_regression_fs_parameters('chi2')
(reg,reg_params) = get_regression_model_parameters('lasso')
pipeline_options.append(('chi2',fs))
pipeline_options.append(('lasso',reg))
pipeline_instance = Pipeline(pipeline_options)
```

Once pipelines are constructed we use hyper-parameter search built into `scikit-learn` along with cross-validation to determine the best performing set of hyper-parameters for each pipeline. The best hyper-parameter values along with the best achieved cross-validation performance is stored for each pipeline instance. We select the best pipeline using a combination of measured performance and pipeline scoring complexity – typically lower complexity pipelines that provide good prediction performance are preferred. We train the selected pipeline on the full training dataset and store the resulting pipeline artifact using joblib along with its metadata as a JSON blob, in cloud storage. The location and the naming of the model/pipeline are determined by the nature of the problem being tackled, the subset of lines the model is targeting, and a version number. The metadata includes information about the training data used, the best hyper-parameters, the cross-validation performance, and the list of selected features. In some cases, we also serialize the model/pipeline to a text JSON representation, by explicitly extracting the core artifacts of the model. This is required to port the model across processor architectures/OS, as some of our edge devices use different architectures.

All selected models are validated against a previously stored Reference dataset - that is chosen using consultation with our customers, and reflects a period that is representative of the operations of the line/s. This extra validation is performed to make sure that the model/pipeline performance is not significantly different from what we have observed with prior models. In cases that there is a major difference, we perform manual intervention to determine the cause of the change, and decide whether to use the new model, and correspondingly whether to update the Reference dataset. Otherwise, the model is selected for deployment and moved to the deployment location from where it is picked up either by the scoring data pipeline in the cloud, or a scoring pipeline deployed at the edge device.

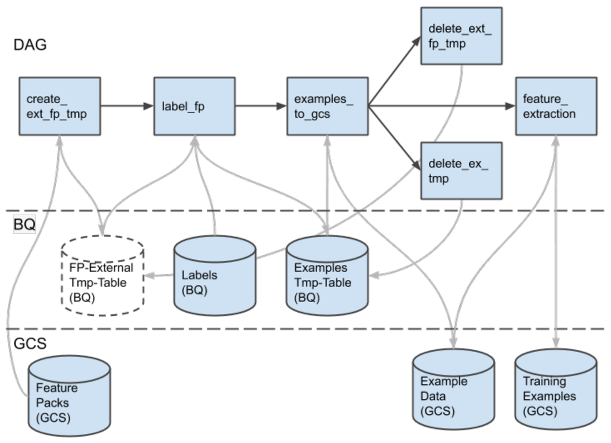

Fig. 4. `example-dag` BigQuery Metrics Tables

E. Workflow Orchestration

We orchestrate individual steps defined in Section III-C using Apache Airflow, a task scheduling and dependency resolution tool. Airflow allows us to define directed acyclic graphs (DAGs) composed of interdependent tasks defined by Operators. We build standardizing abstractions for each each task on top of existing operators in the Airflow standard or contrib libraries. These DAGs describe the sequence of tasks and their dependencies along with inputs and outputs, e.g. where they get data from and where they store their result, and other configuration.

One example DAG for the predictive quality application in Section IV is shown in Figure 4. This DAG corresponds to (Metric Packer →Labeler→Feature Extractor→Model Training) and is responsible for building training examples from metric packs. From left-to-right, `create_ext_fp_tmp` creates the metric packs in a temporary external table. In our next step, `label_fp`, a query reads from this temporary table, assigns labels and writes out labeled metric packs to a new temporary table. These are moved from BigQuery to GCS, and finally, the Feature Extractor uses Apache Beam to extract features from the packs, and write the training examples to GCS.

Similarly, we build a separate DAG for ML model Training and Validation. Once we define the DAGs, we can then configure them with appropriate parameters, including date ranges, storage location etc. These DAGs are configured to run periodically such that we can produce newer versions of the trained ML model as needed. Airflow manages the workflow ensuring that these different stages are executed in order, monitoring the successful completion of each stage before starting the next, and making attempts to restart on failure.

F. Model Scoring in Cloud and at Edge

We leverage networks of on-site IoT Edge devices to acquire and stream data, and also to perform local processing such as with ML model scoring. This ability to distribute our processing across cloud and Edge provides us the ability to meet different customer requirements in terms of performance and latency, to tolerate faults and network connectivity issues, and to maintain data and information security. Our edge devices are ARM-based embedded Linux devices that run different containerized applications. The architecture of our edge device is shown in Figure 5. We have a primary container that runs our base applications and machine communications stack, allowing the device to communicate with manufacturing equipment. In addition, we have a common set of application containers that allow them to stream the data to the cloud as well as support the needs of local visualization and Operator Interface. We also allow for a set of extensible containers where customer environment and processing specific code and models can be deployed. This allows us to dynamically add, modify and extend the types of processing performed at the edge. We do this through a unique combination of an over-the-air (OTA) updating technology and cloud to Edge synchronization system that allows Edge devices to get updates quickly and with minimal network strain. In order to enable

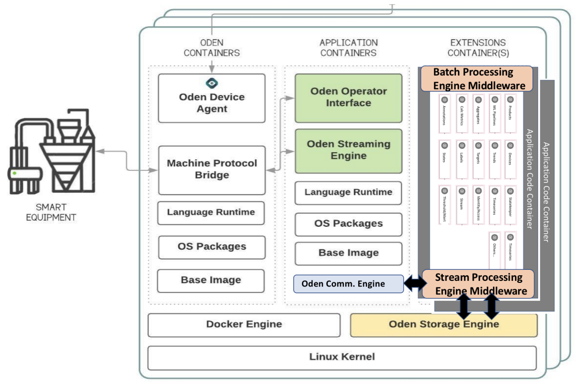

Fig. 5. Edge Architecture

model scoring to run on these devices, we build ML containers that include all the packages and dependencies needed to deploy the ML model and the streaming engine within this container. This allows us to deploy Beam jobs that load and score the model against the streaming data. Using Apache Beam allows us to change the target deployment environment easily between Cloud and Edge by only modifying the target runner. This also allows us to dynamically decide where to deploy our model scoring, as a function of current conditions and workload. This also allows our ML model scoring to be scalable even at the edge (utilizing multiple processing cores) and to be fault tolerant. We continuously monitor the performance of the model scoring using custom metrics that report throughput and latency. The accuracy of the model is measured within the cloud and allows us to determine when a new model is needed. Due to the containerization of the applications, it is easy to add more devices to an Edge network at a factory and deploy software and ML model updates when needed. Within the ML model scoring container, we include a service that checks the cloud for the availability of a new model version, downloads it to the device, and notifies the scorer to reload the model. This allows us to adapt and change our models over time as data characteristics change.

IV. Application: Predictive Quality

We now describe how we use the Mímir framework to build an application for online predictive quality monitoring. We look at the problem of plastics extrusion for wire coating problems. In this case melted plastic is coated over a certain type of wire (e.g. copper) to create a sheath. The quality of the coating is determined by measuring the outer diameter of the resulting cable, and making sure that it lies within a certain tolerance $[L, U]$ where less than the lower limit L indicates insufficient insulation, and greater than U implies too much material coated. This diameter (called `cold-od-average`) is measured using a laser system after the wire leaves the extruder and cools, and is only available 3-5 minutes after the wire has left the extruder. Given that extruders often run at 100s of feet per minute 3-500 feet of bad cable can be produced before a problem is detected. For real-time control we need to provide continuous estimates of the quality, and hence need a predictive model. Importantly, the model needs to be operational close to the manufacturing floor to minimize latency and provide resilience to poor network connectivity to the cloud.

The inputs to this model include metrics such as the previous values of measured quality, `cold-od-y`, `cold-od-x`, `cold-od-average`, measurements of diameter before it cools `hot-od-x`, `hot-od-y`, and the extruder control parameters `speed`, `screw-rpm`, `temp-die`, `temp-flange` and `pressure`. We use the model to predict the expected `cold-od-average` 5 minutes into the future, and then use that to estimate the quality of the cable in real-time, i.e. predict control limit $[L, U]$ violations as *In-Range*, *USL Violation* and *LSL Violation*.

We now describe how we use the Mímir framework to both build this model in the cloud as well as deploy it at the edge. As metric packs, we use a 5 minute window of these streaming metrics, and as labels, we use the average of the `cold-od-average` over a 30 second interval from 5 minutes after the end of the window. In order to generate the training examples, we deploy a Metric Packer$\rightarrow$ Labeler$\rightarrow$Feature Extractor DAG orchestrated using Airflow. The Metric Packer and Labeler jobs use Google BigQuery to create the labeled metric packs, which are then processed by the Feature Extractor running on Apache Beam to create training examples. BigQuery and Apache Beam Dataflow allow us to scale computational resources as the amount of data (number of extruders) grows.

We deploy a separate Airflow DAG to orchestrate the Cloud ML Training$\rightarrow$ Validation job that is triggered periodically, once a week. At each training iteration, the training task selects examples from Cloud Storage for a specified date range, and then explores multiple models including Lasso, Random Forest, k-NN and Gradient Boosted Trees. In each case, the task performs hyper-parameter optimization to search for optimal parameter setting using 10-fold cross-validation.

For each model we store the resulting metadata that captures training performance, model complexity, model parameter values, training data range, and extruder identifier. It is important to note that we train in parallel one model per extruder that we are monitoring, and all steps are designed to scale as this number changes. In our largest deployment, we have around 50 such models that are concurrently built.

We select a model to deploy based on rules applied to the model performance and complexity. These rules are crafted to account for the corresponding edge device computational resources as well as model accuracy. For instance, we prefer Linear Lasso models to more complex models if the performance of these models is within a certain squared error threshold of the best performance. This threshold is computed based on tolerances specified by the process engineers. The models selected for deployment are stored in Cloud Storage in two different formats - a serialized joblib or HF5 format (based on the model) as well as a plain text JSON representation. In order to generate the JSON representation, we extract all the model components and serialize them explicitly as JSON text. For instance a Lasso model may be serialized into JSON by explicitly extracting the `coef_`, `intercept_` and `n_iter_` parameters from the trained model. Each model is given a version number that is unique to the model. Once the final model is trained, we download it onto the appropriate edge device (using the mechanism described in Section III-F). At the edge, we deploy a workflow Metric Packer $\rightarrow$ Model Scorer that run continuously against the live data to generate the prediction as a new metric. The prediction can then be visualized using the operator interface to allow operators to modify and control the process incrementally.

V. Experimental Results

We present results on the scalability of the framework, the accuracy of models created, and real-time model scoring.

Scalability of Mímir Framework Using our optimized multi-resolution data storage framework, we can access large amounts of metric and context very efficiently. For instance, to build a predictive model we may need 6 months of metric data along with the context. Per line, with an average of 10 interesting metrics, this translates into roughly 15-20 GB, with an added significant join complexity for the context. For the predictive quality application, we instead pull data from a view that summarizes metrics at 5 minute boundaries and joins them with sub-intervals aligned at minute boundaries (Section III-B), and are able to retrieve data for 10 lines in under 70 seconds.

Our model training infrastructure scales using Google CloudML. Our models have on average 5-7 hyper-parameters each, our feature selectors have 2-3 hyper-parameters, and we explore 5 types of models and 3 types of feature selectors during each model training. We use `RandomizedSearchCV` for hyper-parameter optimization of `scikit-learn` models, and `RANDOM_SEARCH` within Cloud ML for deep learning models. With 10-fold cross-validation, this often turns out to several thousand model training runs per model training. For the predictive quality application, when training on 3 months of extruder data, our training performed 7500 experiments over 6 hours.

Model Prediction Accuracy One example of the performance achieved by our model, in terms of the predicted and true `cold-od-average` for a Lasso based model are shown in Figure 6. The prediction achieved by the model tracks the real value fairly accurately. In this case, the achieved MSE is $17e^{-6}$. More importantly the model also captures violations of the specified control limits quite well. Even though we do not explicitly optimize for classification performance, we can compute precision and recall for predicting *In-Range*, *USL Violation* and *LSL Violation* based on the prediction of the `cold-od-average` value. With the Lasso model, we can achieve over 85% precision with over 90% recall, which meets the desired tradeoff between false alarms and missed detection of our customers.

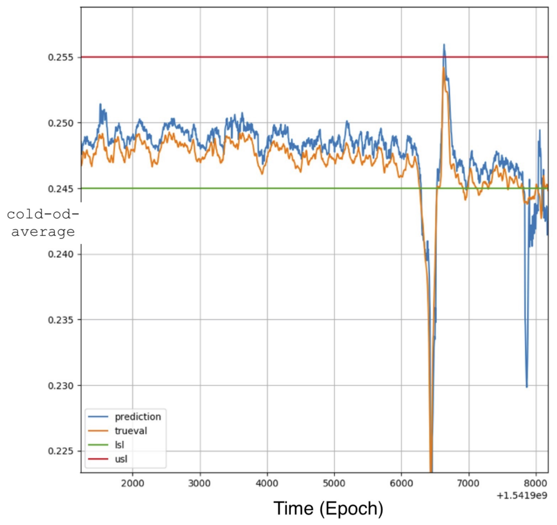

Fig. 6. `cold-od-average` Prediction

Model Scoring in Cloud and at Edge We score the model using a Cloud Dataflow Runner for Apache Beam in the cloud, and an Apache Flink Runner at the edge. In the cloud Dataflow includes an optimizer to scale the scoring by dynamically adding workers as needed to handle spikes in the data volume. For the predictive quality application, we have observed that model scoring achieves a throughput of 430 tuples/sec per worker. This is more than sufficient to handle multiple models in real-time. Even at the edge we have observed throughputs of 325 tuples/sec per core while scoring the Lasso based predictive quality model.

VI. Conclusion

In this paper, we provide an overview of the Mímir framework for training, experimentation, and end to end production grade deployment of ML applications for Industrial IoT applications. This framework is built on top of several cloud services and open source tools, along with our custom extensions for the manufacturing environment. We detail the system architecture, and describe several design and implementation choices that allow us to efficiently and robustly capture, stream, store, and analyze the process data as well as all context information. We first describe how we ingest the streaming data and create an optimized view that normalizes all metrics with appropriate taxonomy labels, merges them with context intervals, and quantizes these into fixed length periods that allows us very efficient access to data and context at multiple time resolutions. We then describe how our ML workflow model, that treats individual steps of the analysis process as atomic tasks, and how we leverage Apache Airflow to flexibly orchestrate them to realize different combinations of offline and online training, validation and scoring. This also allows for workflows that can be deployed at both the edge as well as in the cloud. Our framework includes a range of ML algorithms parameterized with appropriate ranges for their hyperparameters for scalable hyperparameter tuning and model selection based optimization. We include domain specific rules on tradeoffs between model accuracy and scoring computational cost, such that we can deploy our model scoring at the compute-constrained edge. We provide one example of applying the framework for a predictive quality application for a plastics extrusion processl. We show that we can provide `cold-od-average` predictions 5 minutes into the future (at over 85% precision and 90% recall) thereby allowing for the early detection of quality issues, and real-time control. This is supported by the scalability of the platform where we are able to access over 6 months of data and context for 10 lines within 70 seconds, perform thousands of experiments to select the best model and parameters for the task, and deploy models at the edge for real-time scoring.

We are using the Mímir framework to deploy applications for root cause analysis, predictive maintenance, performance optimization, and outlier detection. These also include unstructured data and knowledge such as images, audio signals, reports, manuals etc., within our ML workflows. While there is a lot of interest in IIoT, both commercially as well as academically, there are few technical publications describing instantiations of end to end systems. By laying out our system architecture, design choices, and implementation, we intend to spur further technical exchange, and accelerate the deployment of such frameworks to eventually realize Industry 4.0.

VII. Acknowledgements

We would like to acknowledge the contributions of the Oden engineering team for providing us the platform, support and tools to build the Mímir framework. Specifically, we would like to thank John Turek, James Maidment, Nathan Mehl, Dan Honey, Jake Peyser, Hampus Ohlssen, Jack Byers, Ziyao Wei, Alice Rottersman, and Doug Benedicto. We would also like to thank Mike Monroe for providing us with the domain knowledge related to plastics extrusion, and helping us develop the application for predictive quality monitoring.

References

[1] Apache Airflow. https://airflow.apache.org.
[2] Apache Beam. https://beam.apache.org.
[3] AWS IoT. https://aws.amazon.com/iot/.
[4] Azure IoT. https://azure.microsoft.com/en-us/overview/iot/.
[5] Google Cloud Platform. https://cloud.google.com.
[6] Google IoT. https://cloud.google.com/solutions/iot/.
[7] IBM IoT. https://www.ibm.com/internet-of-things/industries/iot-manufacturing.
[8] Oracle IoT. https://www.oracle.com/internet-of-things.
[9] K. Hazelwood, S. Bird, et al. Applied machine learning at facebook: A datacenter infrastructure perspective. *HPCA*, 2018.
[10] Y. Low, J. Gonzalez, et al. Distributed graphlab: A framework for machine learning and data mining in the cloud. *VLDB*, 2012.
[11] S. Lucero. Iot platforms: Enabling the internet of things. *IHS Technology White Paper*, 2016.
[12] Y. Meidan, M. Bohadana, A. Shabtai, J. Guarnizo, M. Ochoa, N. Tippenhauer, and Y. Elovici. Profiliot: A machine learning approach for iot device identification based on network traffic analysis. *SAC*, 2017.
[13] U. Shanthamallu and A. Spanias. A brief survey of machine learning methods and their sensor and iot applications. *IISA*, 2017.
[14] K. Singh and D. Kappor. Create your own internet of things. *IEEE Consumer Electronics Magazine*, 2017.