Politecnico di Torino

Politecnico di Torino

Master's degree in Data Science and Engineering

A.a. 2024/2025

July 2025

Analytics Ingestion Service: a data ingestion pipeline for MLOps

Relatori:	Candidati:
Prof. Paolo Garza	Francesco Lacriola

Table of Contents

ABSTRACT

With machine learning at the core of detection systems, the ability to systemically capture, store and analyze the outputs the model generates has become critical for operational efficacy and model governance. This thesis presents a MLOPS pipeline, the Analytics Ingestion System (AIS), developed to collect and persist rich analytics from the Machine Learning powered cybersecurity workflows within the Sysdig ecosystem. AIS offers a model-driven approach to managing inferences, so that any prediction, batch or real-time, is logged, contextualized, so that it can be used for further analysis.

The platform has also been designed for scalable storage, message brokers and orchestration tools to allow for modular, cloud native deployments. Particular attention is paid to observability and traceability of the model outputs, which allows good audit and data collection best practices across the different versions of the model and input sources that it receives. By modelling model inference as a first-class analytical event, the AIS provides a foundation for more elaborate analyses downstream, such as between model comparisons, data and concept drift detection, and A/B testing.

In this design, the AIS provides a crucial common carrier for ML-driven security systems at Sysdig that demand with more than just inference and a production quality infrastructure, to get intelligence from those inferences.

Chapter 1

Introduction and context

The deployment of machine learning models in cybersecurity environments creates a critical operational challenge that extends beyond model development: how to systematically capture, process, and store the outputs of production models at scale. While considerable attention has been devoted to training pipelines and initial model validation, the systematic collection of model inferences, the actual predictions generated during operation, remains an under addressed requirement in most ML implementations. This gap is particularly problematic in cybersecurity contexts, where models must continuously detect threats, such as cryptomining malware, among evolving attack patterns and changing system behaviors.

ML models in production environments face several operational realities that necessitate comprehensive analytics collection. First, the distribution of input data inevitably shifts over time, causing gradual performance degradation as models encounter scenarios different from their training data.

Second, in adversarial domains like cybersecurity, threat actors deliberately modify their techniques to evade detection, creating a concept drift, where the relationship between features and target outcomes changes.

Third, security operations require auditability and traceability of detection decisions for compliance and forensic purposes. Without structured mechanisms to capture model outputs, security teams operate with limited visibility into how models actually perform against real-world threats beyond their initial deployment.

Usually, MLOps practices inadequately address these challenges by focusing primarily

on model deployment and training pipelines rather than creating robust infrastructures for inference analytics collection. Organizations typically deploy models that make predictions which trigger immediate security actions, but these valuable signals are often treated as ephemeral outputs rather than persistent analytical assets. This approach creates significant blind spots in operational monitoring, impedes performance diagnostics when models fail to detect threats, and complicates essential governance tasks such as model comparison and version management.

The Analytics Ingestion System (AIS) introduced in this thesis directly addresses these operational requirements through a purpose-built pipeline that treats model predictions as first-class data product, considering their need for systematic processing and persistence. As we will see in the system architecture, the AIS creates a structured workflow that begins when ML worker nodes generate predictions through their inference processes. Rather than merely acting on these predictions, the system captures them through a robust message queue infrastructure, which provides reliable transport while decoupling prediction generation from downstream processing. Its design provides the flexibility to handle both high-volume, latency-tolerant batch predictions and time-sensitive real-time detection events within the same infrastructure. The AIS culminates in structured persistence of all analytics data through a standardized API interface to predictions logs storage, transforming ephemeral prediction events into durable analytical assets that support essential operational capabilities.

With this comprehensive collection mechanism in place, at Sysdig, security and data teams gain the ability to analyze model performance across different time periods, detection scenarios, and system environments. This visibility enables empirical assessment of when models begin to degrade, which threat variants they struggle to detect, and how different deployment configurations affect detection accuracy.

The implementation of the AIS addresses several practical challenges inherent in operational machine learning for cybersecurity.

First, it handles the high volume and velocity of predictions generated by continuous security monitoring across enterprise environments. Second, it accommodates the heterogeneous nature of security analytics by supporting multiple model types and prediction formats within a unified collection framework. Third, it minimizes performance impact on critical detection paths through its asynchronous design. Finally, it creates the foundation for advanced operational capabilities like model performance comparisons and automated drift detection.

By focusing on the systematic collection and processing of model outputs, the AIS fills a critical gap in the operational machine learning lifecycle. While most MLOps approaches emphasize how models are built and deployed, the AIS addresses what happens after deployment, how the actual intelligence generated by these models is captured, processed, and leveraged for continuous improvement.

1.1 Traditional cybersecurity approaches

The cyber environment is constantly changing because the new threat environment itself is dynamic. Legacy detection technology, while vital to today's defense initiatives, typically has a tough time when confronted with advanced, adaptive attack practice. Legacy security systems were made in an era of static threats, viruses and exploits that could be caught based on defined signatures or foreseen patterns of activity. But as threats have become increasingly elusive, adaptive, and sophisticated, the shortcomings of these conventional detection approaches have become more pronounced. In reaction, modern security infrastructures are moving towards more analytics-oriented, flexible architectures, typically constructed with Machine Learning (ML) and analytics capabilities. Prior to an examination of these advanced paradigms, it is important to delve into the classic forms of detection that remain the foundation of much security infrastructure today: rule-based detection and heuristic techniques.

1.1.1 Rule-Based Detection

Rule-based detection is probably the oldest cybersecurity method. Rule-based systems operate by comparing observed data with pre-established rules or signatures in an attempt to determine if they represent threatening or suspicious activity. These rules are usually crafted by experts from a given domain and reflect established indicators of threat such as file hashes, IP addresses, traffic patterns, or system behaviors. What rule-based systems monitor for includes:

- File names, hashes, or behavior signatures that are recognized to be associated with malicious software.
- Outbound network connections to recognized-suspicious IPs or domains, unusual port utilization, or unusual traffic frequency.
- Unusual CPU, memory, or disk resource utilization caused by specific processes.

Rule-based detection systems are prevalent in firewalls, Intrusion Detection Systems, and endpoint security programs. Their advantages are:

- They employ simple matching logic, thus they are computationally inexpensive and well-suited to low-latency environments.
- They are easy to configure and implement.

They do have drastic shortfalls, though:

- They can't recognize new, evasion techniques, or evolving threats that vary even slightly from known patterns.
- It is needed to constantly update rule sets to keep pace with new attack surfaces is

labor-intensive.

- Susceptibility to False Positives/Negatives: innocent activity will sometimes overlap with rule-based activity, whereas new malicious activity will be picked up on not at all. [1]

Since current threats are more likely to incorporate obfuscation and polymorphism, rule-based systems are not the first level of defense and are usually assisted by more dynamic methods.

1.1.2 Heuristic Techniques

Heuristic detection offers a more dynamic alternative to static rules by examining behavioral cues that can signal malicious intent. Instead of relying solely on preconfigured signatures, heuristic systems analyze runtime behavior, system interactions, and usage patterns to identify anomalous deviations from baseline norms. Typical heuristic checks may involve:

- Suspicious process behavior: inconsistency in system calls, file accesses, or fast execution patterns.
- Resource usage anomalies: high CPU, memory, or disk usage sustained over time but not typical for the usual application behavior.
- Timing irregularities: processes running irregularly or displaying suspicious scheduling patterns.

These approaches are intended to detect unknown or altered threats by generalizing from observed activity instead of exact known patterns. Advantages are:

- Heuristic analysis may detect new or zero-day threats that static rules would not.
- More independent of pre-existing threat knowledge, therefore more resilient in dynamic threat landscapes.

However, heuristic techniques also suffer the following disadvantages:

- Higher False Positives: legitimate programs occasionally exhibit behaviors that are highly like malicious behavior and therefore produce false alarms.
- Live behavior monitoring is costly in terms of computation and affects system performance.
- Developing effective heuristics that are sensitive and specific enough requires extensive knowledge and experience with the context. [2]

1.2 Machine Learning approaches to cybersecurity

The application of Machine Learning (ML) in cybersecurity marks a significant departure from traditional, static security measures, offering dynamic and adaptive

solutions to address the increasing complexity of modern cyber threats. By leveraging its ability to learn from data, identify patterns, and predict future occurrences, ML provides an effective means to detect and mitigate a range of cyber threats. This chapter explores how the diverse ML techniques can be employed in cybersecurity, focusing on general applications.

1.2.1 Machine Learning for cybersecurity

Machine Learning (ML) has reshaped the way cybersecurity threats are identified and addressed. Unlike traditional systems, which depend on fixed rules and signature-based detection, ML offers a dynamic, data-driven alternative. This adaptability allows defenses to evolve alongside emerging threats, detecting not only known attacks but also previously unseen ones.

One of ML's greatest strengths in cybersecurity is its flexibility. While traditional defenses can be circumvented with slight modifications to attack patterns, ML models are trained on continuous data streams. This enables them to detect subtle anomalies and learn generalized behaviors—allowing them to flag threats they've never encountered before.

ML also excels in behavioral analysis. Rather than relying solely on pre-set threat signatures, it can examine fine-grained system attributes like CPU usage, memory patterns, and network activity. This makes it especially effective at identifying stealthy threats, such as cryptomining malware, which often run silently in the background. By building a profile of what "normal" looks like, ML systems can detect even minor deviations that older systems might overlook.

Beyond its adaptability, ML brings scalability and efficiency to the table. Today's digital environments generate vast quantities of data, too much for humans to manually sift through. ML algorithms can process this data in real time, supporting rapid threat detection across large-scale enterprise and infrastructure settings.

Another major benefit is ML's ability to improve over time. As new data comes in, models refine their understanding of both regular and malicious behavior. This continuous learning keeps them relevant, even as attackers develop more sophisticated techniques.

However, integrating ML into cybersecurity isn't without challenges. Supervised learning models, for instance, depend on large volumes of high-quality, accurately labeled data—something that's often expensive and labor-intensive to collect. And while real-time detection is a goal, the computational demands of training and deploying such models can strain limited resources.

ML models are also not immune to manipulation. Adversarial attacks, deliberate inputs crafted to fool the system, can cause ML models to misclassify threats. A cryptominer,

for example, might mimic regular software activity to slip past detection.
Lastly, there's the issue of explainability. Security teams often need to understand why a system flagged a certain threat. Some ML models, particularly deep learning ones, act as "black boxes," making it difficult to interpret their decisions. [3]
To better understand how these benefits and limitations play out in real-world applications, it's essential to explore the core ML paradigms used in cybersecurity, specifically, supervised and unsupervised learning.

1.2.1.1 Supervised Learning in Cybersecurity

Supervised learning relies on labeled data sets in which each data instance, e.g., network packet, system log record, process, is tagged as normal or malicious. This type of learning allows the model to separate known behavior categories by learning the attributes associated with each label.
For example, a supervised model trained on system logs for machines infected by cryptomining malware is able to learn to identify important indicators like anomalous CPU spikes and suspicious network communication patterns. After being trained, the model is able to report similar instances in real-time. Supervised learning is especially effective when handling characterized attack vectors and is capable of achieving high detection accuracy assuming that data is representative and diverse.

Another advantage of supervised models is that they are more interpretable. When a threat is detected by the model, analysts generally know exactly which particular input features, e.g., bandwidth activity, file access patterns, or process behavior, determined the decision. This transparency helps with both response to an incident and trust in system judgment.
However, its primary shortcoming is its reliance on labeled data. Labeling cybersecurity data is normally an intensive labor process, usually involving specialist knowledge to classify complex instances. In addition, supervised models suffer from being unable to handle novel attack types whose behavior differs from the training data. [4] For instance, an instance of cryptomining malware that slows its CPU utilization to resemble benign system processes or obscures its communications through encryption to conceal outbound traffic can evade a model that learned from past forms of the same threat when they were more conspicuous. This is evidence of supervised systems' native brittleness in dealing with novel attack approaches.

1.2.1.2 Unsupervised learning in Cybersecurity

In contrast to supervised learning, unsupervised learning is not based on tagged data.

Instead, it examines raw data to determine patterns, clusters, or anomalies that do not fit into the normal behavior. This makes it especially useful in security settings, as threat profiles change rapidly and numerous attacks might never be similar to previously seen patterns.

In the case of cryptomining detection, unsupervised approaches could flag a previously unknown cryptomining activity by detecting an abnormal spike in outbound traffic or unexpected spikes in CPU activity during idle times. These patterns may not be associated with any previously seen malware but still be representative of an intrusion or abuse of system resources.

The power of unsupervised learning is its capability to find zero-day attacks and other emerging threats. Since it does not need to be pre-trained to recognize an attack, it is ideal for deployment in settings where flexibility and responsiveness are primary concerns. Additionally, the absence of a need for labeled training data ensures faster and less costly deployment, enabling organizations to harness ML capabilities even when labeled data is missing or incomplete. [5]

But unsupervised learning poses an interpretation problem as well. Since their anomalies are statistically rather than semantically defined, it is hard to decide whether an alert is genuine or a false positive. An increase in network traffic could be evidence of attacks, or software installation. That uncertainty more often results in false positives, and if they happen too regularly, they lead to a desensitization among security teams and systemwide ineffectiveness.

To sum up, machine learning became an essential driver behind developing cybersecurity defenses in response to increasingly subtle and ever-changing attacks. With an offering of learning from data, identifying nuanced behavioral patterns, and scalability in large and complex settings, ML greatly increases the resiliency and responsiveness of contemporary security solutions. However, ML is not a silver bullet. The data dependence, computational intensity and interpretability need to be carefully controlled. Moreover, the decision to use supervised vs unsupervised approaches is context-dependent as well as data and threat landscape dependent.

1.3 MLOps pipelines and monitoring

As cyber threats grow in complexity and volume, traditional rule-based and heuristic approaches are no longer sufficient on their own. As we said, the cybersecurity industry is increasingly adopting data-driven methods such as Machine Learning to enhance detection capabilities. However, building accurate models is only part of the solution; ensuring their effectiveness in real-world environments demands robust infrastructures for deployment, monitoring, and analysis.

This is where MLOps (Machine Learning Operations) becomes critical.

Combining principles from software engineering, DevOps, and data science, MLOps provides a structured framework for managing the full lifecycle of ML models, from deployment to continuous monitoring and maintenance.
In cybersecurity, MLOps platforms must do more than serve models; they must also enable observability, traceability, and structured data collection to support downstream tasks like performance evaluation, data and concept drift detection, and A/B testing.
This thesis introduces an analytics-oriented MLOps infrastructure tailored for cybersecurity applications. The proposed ingestion service is designed to manage model-generated signals at scale, ensuring that inference outputs are auditable and actionable. In doing so, it shifts the focus from model accuracy alone to the broader operational ecosystem needed for reliable, adaptive, and transparent ML-based defense systems.

In this chapter, we explore the complexities of ML model deployment, the challenges of production monitoring, and the essential role of pipelines in addressing issues like data drift and model drift.

1.3.1 Deploying ML Models and continuous monitoring

Getting machine learning models deployed to production is more than replicating code from a notebook to a production system, and then running it. It involves a whole range of practices with the goal that models work well, are efficient, and reliable when applied in real-world situations. This involves not only the deployment, but also ongoing monitoring and maintenance of model performance throughout its lifecycle. Right at the center of this, we have MLOps pipelines that give us an infrastructure to automate, a streamline this process.
Bringing an ML model from research to production is about making sure that a model, trained and evaluated in a clean room, can work robustly with real data. This operation usually includes the following steps:
• Model productization and deployment in autonomous driving systems or incorporated in products or services.
• Serving real-time or batch predictions, based on use case.
• Compatibility with different environments, such as software dependencies, hardware setups and data preprocessing logic.
And unlike typical software, ML models are inherently data-specific. What a model does depends not only on the code it comprises, but also on the data to which it is applied. Hence, even minute differences in production data pipeline may violate the assumptions made during training and result in performance degradation. Also

production systems tend to have very strong requirements in terms of latency, scalability and fault tolerance, especially in fraud or cyber security.

A model does not stay unchanged when deployed; the environment changes. A model's prediction must be continuously monitored so that it does not drift over time. Critical performance that must be tracked in real-time include prediction accuracy, response times and resource usage.
Instead, two widespread and important challenges for the model degradation are:
• *Data Drift*: the change on the distribution of input data, along the time.
• *Model Drift*: when the statistical relationship between inputs and outputs changes, which affects how well the learned parameters of a model can perform.
Left unchecked, either type of drift can silently degrade model performance and invalidate downstream decisioning.

1.3.2 MLOps Pipelines

Deploying a model is not the end of a process; ML models are not static entities. Any amount of data is ever sufficient data, and, the more we collect, the more models learn. More data tells us more about the world, which, in turn, allows us to learn new things and make better decisions.
Model Lifecycle Management, also known as MLOps, answers these needs. It can be sounded out as a series of actions that guarantees models to be valid and perform as expected, also in the future. The idea is to be proactive, i.e. to be able to notice a drift or performance deterioration even before they actually occur. Auto-retrain mechanisms allow updating a model automatically to keep up with the traffic evolution, but, at the same time, it must be done with care as, in some cases, further updates may yield negative effects only.
MLOps pipelines introduce automation, reproducibility, and observability into every stage of the workflow. Key components include:
• Data ingestion and preprocessing: this is where we fetch, process, and validate the fresh data and check no missing values no new patterns or new ones can be found.
• Training and validation pipeline: only the best 5 percent of models are moved to the next stage. We can also simultaneously put the new model into production and, if after a period of observation it has not yet been proven good, we would not promote it.
• Automated deployment: every time a new model is created that passes all the controls, pipelines help to compile it and put it into the deployment environment with all the other elements, such as the right container.
Deploying a model is nothing but the beginning of a new challenge, with new data which is expected to change and with models which have to adapt. All this is possible with

MLOps.

Chapter 2

Related Works

2.1 Related Works on MLOps Pipelines and Monitoring

The deployment and monitoring of machine learning models have received considerable attention in recent years, resulting in a diverse ecosystem of tools, frameworks, and methodologies. In this chapter, we examine the most relevant literature and existing production-grade solutions to draw inspiration for the design and implementation of AIS. Our review focuses on established practices and innovations in areas such as analytics ingestion, real-time model monitoring, A/B testing, and drift detection.
We analyze how leading platforms and frameworks, such as TFX (TensorFlow Extended), KubeFlow and others, tackle these challenges, identifying key design patterns and trade-offs that informed our architectural decisions for AIS.

2.1.1 TensorFlow Extended

TensorFlow Extended (TFX) is a leading framework for the ML lifecycle, a production-grade system that ties together the ML pipeline stages. TFX is relevant as a related work for its ingestion pipelines and alignment with the goals of scalable analytics ingestion systems.
TFX supports a wide range of ML applications, with a focus on scalability, reproducibility, and automation. It enables a smooth transition from research to production, with modularity to allow custom workflows while using its prebuilt components. This is in line with our Analytics Ingestion Service, as both systems manage high volume, dynamic data pipelines with a focus on modularity and scalability.

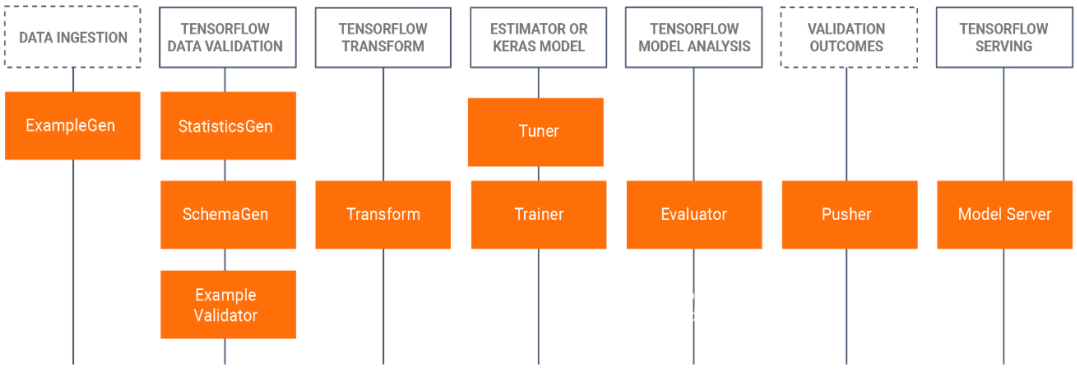

Figure 1: TFX includes both libraries and pipeline components. This diagram illustrates the relationships between TFX libraries and pipeline components.

The TFX pipeline starts with the ExampleGen component, which ingests raw data from various sources like CSV files, TFRecords, BigQuery or columnar formats like Apache Parquet, which is also at the heart of our pipeline. This component reads the input data and partitions it into training and evaluation splits, ready for downstream processing. The data ingestion and transformation logic throughout the pipeline is done using Apache Beam, a unified programming model for batch and streaming data processing. This allows for scalable, distributed computation across different execution environments, from local machines to cloud native backends like Google Cloud Dataflow.

Once ingested, the StatisticsGen component computes statistics over the dataset, providing insights into feature distributions and helping to identify data issues. These statistics are consumed by SchemaGen which infers a schema specifying expected types, ranges and feature shapes. The ExampleValidator then checks the incoming datasets against this schema to detect anomalies like missing values or type mismatches, ensuring data integrity before training.

For preprocessing and feature engineering, the Transform component applies transformations like normalization, one-hot encoding or bucketing defined using TensorFlow Transform.

The Trainer component then consumes the transformed data to train a model using TensorFlow, optionally integrating hyperparameter tuning through the Tuner module. After training, the Evaluator component performs a detailed model quality analysis, using metrics and slicing specifications to verify that the model meets production thresholds. If the evaluation is successful, the Pusher component deploys the validated model to a serving system, such as TensorFlow Serving, for real-time inference.

To manage the end-to-end workflow, TFX integrates with orchestration tools like Apache Airflow and Kubeflow Pipelines. These platforms automate the execution of pipeline components, manage task dependencies, enable experiment tracking, and

support fault-tolerant operation of machine learning workflows. [6]

TFX has been widely adopted by organizations such as Spotify, underscoring its applicability to large-scale, production-grade ML systems. Spotify has indeed integrated TensorFlow Extended into its machine learning infrastructure. They detailed how they standardized their ML workflows using TFX and Kubeflow Pipelines to streamline model development and deployment processes. Additionally, Spotify has utilized TensorFlow and its extended ecosystem, including TF-Agents, for reinforcement learning applications in music recommendation systems. These implementations underscore Spotify's commitment to leveraging TFX for scalable and automated ML pipelines. [7]
In its implementations, TFX is exploited to enforce schema consistency, optimize data storage through columnar formats, and scale horizontally to meet high-throughput requirements. These features highlight its relevance as a related work for our system, particularly in its approach to handling isolated pipelines for production and non-production environments, schema enforcement, and fault-tolerant data processing.

While TFX offers a robust foundation for managing ML workflows, several lessons should be noted from its implementation:

- TFX requires expertise in multiple technologies, including Apache Beam, Airflow, or Kubeflow, and a deep understanding of its modular components. This can result in longer onboarding times and increased complexity in maintaining pipelines.
- TFX's reliance on distributed systems like Apache Beam and its orchestration platforms necessitates significant infrastructure resources. Organizations must invest in scalable computing environments and robust DevOps practices to support its deployment.
- TFX primarily assumes distributed processing but does not inherently address fault tolerance during in-memory operations. This limitation aligns with the challenges identified in our Analytics Ingestion Service, particularly concerning data loss risks when processing messages in memory without a write-ahead log strategy.
- TFX enforces schema consistency to maintain data quality, but this rigidity can pose challenges in dynamically evolving pipelines. For example, Parquet's inability to handle mixed data types within the same column requires strict upstream validation and preprocessing.
- The ability to scale ingestion and processing workloads horizontally ensures that high-throughput systems can handle growing data volumes without bottlenecks.
- Leveraging orchestration tools like Airflow or Kubeflow Pipelines provides not only automation but also crucial monitoring and error-handling capabilities for large-scale systems.

- Incorporating robust schema validation and exploring fault-tolerant strategies, such as write-ahead logs, are critical to ensuring the reliability of ingestion pipelines in production environments.

2.1.2 Kubeflow

Kubeflow is a Kubernetes-native platform developed to facilitate end-to-end machine learning (ML) processes and provide a modular and extensible architecture notably suited to container environments. Kubernetes is an open-source container orchestration platform that automates the use, augmentation, and administration of containers.

Kubeflow's main strenght lies in abstracting ML pipelines as containerized tasks orchestrated over Kubernetes, using native concepts such as pods (the most minute deployable unit of calculation in Kubernetes), support, and persistent capacity for scalable application and lifecycle management.

Kubeflow Pipelines, which allow the composition and execution of complex work flows as directed Acyclic Graphs (DAGs), a graph with an individual verge direct from one vertex to another identical that follows their direction will never structure a closed cycle. The above DAGs are executed using Argo Workflows, an implicit orchestration engine that automates task scheduling, dependency resolution, retry logic, and resource allocation. The corresponding node in the graph represents an individual container action, which makes it possible to modularize systematic analysis phases such as data preprocessing, feature extraction, model training, and analysis. Moreover, Kubeflow pipelines integrate metadata tracking, artifact handling, performance monitoring, facilitating reproducibility and auditability, and mandatory properties for production-grade information frameworks.

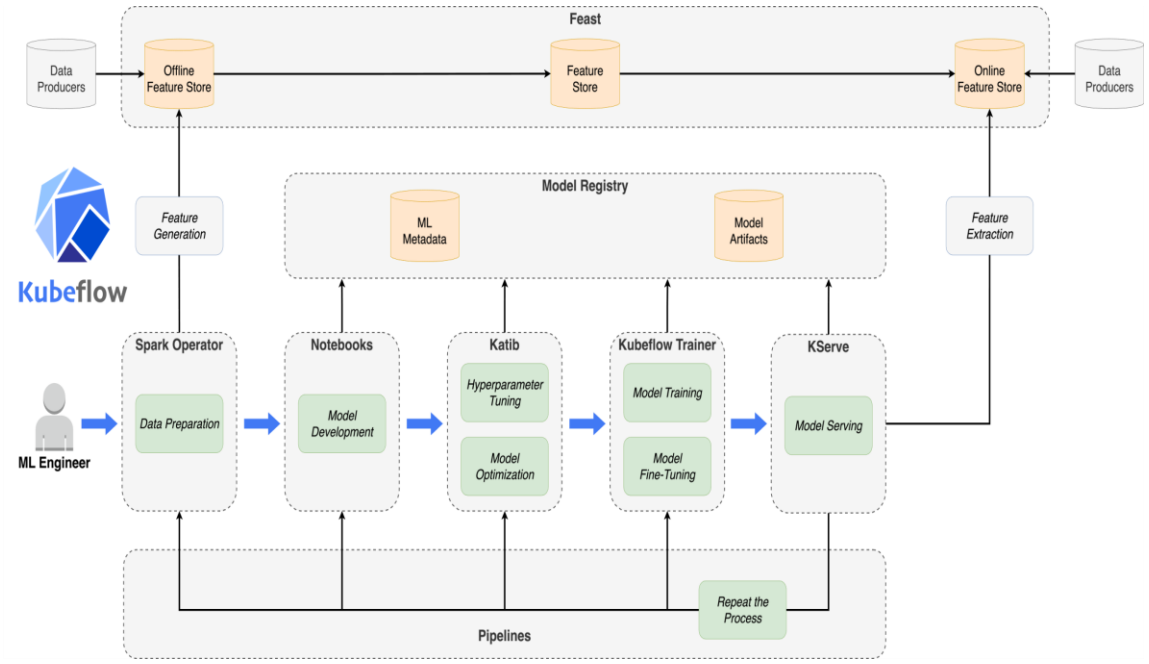

Figure 2: here are shown Kubeflow components and how they are used in its pipeline

For serving models in production, Kubeflow employs KFServing, which supports serverless inference. KFServing provides autoscaling, traffic splitting, and model versioning capabilities. It enables dynamic allocation of compute resources based on request volume, making it highly efficient for deploying lightweight analytics inference tasks.

Despite its comprehensive batch processing capabilities, Kubeflow does not natively support real-time or continuous ingestion pipelines. Its architecture is primarily designed around static workflows triggered by batch jobs or manual invocations. However, Kubeflow's Kubernetes foundation and modular pipeline interfaces make it well-suited for integration with external streaming systems. [8]

Notably, Apache Spark Structured Streaming and Apache Flink can be integrated to bridge this gap:

- Apache Spark Structured Streaming adopts a micro-batch execution model, enabling incremental updates with tight latency constraints. It provides structured APIs and stateful operations ideal for building low-latency ingestion pipelines where data arrives continuously. [9]
- Apache Flink, on the other hand, offers true stream processing with event-at-a-time execution semantics and exactly-once guarantees. [10]

From Kubeflow's design, several architectural insights can inspire the development of the Analytics Ingestion System:

- The separation of pipeline stages and the use of a DAG-based orchestrator (e.g., Argo) inform AIS's need for fault-tolerant scheduling, observability across ingestion, validation, and enrichment stages.
- Kubeflow's compatibility with streaming frameworks illustrates a viable hybrid model where batch pipelines are managed in Kubernetes while ingestion and preprocessing are delegated to real-time engines like Flink or Spark.
- The emphasis on containerized, reusable components suggests a best practice for designing AIS operators as self-contained, stateless services that can be tested and deployed independently.

2.2 Monitoring ML Models in Production

In production environments, machine learning models are susceptible to performance degradation due to factors like data drift, concept drift, and infrastructure anomalies. Effective monitoring systems are essential to detect and address these issues promptly. This section delves into relevant methodologies for detecting data and concept drift, as well as strategies for A/B testing.

2.2.1 Detecting Data Drift and Concept Drift

Data drift refers to changes in the statistical properties of input data over time, while concept drift involves changes in the underlying relationship between input features and the target variable. Detecting these drifts is crucial for maintaining model performance. Various methods have been developed to identify and address drift in streaming data environments.

Error-Rate Based methods monitor the model's error rate to detect significant deviations that may indicate drift. These include:

- Drift Detection Method: DDM assumes that the error rate of a stable model decreases over time. It monitors the error rate and its standard deviation, raising warnings or detecting drift when the error rate significantly increases. Specifically, if the current error rate plus its standard deviation exceeds the minimum recorded error rate plus a multiple (typically 2 or 3) of its standard deviation, a warning or drift is signaled. [11] [12]
- Early Drift Detection Method: EDDM enhances DDM by focusing on the distance between classification errors, making it more sensitive to gradual drifts. It tracks the average distance between errors and their standard deviation, signaling drift when these metrics deviate significantly from their historical maxima. [13]
- Hoeffding Drift Detection Method: HDDM utilizes Hoeffding's inequality to detect drift. There are two variants: HDDM_A, which uses the average error rate, and

HDDM_W, which employs a weighted average to give more importance to recent data. Both methods signal drift when the observed statistics deviate beyond predefined confidence levels. [14]

And then we have the Distribution-Based Methods. These approaches detect drift by comparing the statistical distributions of input features over time. For instance:

- Kullback–Leibler (KL) Divergence and Chi-Squared Tests: these statistical tests measure the divergence between the distributions of current and reference data. Significant divergence indicates potential drift.
- Two-Sample Testing with Dimensionality Reduction: Rabanser et al. propose using pre-trained classifiers for dimensionality reduction, followed by two-sample tests to detect dataset shift. This method effectively identifies shifts in high-dimensional data. [15]

Hence, some considerations that are inspirational for the development of the systems are:

- Choosing an appropriate window size for monitoring is crucial. Smaller windows allow for quicker detection but may increase false positives. Larger windows reduce false alarms but may delay detection.
- Implementing dynamic thresholds that adjust based on historical data variability can improve detection accuracy.
- Embedding drift detection mechanisms within streaming platforms like Apache Flink or Spark Streaming enables real-time monitoring and rapid response to detected drifts. [16]

2.2.2 A/B Testing

Measurement of the machine learning model in production requires rigorous statistical methodology to ensure that the determined performance difference is useful and scheduled for random opportunity. The usual A/B test is essential, but tailored strategies such as McNemar's Trial and Adaptive Experiments provide an extra strong option.

In classical A/B testing, users are randomly assigned to a unique group, each exposure to a specific model discrepancy. The main things that must be kept in mind while testing are:

- Ensuring unbiased assignment to treatment and control groups to mitigate confounding variables.
- Deciding the required number of inspections in order to find a significant divergence between disparate elements and high certainty.

- Preventing spillover effects between groups and ensuring a fair treatment across user segments. [17]

McNemar's test is a non-parametric method used to compare the performance of two classification models on the same dataset, specifically suited for binary classification problems where each prediction can be categorized into correct or incorrect. It is particularly valuable when dealing with paired data—such as the outputs of two classifiers on the same set of observations—and focuses on the disagreements between models rather than their absolute performance. The core idea is to assess whether the proportion of discordant pairs is significantly different, thus enabling the detection of subtle performance differences that might not be evident through traditional accuracy metrics.

Unlike classical A/B testing, which requires large-scale randomized user allocation and multiple experimental runs, McNemar's test evaluates models based on a single testing dataset. This makes it especially appealing in scenarios where model retraining or deployment is computationally expensive or infeasible. It operates on a 2x2 contingency table, counting how often one model is correct and the other is not, and vice versa, thereby directly comparing decision boundaries rather than average performance.

The paired t-test is another statistical method often used for comparing the performance of two models. However, it assumes that the performance metric (e.g., accuracy or error rate) across different folds of a cross-validation scheme follows a normal distribution. This makes the paired t-test more suitable for regression tasks or classification scenarios with continuous evaluation metrics and multiple independent measurements. In contrast, McNemar's test does not rely on distributional assumptions, making it a more robust choice for categorical, paired data and binary classification outputs. [18]

The classic A/B test allocates users equally between discrepancies, which may remain inefficient, particularly when one discrepancy is clearly superior. Multi-armed bandit (MAB) procedures deal with the present by dynamically adjusting the allocation of users in order to achieve different disparities based on their performance.Common MAB Strategies:

1. Epsilon-Greedy: with a probability ε, a random variant is selected (exploration); with probability 1−ε, the best-performing variant is chosen (exploitation).
2. Upper Confidence Bound (UCB): selects variants based on the upper confidence bounds of their estimated rewards, favoring variants with higher uncertainty to encourage exploration.
3. Thompson Sampling: a Bayesian approach that samples from the posterior distribution of each variant's performance, selecting the variant with the highest sampled value.
4. Best Arm Identification (BAI): focuses on identifying the best-performing variant as

quickly as possible, minimizing the time spent on suboptimal variants. [19]

Practical considerations for their usage:

- In environments where user behavior or data distributions change over time, non-stationary bandit algorithms that adapt to these changes are essential.
- Incorporating MAB strategies into MLOps workflows enables continuous model evaluation and deployment, ensuring that the best-performing models are always in production.
- Ensuring fairness and avoiding bias in adaptive experimentation is critical, especially when deploying models that impact user experiences or decisions.

Chapter 3

Project Details

3.1 Overview and Requirements

In any organization deploying machine learning models for real-time detection, the need for a robust MLOps pipeline for analytics ingestion has increased. As ML models generate continuous streams of detection data, having a scalable and reliable ingestion system ensures that insights can be derived effectively while maintaining data integrity and governance. In our case, with the deployment of the crypto mining ML detection model into production and the anticipation of future ML detection streams, the necessity for such a system became evident.

The primary motivation behind this project was to ensure that data generated by ML detection models in production environments could be efficiently collected, stored, and made readily available for analytics purposes. This necessitated the development of an Analytics Ingestion Service (AIS) capable of handling large-scale ML detections while ensuring scalability, reliability, and maintainability.

The main objectives of the Analytics Ingestion Service then are:

- Consumption of ML detections. The AIS must be able to reliably consume ML detections from each ML pipeline deployed in both production and development environments.
- Efficient ingestion into long-term storage. To facilitate future analytics, detections must be ingested in batches into durable and scalable object storage systems, such as Amazon S3.
- Support for MLOps analytics datasets. The ingested data must be structured and organized in a way that enables efficient querying and processing for downstream analytics tasks.

- (Long term goal) Support real time drift detection. The AIS should evolve to enable low-latency streaming capabilities that facilitate the monitoring of statistical properties of incoming data in real time. This includes computing drift metrics on incoming feature distributions and model outputs, triggering alerts when deviations from expected behavior are detected.

3.1.1 Context

The Machine Learning services in production are designed to consume streams of messages from queues fed by other company internal data sources. The general process at a high level that characterizes most of the ML services is the following:

1. The messages are unpacked, and eventually batched (to optimize the prediction phase)
2. Feature Engineering is applied to produce feature vectors
3. Feature vectors are fed into the ML model to get the predictions as a result.

We then decided that each ML detection service publishes, together with some metadata, the produced detections to a dedicated Analytics Ingestion queue subject, ensuring logical separation and operational independence across different ML pipelines. This architecture allows for tailored processing and consumption strategies, enabling each ML service to maintain its own data flow without interference. Moreover, the detection queues act as structured data sources that multiple consumer services can subscribe to and process based on their specific needs.

3.1.2 Requirements and Design Choices

To fulfil the goals described above, several design decisions and technical requirements were made:

- The analytics ingestion system must handle different ML Pipelines: there should be a different AIS instance that consumes from each deployed ML detection queue, since there is one for each detection service.
- We need a batch processing model: with the high volume of ML detections created in production, a batch processing methodology was embraced. This provides optimized performance in terms of ingestion rates and cost savings while avoiding the excess stress on computing assets relative to the real-time streaming solutions.
- Container orchestration and deployment plan: due to the requirement to have a scalable, feature-rich, and manageable production deployment setup, the orchestration platform was chosen to be Kubernetes. It facilitates effective containerized deployments, resource allocation, and failover management and is suitable to handle AIS instances in different production environments.
- Long-Term storage strategy: having compared various kind of storage, Amazon S3

has been selected as the key long-term storage medium for its reliability, scalability, and its ability to integrate with AWS analytics services.

- Predetermined choice in storage data format: to optimize storage, query and analytics processing, we considered various data formats and chose Parquet as the final option. This columnar format provides better compression ratios and query performance and is the ideal choice to store big data.
- Environment configuration differences for MLOps account: For proper governance of the data and separation between environments, we have configured separate AWS Prod-MLOps and NonProd-MLOps accounts:

o Prod-MLOps: Holds S3 buckets used for production ML detections where strict access controls and data residency policies are implemented.

o NonProd-MLOps: Retains development, testing and non-production ML detections so experimentations don't have any impact on production data integrity.

This separation is MLOps' standard best practice since it reduces the risk by keeping experimental models from affecting core production environments.

- Compliance of the data residency principle: the data residency principles specify that the data needs to stay within specified geographic areas for operational and regulatory purposes. AWS S3 multi-region design facilitates this compliance through the ability to host ML detection data in S3 buckets within specified regions so as to ensure regulatory compliances and optimize access latency across consumers in various regions.

With careful consideration of those needs and technical design selections, the Analytics Ingestion Service is structured to be fault-tolerant and scalable as a part of the larger MLOps infrastructure to serve both existing and future ML detection pipelines.

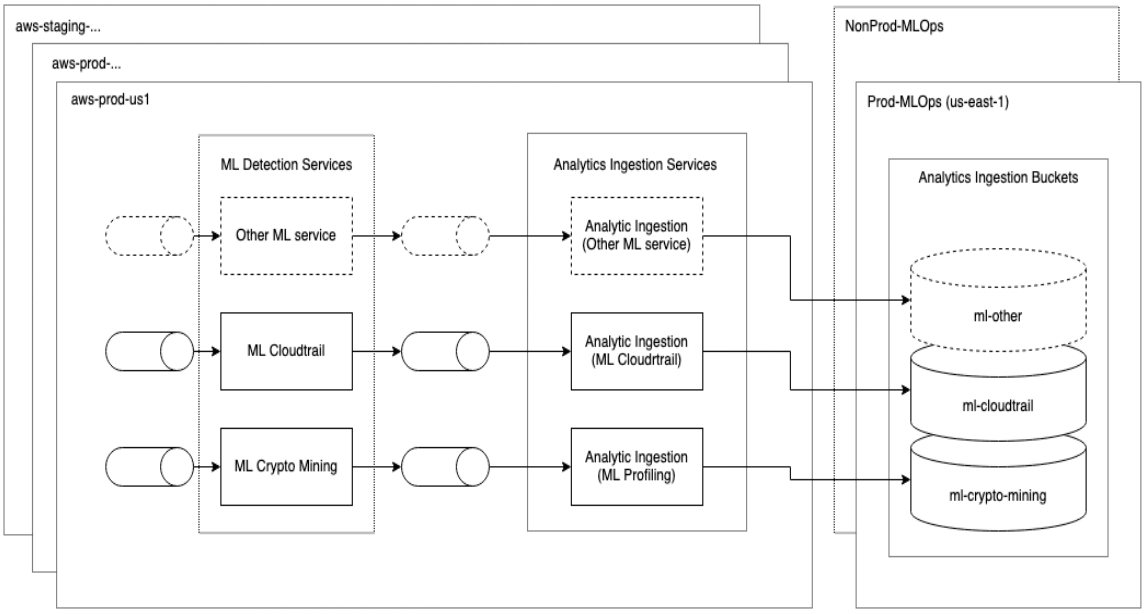

Figure 3: This picture is descriptive of the principal needs we have just stated, illustrating the various instances of AIS stacked up corresponding to every ML service, finally being deployed in various regions, such as the buckets holding the data in the long term.

3.2 Technological Stack

Before discussing the design and architectural choices of the Analytics Ingestion Service (AIS), it is essential to introduce the core technologies utilized in the project.

3.2.1 Protocol Buffers (Protobuf)

Protocol Buffers (Protobuf) is a languageless, platform independent mechanism to serialize structure statistics. It was created by Google's search engine and offers a simple and compact binary design that makes it easier to exchange information across dispersed networks.
Compared to the traditional JSON and XML formats, Protobuf has higher serialization efficiency, thereby reducing the size of the warhead and the cost of parsing.

Protobuf specifies data structures using.proto files, which specify the message layout using a strong typed syntax. This definition can be extended to different programming languages, such as Python, Java, and Go, which allow seamless integration in heterogeneous environments. The generated code provides effective serialization and deserialization systems to ensure interoperation and type safety.

One of the obligatory advantages of Protobuf is its assistance in the back and forth compatibility. Fields within message definitions can be added or deprecated without breaking existing implementations, making it particularly well-suited for evolving systems that require long-term maintainability.
Additionally, Protobuf integrates seamlessly with Remote Procedure Call (RPC) frameworks such as gRPC, allowing efficient communication in distributed architectures.

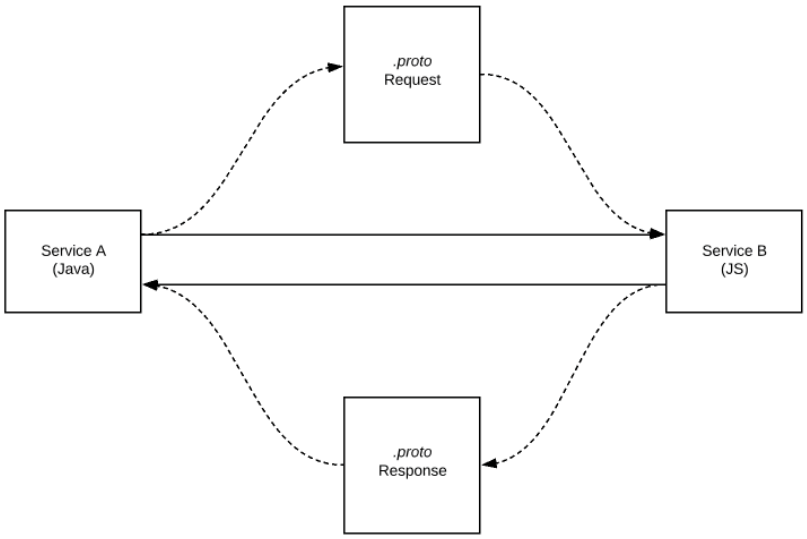

Figure 4: a gRPC-based interaction between a Java-based Service A and a Javascript-based Service B

Given its performance benefits and schema evolution capabilities, Protobuf is widely adopted in modern data-intensive applications, particularly in microservices architectures, event-driven systems, and machine learning pipelines. [20]

3.2.2 Helm

Helm is a Kubernetes package manager that simplifies the use, administration, and configuration of containers. Helm is a templating organization developed by Deis (subsequently acquired by Microsoft) and currently maintained as part of the Cloud Native Computing Foundation (CNCF), which enables the specification, packaging, and distribution of Kubernetes applications using a reusable and customizable configuration known as the Helm Chart.

A Helm chart encapsulates all the indispensable Kubernetes tools, such as deployment, services, Config maps, and Secrets, into a single package. This abstraction enables developers and operators to deploy applications systematically across different environments while minimizing configuration complexity. Parameters can be dynamically adjusted via “values.yaml” file, allowing for flexibility in modifying a Kubernetes manifest without changing its base.

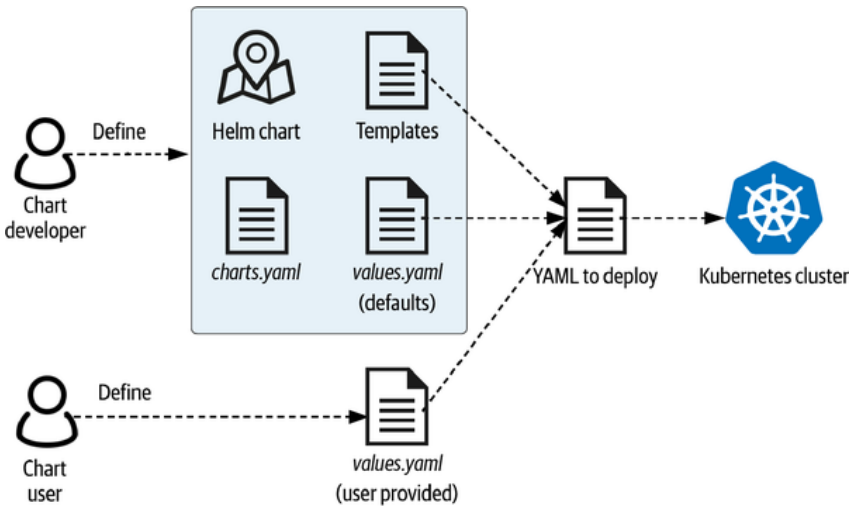

Figure 5: A typical Helm workflow, the chart developer defines reusable Helm charts with templates and default values, while the chart user customizes them via values.yaml to deploy resources to a Kubernetes cluster.

Helm also introduces release management, which monitors the deployment of the implementation and supports versioning, rollback, and upgrade procedures, so that changes can be safely used during the maintenance of the ability to return to previous nations if necessary. In addition, Helm integrates with the Helm repository, facilitating the allocation and sharing of pre-packaged functions, encouraging reuse and standardization in the cloud environment. [21]

Helm is widely accepted in the DevOps and MLOps workflow, facilitating automated, scalable, and maintainable network supervision. One's declarative and modular technique enhances the efficiency of handling complex tasks and ensures consistency across different groups and environments.

3.2.4 NATS

NATS is a high-performance, cloud-native messaging system designed for distributed systems, microservices, and real-time data streaming. It provides a lightweight, scalable, and reliable publish-subscribe communication mechanism that enables seamless data exchange between decoupled services.

At its core, NATS follows a simple and efficient architecture based on subjects and topics. Publishers send messages to a specific subject, and subscribers receive messages by subscribing to relevant subjects. This design ensures high throughput and low latency, making NATS well-suited for real-time applications.

One of the key advantages of NATS is its scalability. It supports horizontal scaling by

distributing messages across multiple nodes, ensuring high availability and fault tolerance. Additionally, it provides different messaging patterns, including:

- Publish-Subscribe: Multiple subscribers can receive messages published to a subject.
- Request-Reply: Enables synchronous communication between services, often used for microservices interactions.
- Queue Groups: A mechanism that allows multiple subscribers to join a queue group, where each message is delivered to only one subscriber within the group. This enables load balancing while preserving the benefits of the publish-subscribe model. [22]

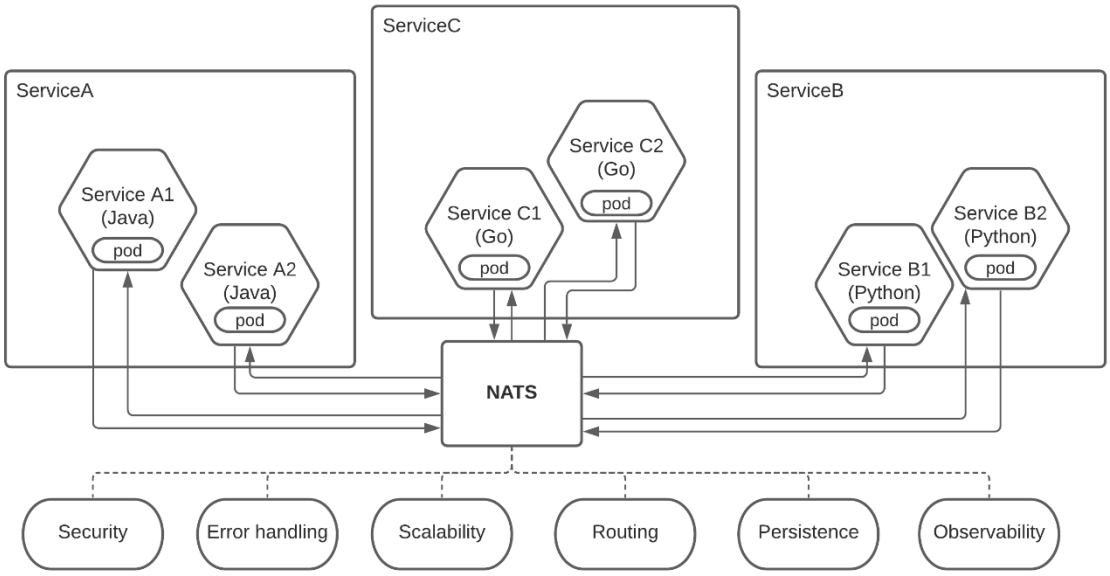

Figure *6*: Example of a microservice architecture using NATS for communication between services written in Java, Go, and Python for scalable and resilient distributed systems.

NATS is widely used in cloud-native and IoT ecosystems due to its lightweight footprint and ability to handle millions of messages per second with minimal overhead. It is often deployed in conjunction with container orchestration platforms like Kubernetes and integrates seamlessly with modern event-driven architectures.

3.2.5 AsyncIO Programming and queues

AsyncIO is a Python library that provides support for writing concurrent code using the async/await syntax. It enables non-blocking, event-driven programming, allowing applications to handle multiple I/O-bound tasks efficiently without relying on traditional multi-threading or multi-processing.

At its core, AsyncIO employs an event loop that schedules and executes coroutines, which are special functions defined with the async keyword. Unlike standard functions,

coroutines can pause execution at certain points (await) to allow other tasks to run, thereby improving performance in applications that require high concurrency, such as network servers, web scraping, and message processing. [23]

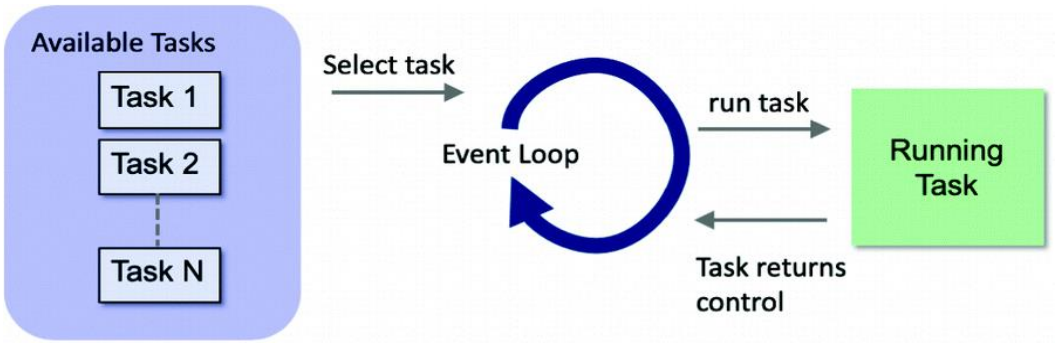

Figure 7: Event loop mechanism. Tasks are selected and run one at a time; when a task yields control, the loop continues, enabling asynchronous, non-blocking execution.

One of the key components of AsyncIO is its support for queues, which facilitate communication and data exchange between coroutines. AsyncIO queues operate similarly to traditional queue implementations but are designed for asynchronous workflows. The primary queue types include:

- asyncio.Queue: A first-in, first-out (FIFO) queue that allows coroutines to produce and consume messages asynchronously. It ensures safe access between multiple coroutines without requiring explicit locks.
- asyncio.PriorityQueue: A variant that orders elements based on priority, useful for task scheduling.
- asyncio.LifoQueue: A last-in, first-out (LIFO) queue, often used for stack-like behavior.

Queues in AsyncIO are widely used in event-driven systems, such as distributed processing pipelines and message brokers, where efficient coordination between producers and consumers is essential. Since these queues are non-blocking, they prevent bottlenecks in applications that need to handle high-throughput data streams.

3.2.6 AWS S3

Amazon Web Services (AWS) is a cloud computing platform that offers a wide range of infrastructure, storage, computing, and analytics services. It provides scalable, on-demand resources that help organizations to build and deploy applications efficiently without managing physical hardware. Some of the key services within AWS include:

- *Compute*: EC2 (Elastic Compute Cloud) for virtual machines, Lambda for serverless computing, and ECS/EKS for container orchestration.

- *Networking*: VPC (Virtual Private Cloud) for networking isolation, Route 53 for DNS management, and API Gateway for managing APIs.
- *Database*: RDS (Relational Database Service) for managed SQL databases, DynamoDB for NoSQL, and Redshift for data warehousing.
- *Storage*: S3 (Simple Storage Service) for object storage, EBS (Elastic Block Store) for persistent volumes, and Glacier for long-term archival storage.

Among these storage options, AWS S3 (Simple Storage Service) is one of the most widely used and fundamental storage services. It provides highly durable, scalable, and cost-effective object storage for a variety of use cases, including data lakes, backup and disaster recovery, big data analytics, and machine learning pipelines.

S3 is designed for 99.999999999% durability (11 nines), ensuring reliability and security. It supports:

- Object-based storage: unlike traditional block or file storage, S3 stores data as objects in buckets, making it ideal for large-scale unstructured data.
- Scalability: it automatically scales to handle massive data volumes, eliminating storage capacity concerns.
- Access Management: integration with IAM (Identity and Access Management) allows fine-grained access control, supporting policies, encryption, and compliance requirements.
- Storage Classes: different tiers like Standard, Intelligent-Tiering, Glacier, and Deep Archive allow cost optimization based on access frequency. [24]

Moreover, AWS S3 operates across multiple regions to ensure low-latency access and compliance with data residency regulations. Organizations can select specific AWS regions to store data within their legal jurisdiction, ensuring adherence to local data protection laws such as GDPR in Europe or CCPA in California.

3.2.7 Parquet

In Artificial Intelligence Systems, particularly those that operate over distributed architectures and handle large-scale data ingestion or feature extraction workflows, the choice of data storage format plays an important role in determining system efficiency, scalability, and throughput. Apache Parquet has emerged as a dominant storage format tailored to such needs, offering a highly optimized columnar data layout that fundamentally departs from traditional row-based storage mechanisms like CSV or JSON. The columnar design of Parquet enables it to store data by fields rather than by records, allowing analytical engines to access only the specific columns required for computation. This selective access capability significantly reduces input/output

operations and is particularly beneficial in machine learning pipelines where typically only a subset of features is relevant during model training or inference.

Internally, a Parquet file is structured into a nested hierarchy comprising row groups, column chunks, and pages. Each file begins and ends with a footer that encodes essential metadata, including the file schema, statistics, and column-level metadata. The data itself is divided into one or more row groups, which are horizontal partitions of the dataset. Each row group contains data for all columns within a bounded number of rows and is designed to be read in parallel by multiple processors. Within each row group, column chunks represent the physical storage of a single column's data, enabling compression and encoding strategies to be applied independently per column. These column chunks are further divided into pages, which are the smallest unit of data accessed during a read operation. Pages can contain data values, dictionaries for encoding repeated terms, and indexes for efficient navigation.
To sum up, advantages of Parquet include:

- Efficient compression: by storing similar values together, Parquet achieves high compression ratios, reducing storage costs and improving data retrieval efficiency.
- Optimized query performance: columnar storage enables faster analytical queries, as only the required columns are read instead of scanning entire rows.
- Schema evolution: it supports schema evolution, allowing modifications over time without requiring a complete rewrite of stored data.
- Compatibility: parquet is widely supported by data processing frameworks such as Apache Spark, Presto, Hive, and Pandas, making it a standard choice in data lakes and analytics platforms. [25]

When combined with object storage solutions like AWS S3, Parquet becomes an efficient way to store and manage large datasets. Storing Parquet files in S3 allows organizations to benefit from cost-effective, durable, and scalable storage while leveraging Parquet's optimized structure for analytical workloads. Additionally, S3 Select can be used to retrieve only specific parts of Parquet files, further enhancing performance by reducing the amount of data transferred.

3.3 System Architecture

In this chapter, we present a detailed explanation of every component of the Analytics Ingestion pipeline. This section describes how data flows through the system, highlighting the mechanisms that ensure efficient and reliable ingestion.

3.3.1 Architectural Design - Write Path

The Analytics Ingestion pipeline is responsible for collecting, batching, and storing predictions and logs generated by multiple machine learning services. The overall workflow is depicted in Figure 8, which outlines the various components and their interactions.

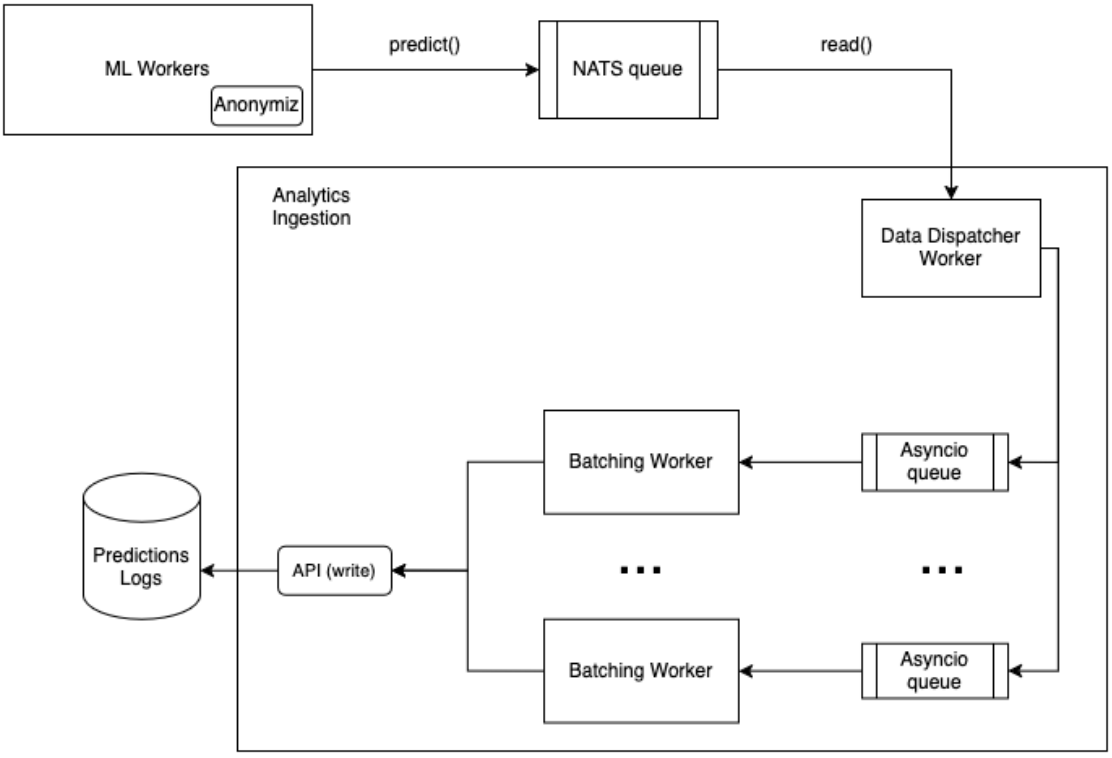

Figure *8*: The AIS architecture

The ingestion process starts with ML Workers, which include a critical Anonymization step, which has been implemented for the AIS project, since the data now are retained. This step ensures that customer data is encrypted before being forwarded to a NATS queue. The NATS queue serves as a message broker, temporarily holding all incoming prediction logs and facilitating the decoupling between ML model execution and data ingestion.

In a realistic deployment scenario, multiple ML services produce different types of prediction logs. To efficiently process these logs, a Data Dispatcher Worker is employed. Its primary function is to consume messages from the NATS queue and distribute them into separate in-memory queues (implemented using Asyncio queues) based on the corresponding message type. This ensures that different types of logs are

processed independently and efficiently.

The actual data ingestion into storage is managed by a pool of Batching Workers. These workers retrieve messages from their respective in-memory queues, aggregate them into batches, and perform efficient bulk writes into the storage system. The storage backend is implemented using an object store (e.g., S3), and the write operations are performed via optimized PyArrow APIs.

Even though these were not stringent requirements for the project, each component in the pipeline is designed to handle high-throughput and minimize processing latency.
In the following subsections, we provide in-depth explanations of the core implementation details of each component.

3.3.1.1 Anonymization Step

Before messages are pushed to the NATS queue from the ML Workers, an anonymization step is performed within the Analytics Ingestion System to safeguard sensitive information. This step makes sure that identifiable data, such as process names, customer identifiers, and other proprietary metadata—is obfuscated in a way that preserves analytical value without exposing private or regulated information.
The anonymization process leverages a deterministic cryptographic hashing technique, which allows the system to consistently generate the same anonymized output for identical inputs within the same environment, facilitating aggregation and trend analysis over time. The procedure involves the following steps:
1. A unique, environment-specific salt is generated and stored securely using SOPS (Secrets OPerationS), a tool designed to manage encrypted secrets. This ensures that salts are both confidential and tightly scoped to their respective deployment environments. [26]
2. The retrieved salt is concatenated with each piece of sensitive data to introduce uniqueness and prevent hash collisions or rainbow table attacks. The first ones occur when two different inputs produce the same hashed output. For the cited attack, rainbow tables are precomputed by mapping plaintext inputs to their hash outputs and are used by attackers to reverse-engineer hashed values. Salting the input disrupts this by making precomputation infeasible.
3. The concatenated string (salt + sensitive data) is then passed through the SHA-256 hashing algorithm, a secure, one-way cryptographic function from the SHA-2 family. This produces a fixed-length, irreversible hash that serves as the anonymized representation of the input.

4. The resulting anonymized values are then transmitted to the NATS message queue for downstream processing, ensuring that no raw sensitive data ever leaves the ML Workers boundary.
The process is resumed in the following pseudo-code of the anonymization algorithm:

Input: original_data (String)
Output: hashed_data (String)

1. Generate a 16-byte random salt using a secure generator
2. Securely store or retrieve the salt using SOPS
3. Convert original_data to bytes
4. Concatenate original_data_bytes with salt → data_with_salt
5. Compute SHA-256 hash of data_with_salt
6. Convert hash to hexadecimal format → hashed_data
7. Return hashed_data

With this approach, AIS maintains data privacy; the use of deterministic salts ensures reproducibility within a controlled scope, together with a reliable correlation of anonymized entities across different data capture events, without ever exposing the original identifiers.

3.3.1.2 Dispatcher Worker

The Dispatcher Worker is a key component in the Analytics Ingestion pipeline, responsible for consuming messages from the NATS queue, validating them, and efficiently distributing them into appropriate topic-based partitions for downstream processing. This worker ensures that different types of machine learning events are correctly classified and routed to the appropriate processing queues.

The Dispatcher Worker operates in an event-driven manner, continuously fetching messages from the NATS queue. When messages arrive, they are first decompressed using gzip, reducing network and storage overhead while ensuring efficient data transmission.
Once decompressed, messages are deserialized using Protocol Buffers (protobuf), specifically the IngestMLEvent proto message format has been defined for this scope. This message structure encapsulates metadata like customer_id, and timestamp_ns, along with a specific ML detection event (e.g., CryptoMiningDetection).
The worker extracts the type of event using the WhichOneof method of protobuf's python library, determining the correct processing logic, and, before processing, it

applies a validation step:

- If the event type is unrecognized, it is promptly rejected.
- If the message structure is not conform to the expected protocol buffer schema, with all required fields present and properly formatted, it is discarded to prevent corrupt data from propagating.

Subsequently, each valid event is classified based on its type. The Dispatcher Worker relies on a configuration management system (the WorkerConfigManager component) to determine the correct topic for each event type. Each event is then converted into an internal representation, ensuring that it conforms to downstream processing requirements.
The configuration includes the number of partitions and other attributes to support scalable event consumption.

A key step in this classification process is determining the partition key. The worker extracts a meaningful partition key from the event, ensuring that related events are processed together for consistency.
To distribute workload efficiently, the dispatcher partitions events based on key attributes that define logical separation within each topic. For instance, the CryptoMiningDetection events are partitioned using the following fields:

- region – The geographical area where the event originated, so that events from the same geographical area are processed together.
- customer_id – The identifier of the customer generating the event, so that all events from a single customer are consistently routed
- year, month, day, hour – Timestamps to group events in a structured time-based manner. This can facilitate downstream time-series analysis and efficient data retrieval.
- model_version – The version of the machine learning model used for detection, so that events processed by the same model version are grouped together.
- prediction – The classification result of the detection system, for efficient processing of events with same label.

Once classified, the event is dispatched to a Topic, a specialized abstraction that manages partitioned message queues. The Topic ensures that messages are evenly distributed across partitions with a dedicated channel for processing, while maintaining ordering guarantees within each partition, enabling the correct handling of different event streams.
This correct handling is enforced via a deterministic hashing function (SHA-256), which maps partition keys to specific partitions, so that events related to the same identifier consistently arrive in the same partition, optimizing state management.

The Topic structure consists of multiple in-memory partitions, each implemented as an asynchronous queue, using asyncio. These partitions prevent message overload by enforcing a maximum queue size, ensuring that backpressure mechanisms regulate excessive message flow.
Workers can then subscribe to specific partitions within a Topic, consuming messages asynchronously. If multiple workers subscribe, partitions are distributed in a load-balanced manner, improving scalability.

Another important implementation detail, is that the Dispatcher Worker incorporates several error-handling mechanisms:

- If a message is invalid, it is acknowledged and skipped, preventing unnecessary retries.
- If the NATS queue is temporarily unavailable, the worker handles timeouts and retries automatically.
- Every processed, skipped, or failed message is logged for traceability, ensuring that issues can be diagnosed.

To sum up, multiple dispatcher instances can operate simultaneously, with automatic load balancing across available workers. The system's configuration management allows for dynamic adjustment of partition counts and processing parameters without requiring system downtime. This architecture allows the Dispatcher Worker to efficiently route and organize events while maintaining system integrity, ensuring that the Analytics Ingestion pipeline remains efficient and robust.

3.3.1.3 Batching

The Batching Worker processes incoming events efficiently by grouping them into batches before writing them to the storage layer. This batching mechanism:

- optimizes resource utilization
- minimizes write operations
- ensures scalability in handling large event streams.

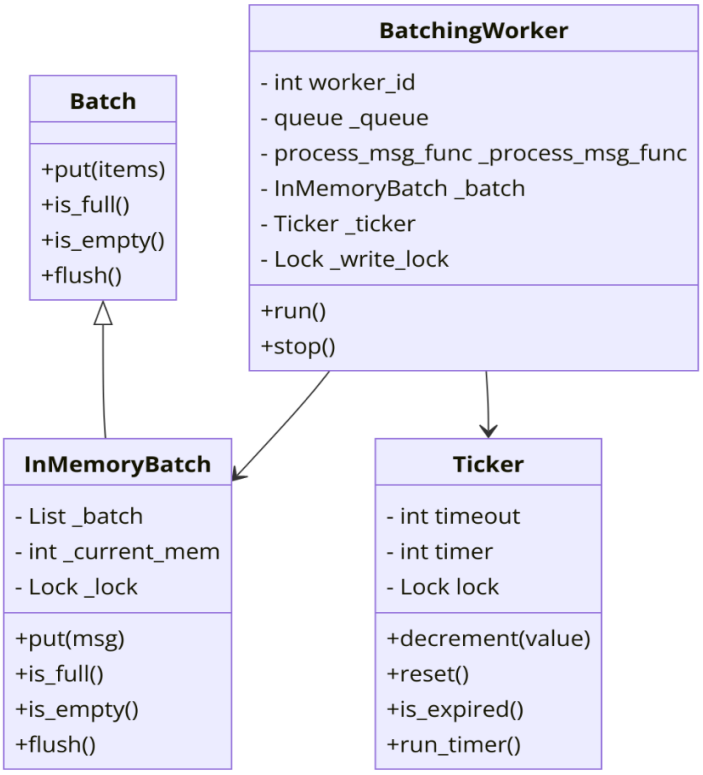

Figure 9: Class diagram showing the structural relationships between the core components involved in the batching process.

As we are going to see, the implementation balances multiple constraints, including batch size and memory usage.

The batching process is managed through an In-Memory Batch component, which temporarily stores events retrieved from the Dispatcher's asyncio queues, until predefined conditions trigger a write operation.
The batch enforces two primary constraints:

- Maximum batch size: A limit on the number of events stored in a single batch.
- Maximum memory usage: A threshold that ensures the batch does not exceed a predefined memory footprint.

Each event added to the batch increases the current memory usage based on a fixed per-message size. If either the size or memory limit is reached, the batch is flushed and processed.
To maintain consistency and prevent race conditions, we added a locking mechanism, which ensures that only one process can modify the batch at any given time.

In addition to the size and memory constraints, a time-based expiration mechanism

ensures that events are processed even if the batch has not yet reached its limits. This is managed through a Ticker, which decrements a timer at regular intervals. If the timer expires before the batch is full, the batch is flushed and written.
The ticker is reset after each batch flush, ensuring that events are not held indefinitely in low-volume scenarios. This has been done to prevent excessive latency in event processing while maintaining efficient batching behavior.

To recap, the Batching Worker continuously monitors incoming events and processes them according to the following sequence:

1. Events are dequeued from the Dispatcher queue and added to the in-memory batch.
2. If the batch exceeds the defined size or memory constraints, it is flushed immediately.
3. If the ticker signals that the maximum wait time has elapsed, the batch is written even if it has not reached its other limits.
4. Once a batch is written, both the batch and ticker are reset to prepare for the next cycle.

While having a good balance between latency and throughput, by dynamically adjusting to incoming event rates while avoiding excessive write operations, the worker continuously runs in an asynchronous loop, checking batch conditions and executing the necessary operations in a non-blocking manner.
In the next section, we will detail the writing mechanism that, as preannounced, follows the batching process.

3.3.1.4 Writer

The Writer component is responsible for persisting event data batches to durable storage after they have been prepared for export. Its design emphasizes scalability, fault tolerance, and modularity to accommodate different deployment scenarios and data volumes without compromising on performance or maintainability.

At its core, the Writer is designed to support a pluggable storage backend model. It dynamically selects the appropriate strategy based on the URI scheme of the target path, thereby abstracting the physical characteristics of the underlying storage medium. For example, write paths prefixed with “file://” are interpreted as local file destinations, while those starting with “s3://” indicate remote object storage using Amazon S3. This guarantees portability and ease of integration across environments, from on-premise setups to cloud-based pipelines.

To achieve consistent behavior across heterogeneous storage targets, the Writer

constructs a virtual file system abstraction by leveraging the filesystem interface provided by PyArrow, a high-performance Python library that provides a bridge between in-memory data structures and on-disk formats. PyArrow's use of Arrow's zero-copy memory model allows efficient memory layout and minimizes serialization overhead, making it ideal for large-scale, high-throughput data processing.
This abstraction encapsulates the specifics of the underlying backend, whether it is a local disk or a remote object store like Amazon S3, behind a common API for file operations. For S3, the system initializes a S3FileSystem object with explicit parameters such as region, access key, and endpoint, enabling direct interaction with the S3 protocol through optimized, low-level network primitives. For local storage, a lightweight LocalFileSystem implementation is used. This approach provides a clean separation of concerns: storage-specific details like authentication, protocol handling, and connection management are delegated to the filesystem layer, significantly reducing system complexity and increasing modularity.

For data serialization, the Writer processes each event by first transforming it into a normalized Python dictionary that conforms to a predefined schema. These dictionaries are then collectively converted into a RecordBatch, a memory-efficient structure provided by PyArrow that represents a table-like collection of rows with a shared schema. Unlike row-oriented formats, RecordBatch stores data in contiguous columnar buffers, allowing for better CPU cache utilization and SIMD-friendly processing. To prepare the batch for Parquet serialization, the system aggregates one or more RecordBatch instances into an Arrow Table, which provides a higher-level abstraction over columnar data and supports rich metadata, schema enforcement, and type-safe operations.
Once the data is represented as an Arrow Table, it is written to disk in the Parquet format. This format is chosen for its efficient compression, schema evolution support and interoperability with analytical steps.
To further accelerate downstream querying and analytics, the Writer organizes the output into logical partitions, typically based on temporal or categorical dimensions such as timestamps or event types. This enables selective scanning and parallel reads in distributed processing engines.

All in all, the write process follows a definite pipeline:

- The received batches are checked to ensure that every component fulfils the anticipated event schema.
- The events are serialized into Arrow-based forms for efficiency and compactness.
- The data is partitioned and stored as Parquet files based on defined partition keys, which provides optimized lookup performance.

- The associated metadata such as schema details and partition structure are preserved in order to aid in future evolution and retrieval efficiency.

3.1.4.3 Helm Configuration

As we already introduced it, Helm is a powerful package manager for Kubernetes that streamlines the deployment, management, and configuration of applications within a Kubernetes cluster. In the context of our project, Helm was leveraged to efficiently manage the deployment and scaling of the Analytics Ingestion Service within a containerized environment. The service needed to handle the ingestion of machine learning detection data at scale, and Helm provided an ideal solution for managing the deployment lifecycle, configuration, and consistency across various environments.
Here we outline the steps taken to configure Helm for the deployment of the Analytics Ingestion Service.

To deploy the Analytics Ingestion Service, we began by creating a custom Helm chart. The chart serves as a package that defines the deployment of all required Kubernetes resources, including pods, services, deployments, and configuration files. This chart encapsulates the specific configurations necessary for AIS to interact with ML detection queues, manage data ingestion, and handle storage integration.
Key configurations included:

- Deployment Specifications: these defined the number of replicas, resource requests and limits, and environment-specific configurations, such as NATS queue endpoints and storage bucket references, enabling the service to write data to the appropriate storage buckets without hardcoding paths or credentials directly into the code.
- Service Definitions: by exposing the necessary services to allow inter-service communication within the Kubernetes cluster.
- ConfigMaps and Secrets: used to securely manage and inject environment variables, such as credentials for accessing queues and storage systems, into the AIS containers.

By integrating Kubernetes secrets into the Helm deployment process, sensitive information, such as API keys and credentials for external services, could be injected securely into the containers during deployment. This ensured that sensitive data was never exposed in the configuration files or code repositories, maintaining compliance with data protection and security best practices.

Helm also helped with the modifications made to CPU and memory allocation to match different deployment stages. For example, during the staging phase, fewer resources might be allocated to reduce costs, while production environments could be configured

for high availability with more robust resource allocation.
Moreover, given that data ingestion may vary in load, we used Helm's ability to scale pods based on performance metrics. Horizontal Pod Autoscalers (HPA) were configured to adjust the number of running pods based on resource usage, ensuring that the ingestion service could handle varying amounts of incoming detection data without overloading resources.

To sum up, by leveraging Helm's templating, customization through values files, and management of resources and secrets, we were able to streamline the deployment process keeping flexibility and facilitating both the scalability and maintainability of the Analytics Ingestion Service across multiple production and non-production environments.

3.4 Analytics Tasks

In the previous chapters, we have explored the write path of our data pipeline up to the point where predictions and logs are collected and stored in the central Logs S3 Bucket. This chapter introduces the analytical layer, which builds upon that foundation to extract meaningful insights, detect anomalies, and guide future decisions based on real-world data. The architecture diagram in figure 10 illustrates our long-term vision for the analytics system.

Ideally, the analytical part of the pipeline would comprise the following key components:

- Models Comparator Worker: this performs periodic A/B testing between different model versions deployed in the environments. It generates evaluation reports and stores them for future inspection. This comparison helps track model performance for models promotions in production environments, and detect regressions or improvements over time.
- Analytics Worker: A general-purpose component capable of handling various tasks using the collected prediction data. It performs analytics operations such as:
  - Outlier Detection
  - Drift Detections

These analyses can then result in further actions, such as:

- Alerting via communication channels, e.g., Slack
- Triggering model retraining workflows when performance degradation or significant drift is observed.

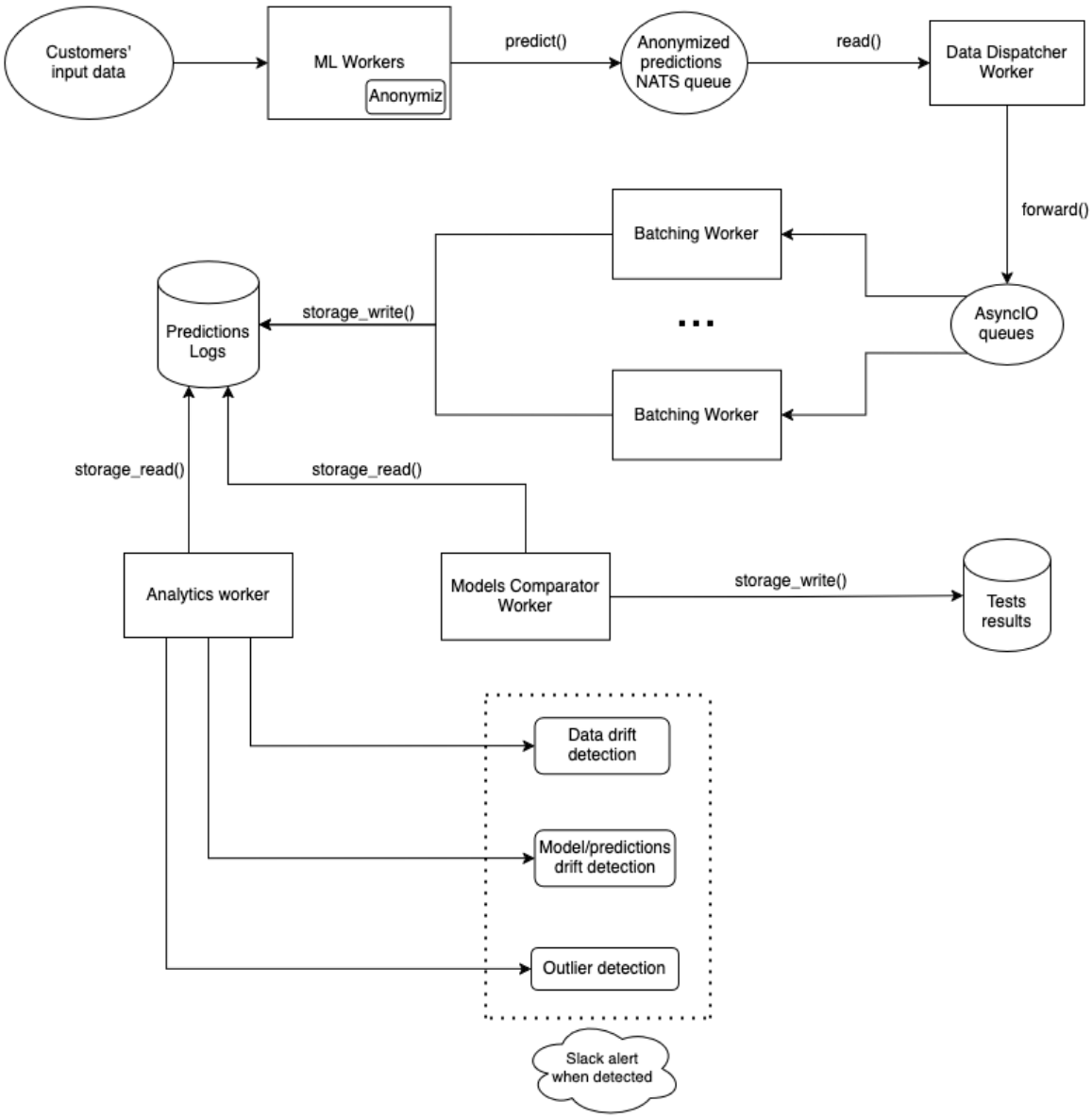

Figure 10: The long-term vision for the pipeline.

For the purposes of this thesis project, we have implemented some demos of the workflow that the Analytics workers should tackle, to demonstrate their utility in real-world company scenarios. These were needed to showcase how actual ingested data can be leveraged to perform drift detection and A/B testing.

In the following sections, we will dive deeper into the specific analytical tasks implemented and techniques used, and the practical results obtained from running them on real data.

3.4.1 Data and Concept Drift

3.4.1.1 Data Drift

Data drift refers to a significant shift in the distribution of the input data over time, which can lead to performance degradation in machine learning models. Essentially, it occurs when the statistical properties of the input data in production differ from those in the training data, potentially rendering the model's predictions less accurate or unreliable. This can happen for various reasons such as changes in user behavior, seasonal effects, shifts in market trends, or technical changes in the data collection process.

To monitor data drift, we need to track the changes in the statistical properties of the features over time. This is typically achieved by continuously evaluating the incoming data stream and comparing it to the reference distribution from the training phase. The key to detecting data drift is to identify whether certain properties of features, like mean, standard deviation, and frequency distribution, have changed significantly.

For continuous features, techniques such as Kullback-Leibler Divergence, Kolmogorov-Smirnov Statistics, and Hellinger Distance are used. These metrics allow for the comparison of probability distributions between two datasets, helping to quantify the degree of change in the feature distribution. Specifically:

- Kullback-Leibler Divergence measures the difference between two probability distributions, highlighting how much information is lost when approximating one distribution with another.
- Kolmogorov-Smirnov Statistics tests whether two samples come from the same distribution, useful when comparing historical and current feature distributions.
- Hellinger Distance provides a measure of the similarity between two probability distributions, effectively quantifying how much the distribution of a feature has changed.

For categorical features, monitoring can focus on the cardinality (number of distinct values) and frequency distribution of categories. Here, statistical tests such as the Chi-Squared Test, and metrics like Entropy can be used.

The Chi-Squared test compares the observed frequency distribution of a feature with the expected distribution, helping to detect shifts in categorical data. Entropy quantifies the randomness in categorical feature distributions, providing insight into whether certain categories have become more or less common.

By monitoring these aspects of the data continuously, we can identify any signs of drift and take action to retrain the model or adjust the feature engineering process before model performance suffers significantly.

We will describe the methodology applied in our demonstration in another chapter.

3.4.1.2 Concept Drift

While data drift refers to changes in the distribution of input features, concept drift involves changes in the underlying relationship between the input features and the target variable over time. In other words, it occurs when the statistical relationship that the model has learned between inputs and outputs no longer holds true in production. Concept drift can be particularly problematic for predictive models, as the model may continue to rely on outdated patterns that no longer accurately represent the real-world scenario.
Concept drift typically manifests when external factors affect the decision boundary of the model, such as shifts in user behavior, regulatory changes, economic fluctuations, or evolving patterns in the data that influence the predictions. This drift is harder to detect than data drift, as it requires identifying a change not just in the data but in the relationship between the features and the target variable.
Detecting concept drift involves comparing model predictions over time and assessing how they align with the expected outcomes. We can monitor model performance and track metrics such as Hellinger Distance, Kullback-Leibler Divergence, and Population Stability Index (PSI) to assess whether the predicted probabilities have diverged from the expected distribution. These metrics can be calculated over different time windows, helping to identify whether the model's predictions are becoming less reliable.
PSI tracks changes in the distribution of predicted probabilities over time, identifying whether certain segments of the predicted population have shifted in a way that requires attention.

3.4.2 A/B Testing of models

In this section, we refer to the process of performing A/B testing on the ingested data to validate a release candidate model. A/B testing in this context is essential to evaluate the performance of new models in a production environment, especially given that models deployed in production often experience shifts in data distribution that can degrade their performance.

Machine learning models are generally trained in offline environments, using historical data, before being deployed into production. However, when a model is deployed in a live, production environment, it may encounter changes in the data; hence A/B testing allows us to assess whether a new model is better suited to the current state of production data compared to an existing model.
In the context of the Analytics Ingestion Service, we plan to use A/B testing to evaluate the performance of a newly deployed model (often referred to as the sidecar model) alongside the primary model already in production. The goal is to compare the models'

performance using specific key performance indicators (KPIs) that are relevant to the model's intended task, such as detecting false positives and accurately recalling miners in the case of a crypto mining detection model.
For example, to compare the models, for the mining detection task we focus on the following KPIs:

- False Positives (FP): The number of times a model incorrectly identifies a non-mining activity as mining.
- Recall: The ability of the model to correctly identify actual mining activities, ensuring that legitimate miners are captured by the model.

Once these metrics are defined, A/B testing can be conducted to assess whether the sidecar model performs statistically better than the primary model. This involves using hypothesis testing to ensure the observed differences are not due to random chance.
To confirm that the performance differences between the models are statistically significant, we can apply several hypothesis tests. Two common approaches for this in the context of A/B testing are the paired t-test and McNemar's test.
We will review both the methodologies and select one for our purpose.

3.4.2.1 Paired t-test

The paired t-test, also known as the dependent samples t-test, is a statistical method used to test whether the mean difference between paired measurements is zero. In the context of model comparison, we would compare the performance of the two models (primary and sidecar) on the same dataset by using their predictions (e.g., probability of being a miner). If we take the before-and-after predictions for each sample, we can apply the paired t-test to assess whether the observed differences between the models are statistically significant.
To do this, we analyze the difference in predicted probabilities for the same set of samples under both models. By computing the mean of these differences and measuring their variability, we can assess whether the observed deviation is likely due to random fluctuations or represents a meaningful performance shift.
Paired t-test is used to evaluate whether the average difference between the two models is significantly different from zero. This test relies on the assumption that the distribution of differences follows a normal distribution, and it accounts for both the magnitude of deviations and their consistency across the dataset. Since the test is paired, each sample's prediction under the primary model is directly compared to its prediction under the sidecar model, rather than treating the two sets of predictions as independent.

To conduct the test, we first calculate the difference in predicted probabilities for each sample i, defined as:

$$d_i = p_{\{i,A\}} - p_{\{i,B\}}$$

Next, we compute the mean of these differences to understand the overall direction and magnitude of the discrepancy between the models:

$$\bar{d} = \frac{1}{n}\sum_{\{i=1\}}^{n} d_i$$

where n is the total number of paired samples.

To measure the variability in the differences across all samples, we calculate the sample standard deviation of the differences:

$$s_d = \sqrt{\frac{1}{n-1}\sum_{\{i=1\}}^{n}\left(d_i - \bar{d}\right)^2}$$

This tells us how consistent or variable the model differences are from one prediction to another.

With these quantities in hand, we compute the test statistic for the paired t-test:

$$t = \frac{\bar{d}}{\frac{s_d}{\sqrt{n}}}$$

This statistic follows a Student's t-distribution with n - 1 degrees of freedom.

Once the test statistic is computed, it is compared against a reference distribution to determine the probability of obtaining such a result under the assumption that the models have no real performance difference. This probability, known as the p-value, indicates whether we can reject the null hypothesis, which assumes that both models perform identically. If the p-value falls below a predefined significance threshold (commonly 0.05), we conclude that the models have statistically different behaviors. Otherwise, we lack sufficient evidence to claim a meaningful distinction. [27]

We must bear in mind that the paired t-test assumes that the measurements are paired, the subjects (samples) are independent, and the differences in predictions are normally distributed. Violations of these assumptions can render the test unreliable, especially if the differences are not normally distributed. In such cases, alternative tests may be more appropriate.

3.4.2.2 McNemar's Test

For categorical data, such as whether a model correctly identifies or fails to identify mining activity, McNemar's test is often used. This non-parametric test is designed for paired nominal data and assesses whether there is a significant difference in the error rates between the two models.

McNemar's test is based on a contingency table that records how the two models classify

the data. The table compares the outcomes of the models on each data sample, categorized into four groups:

Number of examples misclassified by both model A and B	Number of examples misclassified by model A but not by B
Number of examples misclassified by B but not by A	Number of examples misclassified by neither model A or B

We can better name them like:

n_{00}	n_{01}
n_{10}	n_{11}

The null hypothesis of McNemar's test is that the two models have the same error rate. The McNemar test statistic is computed as:

$\chi^2 = \frac{(|n_{01}-n_{10}|-1)^2}{n_{01}+n_{10}}$,

where the continuity correction (-1 in the numerator) adjusts for the discrete nature of the test, improving accuracy for small sample sizes. Hence, the test is based on a chi-squared distribution with one degree of freedom, and if the p-value is below a certain significance level (e.g., 0.05), we can reject the null hypothesis and conclude that the models have significantly different performances. [28]

McNemar's test is particularly useful because it has a lower risk of type I errors (false positives), meaning it is less likely to incorrectly conclude that there is a difference between the models when none exists. [29] Moreover, unlike the paired t-test, it does not rely on assumptions such as normality of the differences.

Based on the result of the test and KPIs, decisions can be made regarding whether to promote the sidecar model to full production status or make further adjustments.

3.4.3 Analytics Detections demo

As cited previously, we are going to describe some experiments that we conducted to demonstrate the application of drift detection techniques to real data ingested with our analytics pipeline. The data was collected from a time when there was a suspicion of data drift.

The dataset used for these tests consists of high-dimensional event-level analytics logs for mining detection, structured in a tabular format with 139 columns. Of these, 5

columns contain metadata (e.g., identifiers and timestamps), 1 column represents the binary prediction target, 13 columns are numerical features, and the remaining 120 are boolean features encoding system and behavioral signals.

While the write path of these logs is not subject to strict latency constraints, a potential analytics worker in place for the detection of drift would rely on fetching and processing historical analytics data as quickly as possible to ensure timely monitoring of data, so it is essential to optimize the read path to analytical data. For this reason, we benchmarked PyArrow.

In our implementation, feature-rich event logs are stored in Amazon S3 using partitioned Parquet format. Each file is relatively small (~120 KB) and contains high-dimensional records with 139 columns. For benchmarking, we evaluated the ingestion of a filtered subset amounting to 4,936 rows distributed across approximately 110 Parquet files.
In production on-premises environments, data processing workloads typically run on nodes with 8–16 vCPUs and 32–64 GB RAM per worker. To match this capacity in a managed cloud setting, for our experiments we selected the ml.m5.4xlarge instance for SageMaker Studio.
This instance provides 16 vCPUs and 64 GB of memory, offering similar performance to a typical on-premise data processing node.

Reading the dataset with PyArrow took ~40.5 seconds. This latency is primarily due to the high overhead of sequentially accessing a large number of small Parquet files over S3, which introduces cumulative I/O and network latency. However, once loaded, in-memory operations such as .count() were extremely fast (~0.01 seconds), reflecting the efficiency of pandas-based computation on moderate-sized data held in memory. Hence, to optimize ingestion performance in S3-based architectures, both the number and size of Parquet files and processing framework must be carefully considered.

3.4.3.2 Drift Detection Implementation and Results

In order to operationalize the data and concept drift theoretical notions presented in the first four sections, we used the following statistical detection pipeline, specifically geared toward a real-case anomaly identified in production. Namely, we had reason to suspect drift when we saw that one specific sub-group of miners was not being appropriately picked up by the model anymore. To check our hypothesis, we performed a comparison analysis between the miner-related examples of the initial training set and a sub-sample of miner-related records pulled out of the analytics logs which we had described in the paragraph before. Such ingestion was done by applying filters to the

analytics data over relevant fields to separate out the interesting examples. From ingested data, we extracted 48 miner samples in order to be compared to 122 miner samples belonging to the training set of production.
Our dataset consists of both numerical and boolean features. The purpose of the analysis was to see whether the feature distributions in the data being produced had changed significantly in comparison to the training data, hence implying data or concept drift.

To detect changes in the distributions of features, we applied statistical tests appropriate to each data type:

- Numerical Features: for continuous variables, we used the two-sample Kolmogorov-Smirnov (KS) test, which evaluates whether two samples originate from the same continuous distribution. The null hypothesis states that both samples are drawn from the same distribution. The KS statistic is defined as:

$$D_{\{n,m\}} = \sup_x \left| F_{n(x)} - G_{m(x)} \right|$$

where $F_{n(x)}$ and $G_{m(x)}$ are the empirical distribution functions (EDFs) of the two samples, based on sample sizes n and m, respectively.
The KS statistic captures the maximum absolute difference between the two empirical cumulative distribution functions (ECDFs) of the samples. This means that we look for the single point where the discrepancy between the cumulative frequencies of the two samples is greatest. A large value of this difference suggests that the distributions differ in shape, location, or scale.
We compute a p-value corresponding to this statistic, which quantifies the probability of observing such a difference (or a more extreme one) under the null hypothesis that both samples are drawn from the same underlying distribution. If the p-value falls below a predefined significance level, commonly 0.05, we reject the null hypothesis. This statistical rejection indicates a significant difference in the distributions, providing evidence of a shift or drift in the feature's behavior over time.

- Boolean Features: For binary categorical variables, we applied the Chi-Squared Test of Independence using 2×2 contingency tables. These tables are constructed by counting the frequency of each possible value (0 or 1) for the feature in both the training and production datasets. The Chi-Squared Test is designed to determine whether there is a statistically significant association between two categorical variables—in our case, the binary feature value and the dataset origin (training or production). The null hypothesis assumes that the distribution of feature values is independent of the dataset source, meaning any observed difference in frequencies is due to random variation. By computing a test statistic that measures the deviation between observed and expected frequencies under this null hypothesis, we assess whether the two variables are independent.

So, we construct a contingency table where each cell (i, j) contains the observed frequency $O_{i,j}$ of a particular feature value i (e.g., 0 or 1) in a dataset j (training or production). We then compute the expected frequency E_{ij} for each cell, assuming independence between feature value and dataset source:

$$E_{\{ij\}} = \frac{\sum_{j=1}^{c} O_{ij} * \sum_{j=1}^{r} O_{ij}}{\sum_{i=1}^{r} \sum_{j=1}^{c} O_{ij}}$$

This tells us what frequency we would expect in each cell if the feature value and the dataset source were truly independent.
The Chi-Squared test statistic aggregates the deviations between observed and expected frequencies using the formula:

$$\chi^2 = \sum_{i=1}^{r} \sum_{j=1}^{c} \frac{\left(O_{\{ij\}} - E_{\{ij\}}\right)^2}{E_{\{ij\}}}$$

Where:

- $O_{\{ij\}}$ is the observed count in cell (i, j)
- $E_{\{ij\}}$ is the expected count in that same cell,
- r is the number of rows (feature values),
- c is the number of columns (dataset sources).

Now we can notice how this statistic measures how much the actual frequencies deviate from what we'd expect if there were no association. The more the observed counts differ from expected counts, the higher the test statistic.
This statistic follows a Chi-Squared distribution with $(r-1)(c-1)$ degrees of freedom.

We then compute the p-value, under the assumption that the null hypothesis is true. A small p-value (typically less than 0.05) leads us to reject the null hypothesis, proving that the feature's distribution differs significantly between training and production, i.e., drift has occurred.

3.4.3.3 Numerical Feature Results

The statistical testing for the numerical features was conducted using the *ks_2samp* function from the *scipy.stats* module, which implements the two-sample Kolmogorov–Smirnov test to assess distributional differences.

The following table summarizes the outcomes of the KS tests:

Feature	p-value	Drifted
numerical_feature_1	1.000	No
numerical_feature_2	1.000	No
numerical_feature_3	1.000	No
numerical_feature_4	0.705	No
numerical_feature_5	0.035	Yes
numerical_feature_6	1.000	No
numerical_feature_7	0.852	No
numerical_feature_8	0.000	Yes
numerical_feature_9	0.000	Yes
numerical_feature_10	0.011	Yes
numerical_feature_11	0.000	Yes
numerical_feature_12	1.000	No
numerical_feature_13	0.000	Yes

The results show that over half of the numerical features experienced significant distributional changes. This confirms the presence of data drift in the continuous domain.

3.4.3.4 Boolean Feature Results

For the boolean features, contingency tables were constructed using *pandas.crosstab*, and the Chi-squared test for independence was performed via the *chi2_contingency* function from the *scipy.stats* module.
These are the results of the tests:

Feature	p-value	Drifted
boolean_feature_1	0.001	Yes
boolean_feature_2	0.002	Yes
boolean_feature_3	0.1	No
boolean_feature_4	0.955	No
boolean_feature_5	1.000	No
boolean_feature_6	0.000	Yes
boolean_feature_7	1.000	No

boolean_feature_8	1.000	No
boolean_feature_9	0.184	No
boolean_feature_10	0.275	No

The detection pipeline identified drift in 5 out of 10 boolean features, signaling notable changes in binary behavioral signals.

3.4.3.5 Drift visualization

To better highlight these results, we create Kernel Density Estimation (KDE) plots of each of the quantitative features. KDE is one form of non-parametric estimation of the probability density function (PDF) of a continuous variable. It provides smooth estimation of the histogram of the data without being tied to fixed bin boundaries, trying to reveal the shape of the data's underlying distribution.

The kernel function $K(u)$ is one of the fundamental parts of Kernel Density Estimation. The shape of the weighting function to be used when smoothing the observed data is determined by it. It controls the degree of influence each observed point x_i should place on the density estimate at a certain location. Overall, the kernel function must meet the following requirements:

1. $K(u) \geq 0 \text{ for all } u \in \mathbb{R}$
2. $\int_{\{-\infty\}}^{\{\infty\}} K(u)du = 1$
3. $K(u) = K(-u)$

A commonly used kernel is the Gaussian kernel, which is infinitely differentiable and gives higher weights to points closer to the target location. It is defined as:

$$K(u) = \frac{1}{\sqrt{2\pi}} e^{-\frac{u^2}{2}}$$

In KDE, the kernel function is applied to the normalized distance between a target point x and each data point x_i, scaled by a bandwidth parameter h, which controls the smoothness of the estimate. The full KDE formula is:

$$\hat{f}_{h(x)} = \frac{1}{n * h} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right)$$

where:

- $\hat{f}_{h(x)}$ is the estimated probability density at point xxx,
- n is the number of observations,
- h is the bandwidth or smoothing parameter,
- $K(*)$ is the kernel function.

The KDE procedure works as follows:

1. For a given point x, compute the distance $x - x_i$ from each data point,
2. Scale this distance by dividing by the bandwidth h,
3. Apply the kernel function to the scaled value $\frac{x - x_i}{h}$,
4. Average the resulting kernel values across all data points to get the estimate $\hat{f}_{h(x)}$.

This process is repeated over many values of x (typically across a fine grid) to generate a smooth density curve.

By generating the KDE plots, and by using histograms as plots for the Boolean features counterpart, we were able to confirm the presence of drift across both numerical and boolean domains and support the hypothesis that some features' statistical properties have changed between training and production, like the statistical tests revealed. This justified further model diagnostics and potential retraining.

Some relevant examples are shown in the next images:

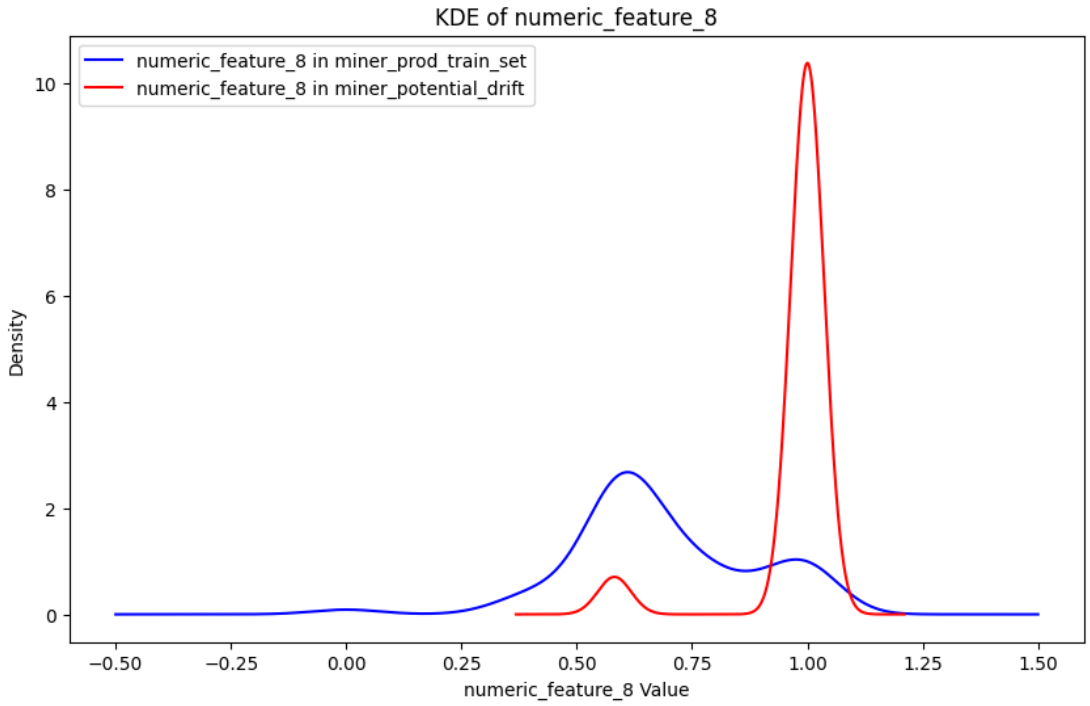

Figure *11*: this feature had a p-value approximated to zero. As we can see the distributions are nowhere similar.

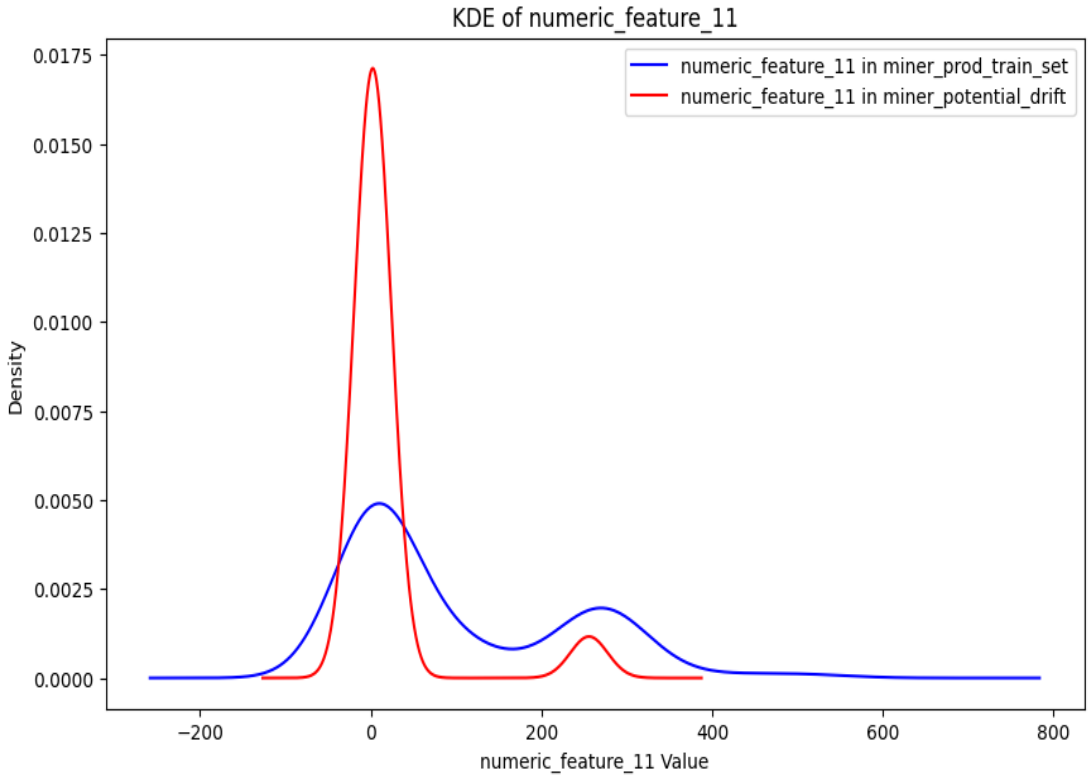

Figure *12*: similar distributions with p-value = 0.705. Both curves show primary peaks around 0 with comparable shapes, though the red curve (new set) has a slightly more pronounced secondary peak around 1, consistent with no significant drift detected.

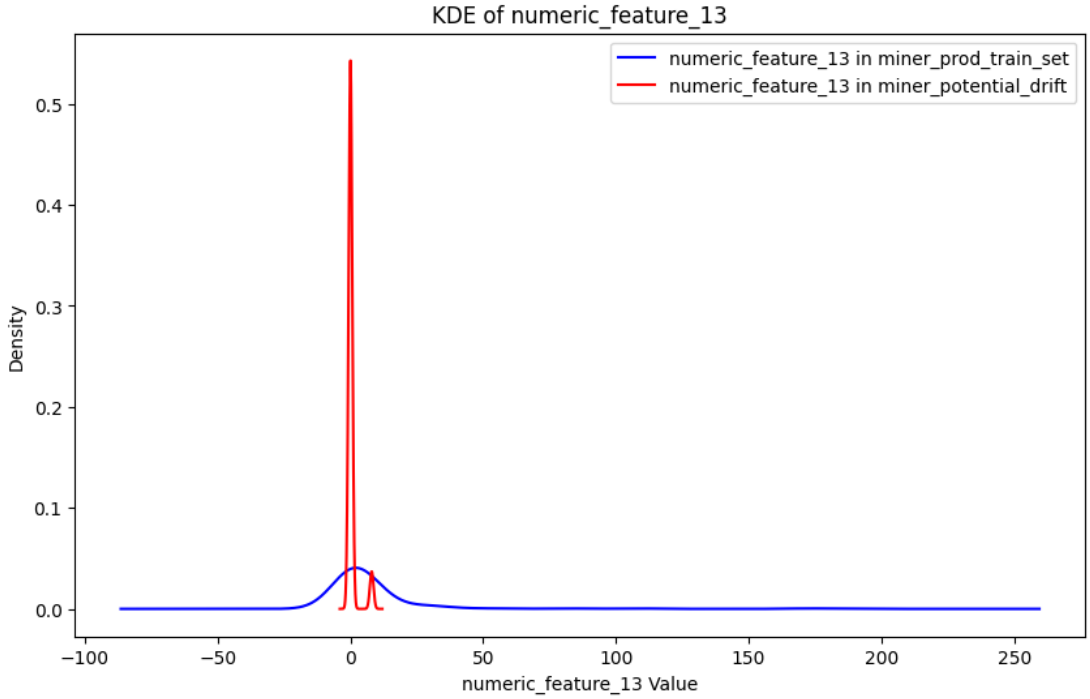

Figure *13*: here we can notice a drastic distribution shift with p-value < 0.001. The red curve (drift set) forms an extremely narrow spike at 0, while the blue curve (training set) shows a much broader, gentler distribution, clearly

indicating severe drift

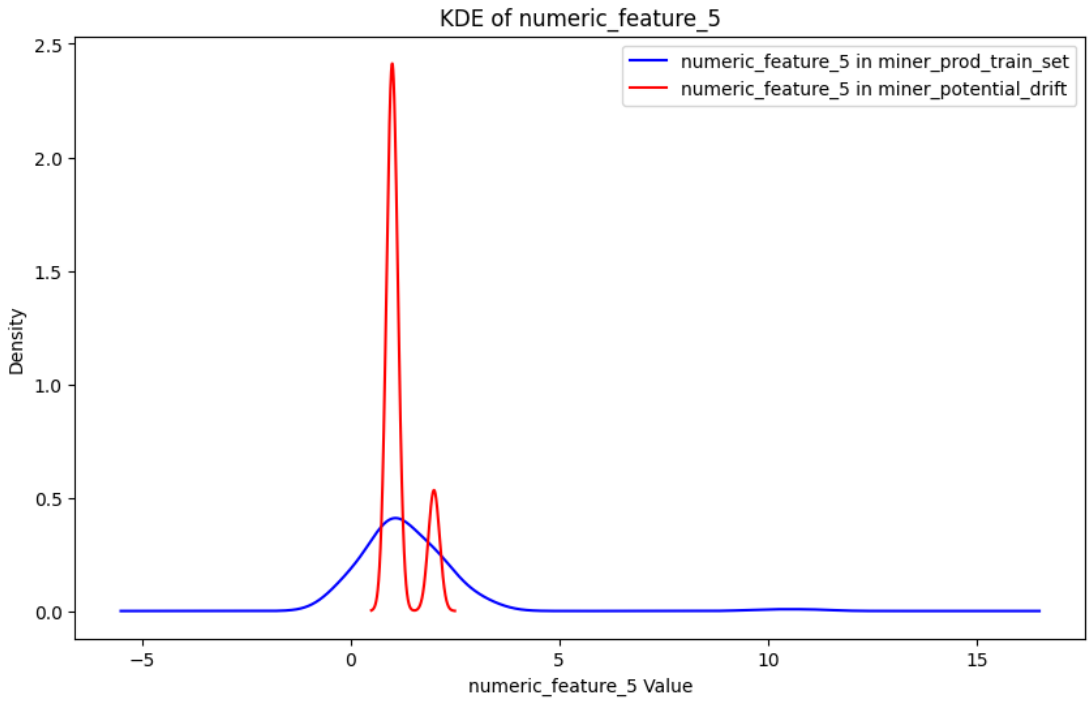

Figure *14*: similar situation to feature 13 but only slightly, as confirmed by the higher p-value = 0.035.

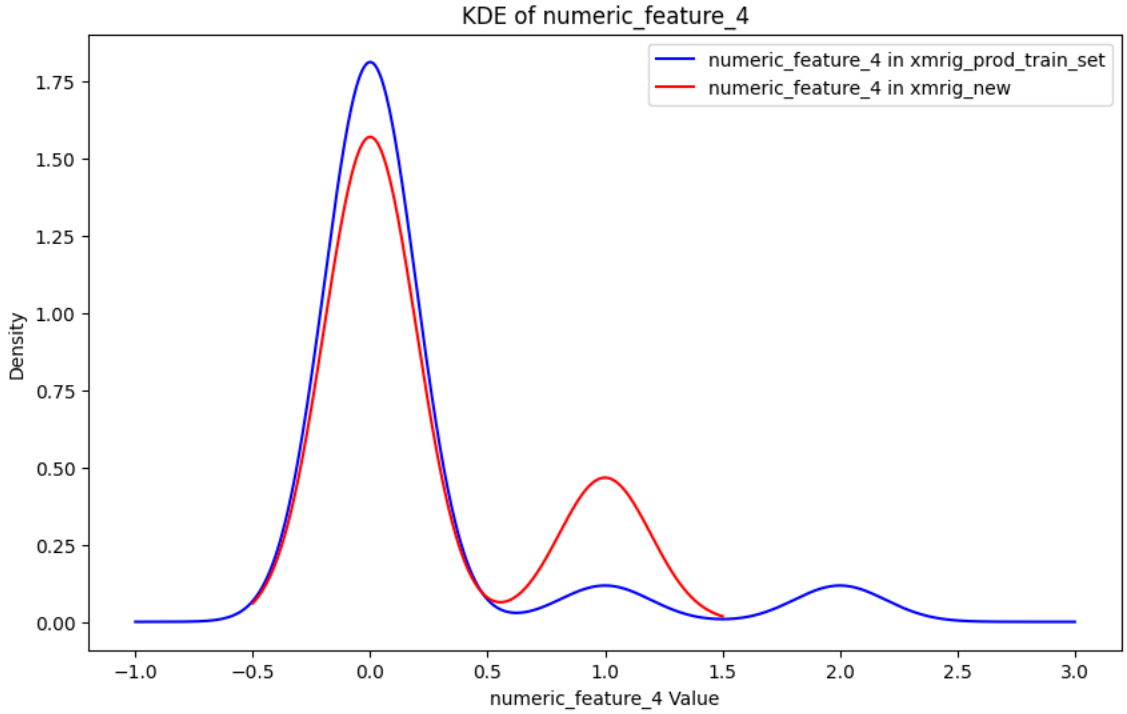

Figure *15*: here instead we illustrate two similar distributions with p-value = 0.705. Both curves show primary peaks around 0 with comparable shapes, though the red curve (new set) has a slightly more pronounced secondary peak around 1, consistent with no significant drift detected.

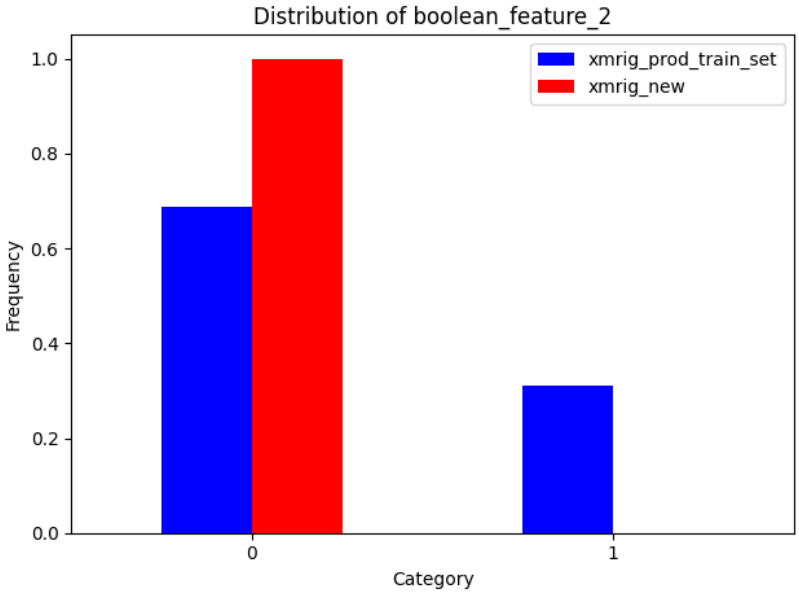

Figure 6: Distribution comparison showing significant drift with p-value = 0.002. The red bars (new set) show complete concentration in category 0 (100%), while the blue bars (training set) display a 70%-30% split between categories 0 and 1, indicating substantial distributional shift.

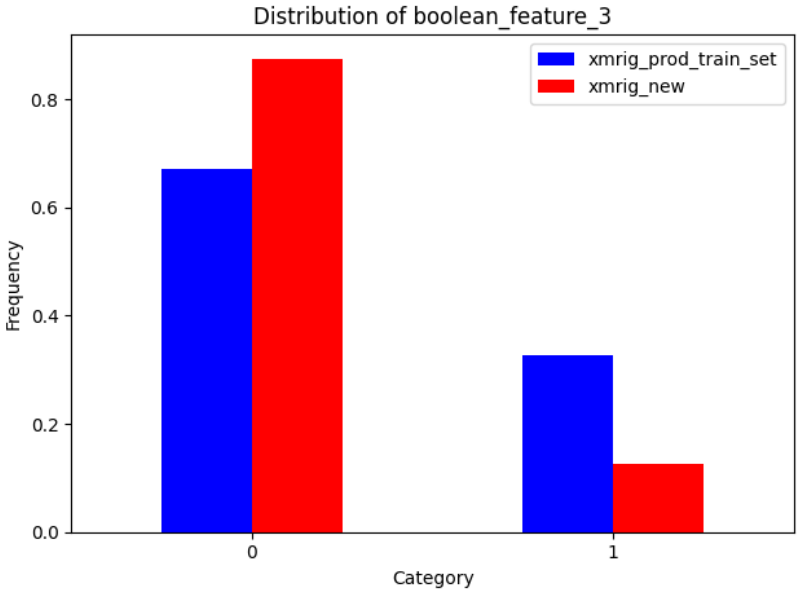

Figure *17*: The p-value is not that high (0.1), hence some changes are present, indeed, the red bars (new set) show higher concentration in category 0 (~87%) compared to blue bars (training set) at ~67%, but the differences are not statistically significant enough to indicate drift.

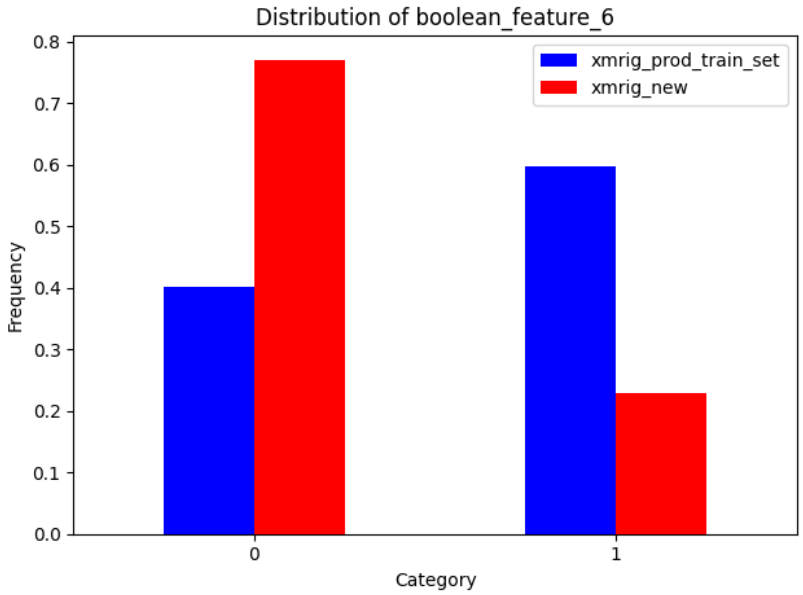

Figure 7: Distribution comparison showing significant drift with p-value < 0.001. They display opposite patterns, blue bars (training set) show 40%-60% split favoring category 1, while red bars (new set) show 77%-23% split favoring category 0.

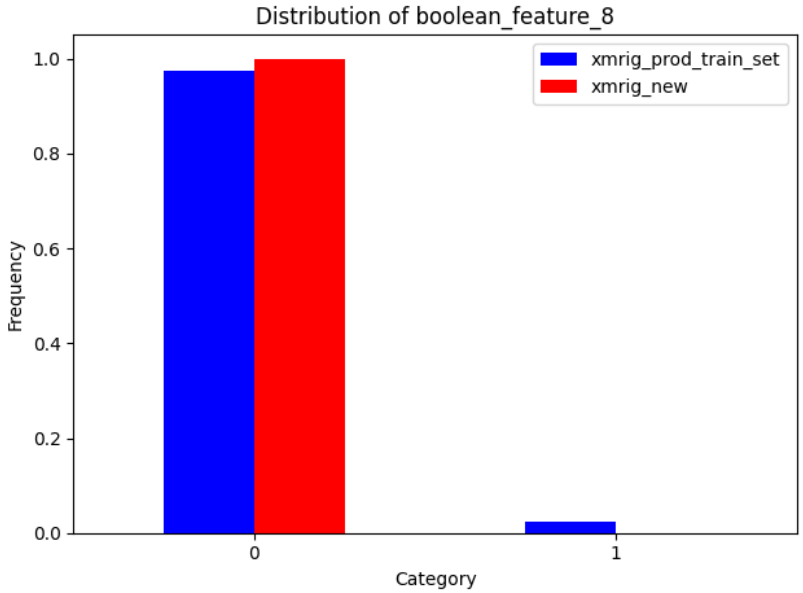

Figure 8: No drift with p-value = 1.000. Both distributions show nearly identical patterns with very high concentration in category 0 (~97% vs ~100%) and minimal presence in category 1 (~3% vs ~0%), confirming strong statistical similarity.

3.4.3.6 Experiment final considerations

The drift detection example illustrated the power of coupling statistical hypothesis testing with visualization in revealing emergent data dynamics in a production setting.

Through an application of the Kolmogorov–Smirnov test to continuous features and the Chi-squared test to boolean features, we quantified notable shifts in both spaces. Statistical results were supported by data visualizations: Kernel Density Estimation plots pointed to both small and significant shifts in the shape and central location of continuous feature distributions, with histograms giving a concise comparative overview of proportions of binary features.
The alignment of statistical and visual findings reinforces the conclusion that a set of features, both numeric and boolean, exhibited drift, reflecting that the miner data's features shifted since model training. This not only confirms the suspicion provoked by the unexpected model performance during the selected period but also highlights the importance of ongoing monitoring and possible retraining of models in response to changing operating conditions.
Since the statistical tests and visual analyses were conducted independently for each feature used in the ML model, as we started to consider in previous paragraphs, this framework could be extended into an automated monitoring system. A dedicated worker process could routinely compute drift metrics and trigger model retraining when predefined thresholds are exceeded. For instance, retraining could be initiated when more than 30% of the model's input features show statistically significant drift (e.g., $p < 0.01$) within a given time window.

3.4.4 A/B Testing demo

Whenever, after some round of testing in a controlled environment, a new candidate model is deployed in production, to evaluate the effectiveness for the mining activity detection, we need to conduct a structured A/B testing against the current primary model. We are going to illustrate a real situation in which we needed to discriminate among two different model in a definite period of time.

In our setup, two models were tested simultaneously on the same incoming data:

- Model A (Primary Model): the actual production model responsible for handling detection requests in real-time.
- Model B (Sidecar Model): a release-candidate deployed in parallel with the primary model to shadow its inferences without influencing production behavior.

This setup permitted us to gather predictions from both models for each sample in a real-world environment without affecting system performance. The purpose of this test was to determine whether the differences we got with Model B are statistically significant and, therefore, if it is improving the performance of the detection system, it should be promoted to production.

The test was conducted over a controlled observation period in which both models worked through the same 6,630 data examples. All were mining-related detection events with pre-existing ground-truth labels based on either manual validation or past classification.
We first made a 2 x 2 contingency table based on the two models' predictions on each sample. In this table, rows and columns represent whether a model predicted correctly or incorrectly with respect to a true label.
The table encompasses four possible results:

- Two models are correct
- Both models are in error
- Model A is right and Model B is incorrect
- Model B is correct and Model A is wrong

In our test, the table of contingencies had the following:

	Model B Correct	Model B Incorrect
Model A Correct	6511	2
Model A Incorrect	73	44

This reveals that:

- In 6,511 cases, both models made correct predictions.
- In 44 cases, both models were incorrect.
- In 2 cases, Model A was correct while Model B was not.
- In 73 cases, Model B was correct while Model A was not.

The most interesting part for McNemar's test lies in the disagreement cases (i.e., the off-diagonal counts).

Given the contingency table, we applied McNemar’s test using the chi-squared approximation with continuity correction. This version of the test is appropriate when the number of discordant cases is moderate to large, as in our case.
The continuity correction adjusts the test statistic to account for the discrete nature of

the data and reduce the risk of overestimating significance.

The test statistic was computed as:

$$\chi 2 = \frac{(|b - c| - 1)^2}{b + c}$$

Where:

- b=2: Model A correct, Model B incorrect
- c=73: Model B correct, Model A incorrect

Substituting these values into the formula yields a test statistic of approximately 65.33. The p-value associated with this chi-squared value (with 1 degree of freedom) was approximately:

$$p = 6.32 \times 10^{-16}$$

The p-value is orders of magnitude below the typical significance threshold of 0.05, leading us to reject the null hypothesis. This provides strong statistical evidence that the performance difference between the models is not due to chance.

More specifically in this case Model B (sidecar) made significantly more correct predictions in cases where Model A failed than vice versa.

Based on the outcome of McNemar's test, we conclude that the sidecar model demonstrates statistically superior performance on the same input data and ground-truth labels. As a result, it is a strong candidate for promotion to production, replacing the existing model.

The test proves to be both efficient and reliable, allowing us to:

- Safely test improvements without risking production stability,
- Make evidence-based model upgrade decisions,
- Minimize Type I errors by applying McNemar's test

Chapter 4

Conclusions and future developments

This thesis presented the design and implementation of the Analytics Ingestion System (AIS), a production-ready architecture for capturing, persisting, and analyzing machine learning model outputs in cybersecurity environments. The system was developed to meet a concrete operational need at Sysdig: transforming model predictions, typically transient and underutilized, into durable analytical assets that support robust post-deployment observability. Unlike traditional MLOps pipelines that focus primarily on model training and deployment, AIS focuses on the post-deployment lifecycle, where real-world predictions, input features, and detection decisions become the foundation for performance analysis, model validation, and behavior monitoring. Through this work, we built an end-to-end infrastructure capable of reliably ingesting high-volume inference logs, asynchronously processing them for persistence, and enabling downstream analytics workflows. This included the development of a flexible queue-based architecture, metadata-enriched logging interfaces, and integration points for statistical testing and comparative evaluation. The system is tightly aligned with Sysdig's operational requirements and establishes a critical building block for long-term machine learning governance in adversarial and rapidly evolving threat environments.

And so we developed the Analytics Ingestion Service as a flexible setup that can be added to any existing machine learning workflow without causing disruptions. AIS is made up of separate, container-based components that handle data processing, transforming it, and sending it through different stages. Customer data comes into the main ML services, where predictions are made and sensitive information is anonymized first. After this, the anonymized data goes into a message queue designed to handle lots of messages reliably. From there, multiple batching workers pick up these messages and save them in a prediction log, where each entry has a unique ID, features, timestamps, and model version info. This setup keeps prediction generation and data storage separated, which helps with scaling and reduces how much the systems rely on each other. At the same time, we have a data dispatcher worker that gathers new data and labels and sends them to queues for processing. These queues allow different services, like evaluation and testing tools, to work independently and at the same time.

Beyond building the AIS system, this thesis conducted concrete experiments to demonstrate the analytical value of collected inference data, particularly through the use of statistical testing. These experiments laid out how statistical tools can be applied to ingested detections to uncover significant data shifts and validate improvements in model performance. Specifically, we used the Kolmogorov–Smirnov test for continuous features and the Chi-squared test for categorical ones to identify distributional shifts over time. These findings were reinforced through Kernel Density Estimation (KDE) plots and histograms, which offered intuitive visual confirmation of the statistical results. This dual analysis approach, statistical and visual, not only confirmed that the miner data (detection data used for testing) had evolved since the model was trained, but also demonstrated how monitoring can inform decisions around retraining and model maintenance.

In another experiment, we evaluated the effectiveness of a candidate model against the current production model using McNemar's test for A/B testing, a robust non-parametric method ideal for comparing classifiers on paired data. By running the two models in a sidecar setup and comparing their predictions against shared ground truth labels, we were able to quantify and interpret performance differences in a statistical way. The test returned a p-value of 6.32×10^{-16}, strongly rejecting the null hypothesis and confirming that the candidate model outperformed the incumbent. This result validated the power of statistical analysis in guiding model upgrade decisions without disrupting production workflows.

Looking ahead, one of the key future developments for AIS involves evolving the architecture into a fully operational analytics worker. This component, envisioned as a persistent service, would continuously analyze model outputs in real time, detect data or concept drift, and flag anomalies using the same statistical techniques applied in our experiments. By integrating with communication platforms such as Slack, it could deliver alerts automatically, allowing teams to respond quickly to changes in model

behavior or data quality. We also foresee extending AIS with automated retraining triggers, such as activating a retraining pipeline when over 30% of input features exhibit statistically significant drift within a predefined window, thereby closing the loop between detection and response. These enhancements would transform AIS from a passive data collector into a proactive monitoring and governance tool for production ML and AI systems.

Designed specifically for Sysdig's machine learning and AI needs, AIS fits right into the company's existing systems, allowing for a good view of predictive services and models on a larger scale. It helps create safer, clearer, and more adaptable AI processes, giving Sysdig a nice edge in keeping high-quality, reliable AI up and running.

Bibliografy

[1] o. emesoronye, « Rule-Based vs. Machine Learning-Based Cybersecurity: Understanding the Differences,» 2024. [Online]. Available: https://www.linkedin.com/pulse/rule-based-vs-machine-learning-based-cybersecurity-obinna-emesoronye-lnj1e/.

[2] «Heuristic Detection,» [Online]. Available: https://multilogin.com/glossary/heuristic-detection/.

[3] E. A. Ö. A. S. K. I. (. I. I. S. A. I. B. MERVE OZKAN-OKAY, «A Comprehensive Survey: Evaluating the Efficiency of Artificial Intelligence and Machine Learning Techniques on Cyber Security Solutions,» January 2024. [Online].

[4] Mohamed, «Artificial intelligence and machine learning in cybersecurity: a deep dive into state-of-the-art techniques and future paradigms,» 2025. [Online]. Available: https://doi.org/10.1007/s10115-025-02429-y.

[5] Pattanayak, Suprit Kumar, «Unsupervised Learning for Anomaly Detection in Cybersecurity,» 2024. [Online]. Available: https://www.researchgate.net/publication/387470903_Unsupervised_Learning_for_Anomaly_Detection_in_Cybersecurity.

[6] Google, "TensorFlow Extended," [Online]. Available: https://www.tensorflow.org/tfx/guide?hl=it.

[7] Google, «Simulated Spotify Listening Experiences for Reinforcement Learning with TensorFlow and TF-Agents,» [Online]. Available: https://blog.tensorflow.org/2023/10/simulated-spotify-listening-experiences-reinforcement-learning-tensorflow-tf-agents.html.

[8] Kubernetes, «Kubeflow - Architecture,» [Online]. Available: https://www.kubeflow.org/docs/started/architecture/#kubeflow-ecosystem.

[9] Apache Spark, «Structured Streaming Programming Guide,» [Online]. Available: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html.

[10] Apache Flink, «What is flink,» [Online]. Available: https://flink.apache.org/what-is-flink/use-cases/.

[11] P. M. G. C. P. P. R. João Gama, «Learning with Drift Detection,» 2004. [Online]. Available: https://doi.org/10.1007/978-3-540-28645-5_29.

[12] j. wang, «DDM Drift Detection Method for Concept Drift Detection,» [Online]. Available: https://jimmy-wang-gen-ai.medium.com/ddm-drift-detection-method-for-concept-drafting-detection-94da0cf69841.

[13] J. D. C.-A. R. F. A. B. R. G. R. M.-B. Manuel Baena-Garcia, «Early Drift Detection Method,» 2006.

[14] d. C.-Á. J. R.-J. G. e. a. Frías-Blanco I, «Online and non-parametric drift detection methods based on Hoeffding's bounds.,» January 2015. [Online]. Available: https://doi.org/10.1109/TKDE.2014.2345382.

[15] S. G. Z. C. L. Stephan Rabanser, «Failing Loudly: An Empirical Study of Methods for Detecting Dataset Shift,» 2018. [Online]. Available: https://doi.org/10.48550/arXiv.1810.11953.

[16] M. K. Tegjyot Singh Sethi, «On the Reliable Detection of Concept Drift from Streaming Unlabeled Data,» March 2017. [Online]. Available: https://doi.org/10.48550/arXiv.1704.00023.

[17] R. L. R. Kohavi, «Online Controlled Experiments and A/B Testing,» 2017. [Online]. Available: https://doi.org/10.1007/978-1-4899-7687-1_891.

[18] «Model Evaluation 4: Algorithm Comparisons,» 2018. [Online]. Available: https://sebastianraschka.com/pdf/lecture-notes/stat479fs18/11_eval-algo_slides.pdf.

[19] «What is multi armed bandit,» [Online]. Available: https://www.omniconvert.com/what-is/multi-armed-bandit/.

[20] Protobuf, «Protobuf Overview,» [Online]. Available: https://protobuf.dev/overview/.

[21] «Learn Cloud Native: What is Helm in Kubernetes?,» [Online]. Available: https://sysdig.com/learn-cloud-native/what-is-helm-in-kubernetes/.

[22] «Introduction to NATs [Part 1] : A lightweight messaging platform,» [Online]. Available: https://medium.com/@ansabiqbal/introduction-to-nats-part-1-a-lightweight-messaging-platform-fbcf1d462823.

[23] «Async IO in Python: A Complete Walkthrough,» [Online]. Available: https://realpython.com/async-io-python/.

[24] «How Amazon S3 Achieves 99.999999999% Durability?,» May 2024. [Online]. Available: https://www.geeksforgeeks.org/how-amazon-s3-achieves-99-999999999-durability/.

[25] «What is the Parquet File Format? Use Cases & Benefits,» April 2023. [Online]. Available: https://www.upsolver.com/blog/apache-parquet-why-use.

[26] «SOPS,» [Online]. Available: https://github.com/getsops/sops.

[27] «Paired T-Test,» [Online]. Available: https://www.statisticssolutions.com/free-resources/directory-of-statistical-analyses/paired-sample-t-test/.

[28] Wikipedia, «McNemar's test,» [Online]. Available: https://en.wikipedia.org/wiki/McNemar%27s_test.

[29] T. G. Dietterich, «Approximate Statistical Tests for Comparing Supervised Classification Learning Algorithms,» October 1998. [Online]. Available: https://doi.org/10.1162/089976698300017197.