A MLOps Architecture for Near Real-time Distributed Stream Learning Operation Deployment

Miguel G. Rodrigues*, Eduardo K. Viegas*, Altair O. Santin*, Fabricio Enembreck*
* Pontificia Universidade Catolica do Parana (PUCPR)
Graduate Program in Computer Science (PPGIa), Brazil
{miguel.rodrigues, eduardo.viegas, santin, fabricio.enembreck}@ppgia.pucpr.br

***Abstract*—Traditional architectures for implementing Machine Learning Operations (MLOps) usually struggle to cope with the demands of Stream Learning (SL) environments, where deployed models must be incrementally updated at scale and in near real-time to handle a constantly evolving data stream. This paper proposes a new distributed architecture adapted for deploying and updating SL models under the MLOps framework, implemented twofold. First, we structure the core components as microservices deployed on a container orchestration environment, ensuring low computational overhead and high scalability. Second, we propose a periodic model versioning strategy that facilitates seamless updates of SL models without degrading system accuracy. By leveraging the inherent characteristics of SL algorithms, we trigger the model versioning task only when their decision boundaries undergo significant adjustments. This allows our architecture to support scalable inference while handling incremental SL updates, enabling high throughput and model accuracy in production settings. Experiments conducted on a proposal's prototype implemented as a distributed microservice architecture on Kubernetes attested to our scheme's feasibility. Our architecture can scale inference throughput as needed, delivering updated SL models in less than 2.5 seconds, supporting up to 8 inference endpoints while maintaining accuracy similar to traditional single-endpoint setups.**

***Index Terms*—MLOps, Stream Learning, Kubernetes, Microservices.**

I. Introduction

THE Machine Learning (ML) is a branch of Artificial Intelligence (AI) that enables computing systems to learn from data without explicit programming [1]. The applications of ML range from process automation and chatbots to supply chain optimization, targeted marketing, cybersecurity, and autonomous vehicles. It involves creating models that identify and generalize patterns in data to predict new, unseen inputs [2]. Supervised learning, the most common ML application, builds models on labeled datasets to solve classification or regression problems, where the goal is to categorize data or predict continuous values [3]. Building and deploying ML models typically involves several steps. These include data collection, preprocessing, model training, validation, deployment, performance monitoring, and updates to maintain accuracy.

Production management of a ML model, referred to as Machine Learning Operations (MLOps), is an iterative task involving multiple disciplines that require adequately managing the processes involved [4]. It is a collaborative approach that streamlines the development task by automating workflows, managing model versions, standardizing code, and ensuring scalability, ultimately improving security and compliance in production environments [5]. In this context, a critical MLOps phase is model deployment, called inference, where trained models predict new data, often remotely accessed by multiple users at scale, as in credit card fraud detection and recommendation systems [6]. To support such requirements, distributed architectures are built using tools such as MLflow [7] for tracking and versioning, AWS SageMaker [8], and Google Cloud AI Platform [9] for providing ML frameworks, and technologies such as microservices and containerization for adaptive system design [10].

A typical MLOps framework must implement four key components, namely *inference*, *model update*, *versioning*, and *monitoring* modules. The *inference* module loads ML models, handles queries, performs predictions, and returns results. The *model update* module manages retraining. It uses labeled events at scheduled intervals to ensure the deployed model remains current and reliable. The *versioning* module stores model versions generated during training and retraining, making them available to inference modules when needed. Finally, the *monitoring* module evaluates the performance of the deployed model in production, comparing it to new versions and allowing it to be replaced if an improved version is identified [5].

Current MLOps architectures generally meet the needs of traditional ML scenarios where data patterns evolve incrementally. However, traditional MLOps practices may fall short in Stream Learning (SL) environments, as they are not designed to handle incremental updates or to deliver updated models in very short time frames. SL refers to ML algorithms that continuously update models with incoming data streams without having to perform multiple passes over the data [11]. These data streams often come from various sources and are characterized by unbounded size, high and variable arrival rates, changing data patterns, and memory constraints during processing [12]. These characteristics present challenges that may not be adequately addressed by traditional MLOps. In a SL-enabled architecture, the inference process must handle many requests and often relies on parallelism techniques to manage extensive data flows [13]. While similar to traditional MLOps systems, a SL architecture uniquely requires constantly replacing models in memory with the latest versions, posing significant challenges, particularly when the number of endpoints or the model replacement rate is high [14].

Frequent model versioning in SL is challenging because loading new model versions to the endpoints and serializing those versions must be done quickly [15]. The versioning engine must also be able to efficiently manage the large number of model versions generated by these frequent updates. In contrast to traditional MLOps, where updates are typically performed in batches or mini-batches, model updates in SL environments need to be incremental and continuous. Ensuring the model remains available to inference endpoints is not easily achieved when their parameters are updated with each new event. This is because endpoints require significant time and computational resources to deserialize the updated model before inference can proceed, which can disrupt the inference service [16].

Surprisingly, there is a lack of research on developing new SL-oriented architectures that address the challenges of near real-time model updates within the MLOps framework [5], [17]. Distributed MLOps architectures for SL face the challenge of performing model updates in near real-time, ensuring that new events are efficiently incorporated into the model without disrupting inference performance. This requires continuous, incremental model updates with minimal latency while maintaining the overall system's responsiveness and scalability across distributed infrastructure [18]. Current literature generally focuses on scaling either model updates or inference, often overlooking their integration within an MLOps framework [19]. This becomes particularly challenging when model versioning is introduced, as frequent model updates typically halt inference to perform model deserialization. As a result, the impact of continuous model updates on the system's predictive performance over time is often only partially addressed or overlooked in existing studies [20], [21]. Most approaches rely on traditional MLOps update procedures that perform periodic batch-oriented updates, which can ultimately result in performance degradation [22]. Other approaches use concept drift detection strategies, triggering updates only after data pattern changes or model performance declines are detected [23]. Thus, these methods often render the updated model obsolete upon deployment and typically require continuous expert supervision. As a result, MLOps adoption for SL tasks in production environments has been hindered by the scarcity of literature and the absence of tools and procedures that support continuous inference and model updates on data streams.

Contribution. Our work paves the way for a scalable architecture for deploying SL models in production environments under the MLOps framework. The goal of the proposed scheme is to incrementally update the SL models and make them available with a configurable frequency while simultaneously handling the inference requests of the users at scale in a decentralized manner. The implementation of our proposed architecture is twofold. First, its core components are implemented as microservices in a container orchestration environment. This allows for low computational cost, high portability, and adaptability. Second, a dynamic model versioning approach is introduced. This enables new SL models to be made available to inference endpoints without affecting the accuracy performance of the system. Our approach leverages the inherent characteristics of SL algorithms by triggering the model versioning task only when significant adjustments occur in their decision boundaries. As a result, our scheme enables horizontal scaling of the inference module and supports SL incremental updates and versioning through incoming data flows. Each component of the architecture is designed to provide a high inference throughput without compromising the accuracy of the resulting model.

In summary, the main contributions of the paper are as follows:

- A new distributed architecture that enables SL-oriented MLOps deployment at scale. Our proposed scheme offers a configurable deployment frequency for new models to inference endpoints without affecting the overall accuracy and inference throughput performance;
- A proposal prototype implemented as microservices running on a container orchestration environment. Our architecture can scale the inference throughput as needed, delivering updated SL models in less than 2.5 seconds, supporting up to 8 inference endpoints, while maintaining an accuracy similar to that of traditional single endpoint setups

Roadmap. The remainder of this paper is organized as follows. Section II describes the fundamentals behind SL and MLOps. Section III overviews the related works on MLOps implementation. Section IV describes our proposed solution, Section V introduces our prototype, and Section VI evaluates its performance. Finally, Section VII concludes our work.

II. Preliminaries

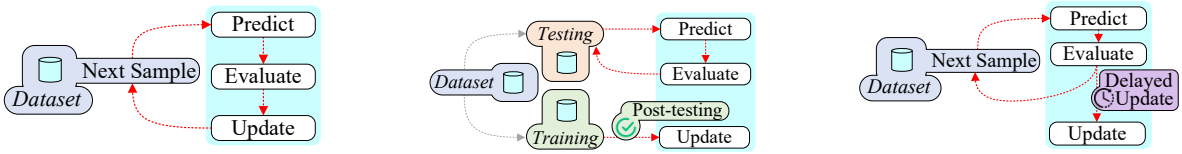
Deployment of SL techniques in production environments has increased substantially in recent years. This section covers the fundamentals of SL, followed by an overview of MLOps operations for SL-oriented systems. Finally, we review the challenges traditional MLOps implementations typically face in SL environments.

A. Stream Learning (SL)

In contrast to traditional batch-oriented ML, where models are static and built using often immutable historical data, SL involves data that is constantly changing its patterns in response to changes in the environment [23]. In a streaming environment, data is generated in a continuous, ordered flow that is potentially infinite and is transmitted at high speed, with examples arriving as a stream of data S.

Let $x_i \in X$ be an event, where $x_i \in \mathbb{R}^D$ denotes a D-dimensional feature vector input for the i^{th} event. The goal of the SL system is to continuously update a classifier function $f(x) : x \rightarrow y$ that outputs the predicted class y for a given input feature vector x. Here, the SL classifier function is usually updated incrementally such that it minimizes the resulting model's error rate. In practice, individual events must make model updates without access to past events.

This is because the data in these scenarios can only be accessed once or retained for a short time, so each step of SL must be performed quickly and resource-efficiently [24]. Since SL models update with each new training instance, they



(a) Prequential (b) Holdout (c) Delayed

Fig. 1: Evaluation approaches for stream learning classifiers. *Prequential* evaluation tests each incoming sample before using it for training. *Holdout* evaluation periodically tests the model on a fixed subset of data not used for training. *Delayed* evaluation tests each incoming sample and uses them for updates after a specified delay.

naturally adapt to changing data concepts. In this case, for detecting abrupt concept shifts, a concept drift detector may be necessary for faster adaptation. Unlike traditional batch-oriented schemes, SL models are generally better suited to adapt to these inherent changes in data flows [25].

Balancing the retention of prior knowledge while adapting to new concepts during incremental SL model updates is a challenging task [26]. This challenge arises because the decision boundaries of SL classifiers are continuously updated to accommodate new events. A widely adopted approach in the design of SL-oriented classifiers is incorporating a grace period parameter to tackle this issue. This parameter ensures that significant adjustments to the classifier's decision boundaries occur only after processing sufficient events for incremental training. For instance, the Hoeffding Tree classifier [27] splits a leaf node only after it has observed a predefined number of events, as specified by the grace period parameter. As a result, although incremental model updates are conducted, the decision boundaries of SL classifiers are usually only substantially affected according to the previously defined grace period parameter.

Evaluating a SL classifier is inherently challenging due to data streams' continuous and evolving nature [28]. Researchers typically adopt one of three main approaches for evaluation, namely *prequential*, *holdout*, or *delayed*, as shown in Figure 1. Prequential evaluation involves testing each incoming sample before using it for training, enabling real-time performance assessment but requiring careful handling to avoid bias in evolving data distributions. On the other hand, holdout evaluation relies on periodically testing the model against a fixed subset of data not used for training, providing a stable reference for comparison but potentially overlooking concept drift in the stream. Lastly, delayed evaluation assesses the model's performance on a batch of data after it has been trained on earlier samples, introducing a controlled delay to account for temporal dependencies while balancing computational and memory constraints. Each approach offers distinct advantages and trade-offs, making the evaluation method important for ensuring accurate and meaningful performance assessments in SL.

Managing the life-cycle of a SL model is more complex than that of a traditional ML model, as online systems need to handle continuous data ingestion while at the same time ensuring near real-time model update and inference, usually in a distributed setting [11]. As a result, despite the increased efficiency in various applications, implementing SL in production environments remains a significant challenge for ML professionals [29]. These challenges make SL more suitable for academic environments, where the scenarios can be more easily controlled, and the delay in inference is less critical than that observed in commercial environments. This situation often leads companies to rely on near-real-time learning in streaming scenarios to partially mitigate the obstacles associated with SL [30].

B. Machine Learning Operations (MLOps)

MLOps is a practice designed to optimize the development, deployment, monitoring, and ongoing management of ML-based systems in production environments. It extends the principles of Development and Operations (DevOps) to address the unique challenges of the ML model life-cycle, ensuring that they are successfully deployed and maintained in production while efficiently updating them as new data arrives [5]. This approach combines provision, monitoring, and update processes to maintain model performance and proactively address concept drift by integrating practices and tools tailored to the ML life-cycle.

The MLOps operation is typically carried out through the following phases:

- *Provision*. It involves deploying the trained ML model into production, ensuring it has the resources and infrastructure to be queried by multiple users at scale. This is typically achieved by deploying multiple endpoints as web services in a distributed environment that executes the inference phase in parallel to handle user requests;
- Monitoring. It continuously tracks the model's performance and detects concept drift or accuracy degradation issues. It is used to ensure that the deployed model maintains the expected accuracy despite possible changes in the behavior of the deployed environment;
- *Update*. It modifies or retrains the deployed model based on new environment data or changing requirements to maintain system accuracy. On a traditional ML setting process, it is usually triggered periodically based on the conditions identified by the *monitoring* module;
- *Versioning*. It manages the model's different iterations, including keeping track of parameter changes and facilitating possible rollbacks. It aims to allow for possible iterations across the different stages of deployment;

Meeting the requirements of the operational phase at scale is a challenging task that often requires multiple frameworks that can operate in a distributed environment. These include tools such as AirFlow [31], Kubeflow [32], and Seldon [33], which

provide orchestration and automation across the ML life-cycle, while others such as MLflow [7] to focus on specific phases such as experiment tracking and model versioning.

Given the diverse requirements of MLOps, service providers are also implementing platforms such as AWS SageMaker, Azure Machine Learning [34], and Google Cloud AI Platform, which offer integrated solutions to manage the entire MLOps life-cycle. Implementing MLOps is necessary to ensure the proper scaling and maintenance of ML systems in production, reduce errors, save time, and improve governance by ensuring that these systems are effectively meeting the organization's needs.

C. When Stream Learning Meets MLOps

Given the increasing use of SL techniques in recent years, several tools have been proposed for deploying designed models in production. For example, Massive Online Analysis (MOA) [35] provides a framework for the implementation of algorithms and the execution of experiments on evolving data streams. Similarly, SAMOA [36] aims to extend MOA's capabilities for mining large data streams through a distributed architecture. It supports classification, clustering, and regression [2], and is mainly used in academic environments, although it can be interfaced with tools like Apache Storm [37] and Apache Samza [38].

In commercial environments, frameworks such as Scikit-multiflow [39], Creme [40], and River [41] are gaining popularity. Scikit-multiflow offers a streaming version of Scikit-learn [42], while Creme is known for its simplicity, and River combines both strengths by offering a wide range of SL algorithms. Unfortunately, while they allow for incremental updates, they typically do not cover the entire SL lifecycle, especially when scalability is required, which demands the utilization of additional tools for full production deployment.

Implementing MLOps for SL algorithms presents many challenges that current tools struggle to address. The SL *provision* requires near real-time model access, which makes it difficult to dynamically allocate resources while maintaining low latency, especially when provisioning must occur for each new event. Frequent deserialization of updated models is particularly challenging in near real-time, as the classification task can only be performed after the updated model version has been adequately parsed. Additionally, relying on concept drift detection to trigger model updates introduces further complexity, as the drift detection algorithm must be carefully tuned to the specific dataset and deployment environment, adding a layer of dependency and making the process less adaptable to varying conditions.

The need for continuous model updates triggered for each new event to adapt to evolving data further complicates the deployment task. Each update requires model serialization, versioning, and deserialization within the deployment modules. In addition, managing multiple versions of a model in a streaming context is complex due to the constant flow of data, and ensuring seamless versioning and reproducibility can lead to an unreliable infrastructure as each new event can potentially change the model. In this context, the near real-time demands of SL algorithms are typically not addressed by current MLOps architectures.

III. Related Works

Developing new architectures for MLOps operations has been a widely studied topic in the literature over the past years. However, SL-oriented MLOps architectures are still in their infancy. For example, StreamDM [19] enables distributed data processing for SL tasks, but its near real-time effectiveness is limited because it relies on mini-batches, which introduces latency. SOLMA [43], a component of the Apache Flink ecosystem [44], provides scalability, fault tolerance, and parallel processing support but lacks model versioning capabilities despite enabling near real-time inference. A microservices-based architecture for real-time analytics, where models are trained in batches and updated in mini-batches, is proposed by Donna Xu *et al.* [45]. This architecture is implemented on Apache Spark [46] and the MLlib library [47] to support parallel processing and ensure scale performance. However, SL-oriented model updates are neglected. Similarly, Igor L. Markov's *et al.* [22] streamlines the ML life-cycle but limits model updates to batch processing, a feature that may not be suitable for all SL applications.

Typically, proposed MLOps architectures are targeted at specific use cases, limiting their broad adoption. As an example, a packaging and deployment mechanism for analytic pipelines has been proposed by Raúl Miñón *et al.* [48] for streaming pipelines. It supports distributed deployment through dockers and is decoupled from the training phase to avoid integrating experiment code into operational deployments [49]. Unfortunately, although it supports streaming model inference, it fails to address near real-time model updates. Similarly, Rohan Ramanath *et al.* [50] predict ad click rates by combining batch processing with mini-batch learning while adapting to data variations over time using a lambda architecture. Their approach performs online training with mini-batches and requires code adaptations for other use cases. Fabrizio Carcillo *et al.* [51] integrated several Big Data tools with ML to provide near real-time credit card fraud detection using random forests and sliding windows for model updates. Its structure cannot be easily customized, and its applicability is limited to specific scenarios. Similarly, Janez Kranjc *et al.* [52] presented a web application that supports data mining workflows through a graphical user interface, enabling workflows to be executed in streaming environments using daemons. The near real-time capabilities of their system are limited because training is performed in batch mode.

It is a common practice in the literature to rely on well-known frameworks for stream processing. The Big Data Engine tool developed by Mikołaj Komisarek *et al.* [20] integrates Apache Spark, Apache Kafka [58], Elasticsearch [59], Kibana [60], and HDFS [61] for network anomaly detection. It supports near real-time processing and vertical scaling. However, it does not detail model update mechanisms; instead, it focuses on training with data flow. Similarly, Spring XD, presented by Sabby Anandan *et al.* [30], is module-based and interacts with technologies such as Apache Spark,

TABLE I: A summary of related work and the characteristics of their MLOps implementations.

Work	Streaming Architecture	Streaming Update	Streaming Processing	General Purpose	Distributed Architecture	Model Versioning
A. Bifet *et al.* [19]	✓	×	×	✓	✓	×
W. Jamil *et al.* [43]	✓	✓	✓	✓	×	×
D. Xu *et al.* [45]	×	×	×	✓	✓	×
I. Markov *et al.* [22]	×	×	✓	×	×	×
R. Miñón *et al.* [48]	×	×	✓	✓	×	×
R. Ramanath *et al.* [50]	×	×	✓	×	×	×
F. Carcillo *et al.* [51]	×	×	×	×	✓	×
J. Kranjc *et al.* [52]	×	×	✓	✓	✓	✓
M. Komisarek *et al.* [20]	✓	×	×	×	✓	×
S. Anandan *et al.* [30]	×	×	×	✓	×	×
S. Rodriguez *et al.* [53]	×	×	✓	✓	×	✓
C. Martín *et al.* [54]	×	×	✓	✓	✓	✓
D. Crankshaw *et al.* [55]	×	×	×	✓	✓	×
D. Baylor *et al.* [56]	×	×	✓	✓	✓	✓
A. Pareek *et al.* [57]	×	×	✓	✓	×	×
Ours	✓	✓	✓	✓	✓	✓

Apache Kafka, RabbitMQ [62], Hadoop [63], Redis [64], and Apache Zookeeper [65]. It supports near real-time inference via Spark Stream and includes monitoring tools. However, it performs model updates using Spark's mini-batch mode. Another module-based system, STREAMER, introduced by Sandra Garcia Rodriguez *et al.* [53], integrates with Apache Kafka, InfluxDB [66], Redis, and Kibana. It provides scalability and fault tolerance while allowing users to focus on algorithms and data preprocessing. Unfortunately, it does not support incremental updates and relies on mini-batches or the processing of entire datasets for model updates. Kafka-ML, presented by Cristian Martín *et al.* [54], manages ML pipelines with TensorFlow [67] and PyTorch [68]. It provides a web interface for training and inference with Docker containerization [10] for portability while processing data in batches rather than incrementally. Here, trained models cannot be updated but saved or deployed for future use.

Daniel Crankshaw *et al.* [55] presented the Clipper tool, a low-latency prediction system for near real-time online deployments. It supports multiple ML models and deployment strategies through a modular architecture and unified interface. It also provides a dynamic load-balancing ingestion strategy for the even distribution of prediction tasks. However, it does not support incremental updates via a mini-batch approach. Denis Baylor *et al.* [56] introduced TensorFlow Extended (TFX), an open-source platform for managing ML pipelines. It integrates analyzing, transforming, validating, and deploying components into a unified system. TFX supports continuous model updates and transfer learning. This reduces training time by transferring parameters from a base to a target network. The feasibility of incremental updates is still being discussed, although continuous updates are supported. Alok Pareek *et al.* [57] presents Striim, an enterprise-grade platform for near real-time data ingestion and ML that uses models such as random forests and Gaussian process regression and makes use of sliding windows for model updates. StreamMLOps, proposed by Mariam Barry *et al.* [17], provides an architecture for online learning. It uses open-source tools such as River and MLFlow. It supports continuous learning and deployment but requires customization for specific applications.

A. Discussion

Table I reviews the literature on MLOps support for SL-oriented operations. It can be observed that most of the studies focus on updating the model offline. In this approach, labeled data is stored and periodically used to generate models more consistent with the latest information. This approach is usually based on batch processing, where models are retrained regularly to incorporate new data. At the same time, near real-time updates occur at discrete intervals rather than incrementally and often rely on sliding windows to create mini-batches for model building. Despite the challenges associated with SL support, traditional MLOps practices typically involve periodic updates through batch or mini-batch strategies.

Nevertheless, when scaling their designed systems, it is common practice in the literature to deploy additional peers using virtual machines [19], [56]. However, this approach introduces significant computational overhead due to the need to deploy an entire operating system and the required libraries. In contrast, the use of containers for infrastructure scaling in MLOps is still in its early stages in the literature [53], [54]. Containers have the potential to significantly reduce service provisioning overhead by sharing the host OS libraries and binaries while utilizing namespaces to ensure isolation [69].

Several challenges are associated with deploying SL models in a production environment. These include maintaining system responsiveness at scale while ensuring an efficient inference model under high prediction demand. Traditionally, researchers scale their training procedures in a classifier-dependent manner, which poses a challenge in adapting the architecture to different classifiers. Designing an architecture that enables the training task to be scaled without modifying the supporting architecture remains a significant challenge in the literature. Furthermore, using traditional MLOps architectures is not feasible due to the need for incremental updates as soon as event labels become available. Current MLOps architectures cannot easily manage the multiple versions of constantly evolving SL models while making them available for inference.

IV. A MLOps Architecture Towards Stream Learning Deployment

In light of this, we propose a new architecture that paves the way for implementing SL-based applications under the MLOps operating principle at scale. Our proposed architecture addresses three key challenges associated with SL-oriented MLOps environments:

- *High Inference Throughput*. To accommodate the evolving demand, the architecture introduces a horizontal scaling system that allows the number of inference endpoints to scale as required. In addition, a decoupled model

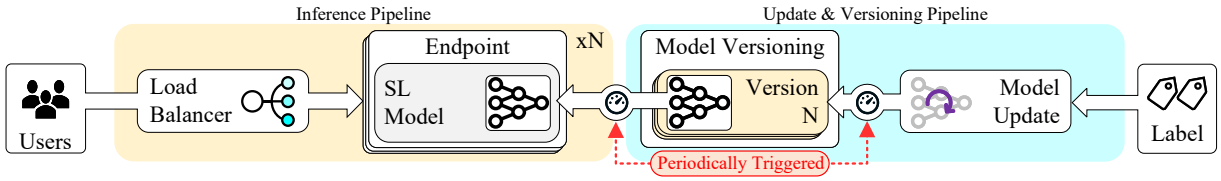
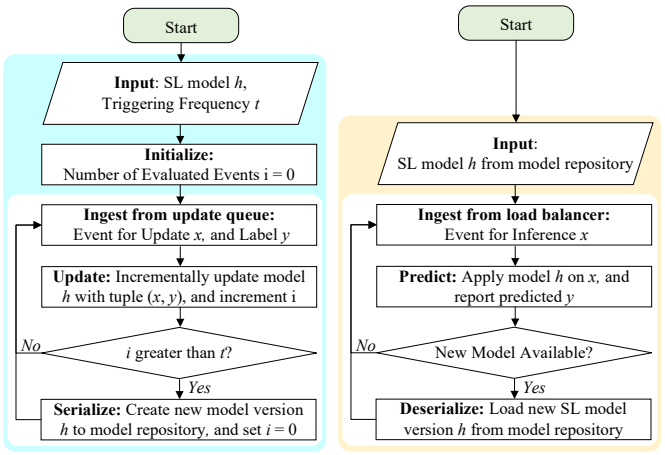


Fig. 2: An overview of the proposed pipeline for implementing the MLOps operational phase in SL-oriented applications. The *Inference Pipeline* consists of multiple endpoints designed to efficiently handle user requests at scale in a continuously dynamic data stream. The *Model Update and Versioning Pipeline* employs a periodic triggering mechanism based on an event counter to generate and deploy new SL model versions.



(a) Incremental update pipeline (b) Inference pipeline

Fig. 3: Pipeline for both inference and incremental updates. The *Inference* pipeline continuously ingests events for inference and dynamically deserializes new SL model versions as they become available. The *Incremental Update* pipeline incrementally updates the model and serializes it based on a predefined triggering frequency.

loading method is implemented, enabling the inference module to retrieve new models directly from the model repository without requiring endpoint restarts, enhancing system availability in near real-time applications;

- *Trigger-based model updates*. In contrast to traditional ML frameworks that rely on batch or mini-batch updates, our proposed architecture integrates a triggering system for incremental SL model updates via a continuous data flow. To achieve this goal, we introduce a model serialization window that is triggered periodically, allowing updated models to be serialized in the model repository and made available for inference within optimal time or event windows;
- *Decoupled Model Versioning*. The frequent model update and versioning can result in hundreds or even thousands of model versions within a short time frame. To address such a challenge, the architecture decouples the model versioning task to handle multiple model versions while also enabling a faster inference module's loading of specific model versions;

Figure 2 illustrates our proposal pipeline implementation. It includes the *Inference Pipeline* and the *Model Update and Versioning Pipeline*. The former handles user prediction requests at scale, while the latter manages the incremental update of the SL model using labeled events and appropriate model versioning.

The Inference Pipeline implementation aims to handle multiple user inference requests simultaneously at scale. It implements multiple endpoints executing the latest available SL model version to achieve this goal. Each endpoint is queried through a load balancer. This load balancer efficiently distributes the user load across all available endpoints. As a result, the inference can be scaled by loading multiple model versions on each endpoint, expanding compute capacity as needed.

The Model Update and Versioning Pipeline receives event labels for incremental model updates following a specific process. These labels are used for the incremental SL update while the pipeline initiates model versioning periodically. We designed our architecture to be classifier-independent, ensuring flexibility across different use cases. Therefore, the implementation allows the training task to be implemented as needed, allowing the operator to adjust it based on specific requirements. This is made possible by treating the training task as a standalone endpoint, independent of the classifier or underlying architecture. For instance, the operator could scale the training process without demanding significant modifications in our designed architecture. The newly generated model version is handled by a decoupled module responsible for the versioning and then made available to the deployed *Inference* modules for seamless integration and deployment. This periodic triggering of the model versioning allows the update of the SL model to be carried out under the MLOps framework. Our proposal has two main insights. First, we decouple inference, versioning, and updates, allowing incremental SL model updates without significantly impacting the performance of other modules. Second, we periodically perform model versioning and deployment, resulting in fewer model versions and increased inference throughput.

The modules that implement our proposal are described in the following subsections.

A. Model Update and Versioning

SL-oriented MLOps face significant challenges in handling frequent model updates and versioning, as these applications must operate in near real-time to accommodate continuously incoming data. Unlike traditional batch-oriented applications, which can store training data and perform model versioning at long intervals, SL models require incremental updates,

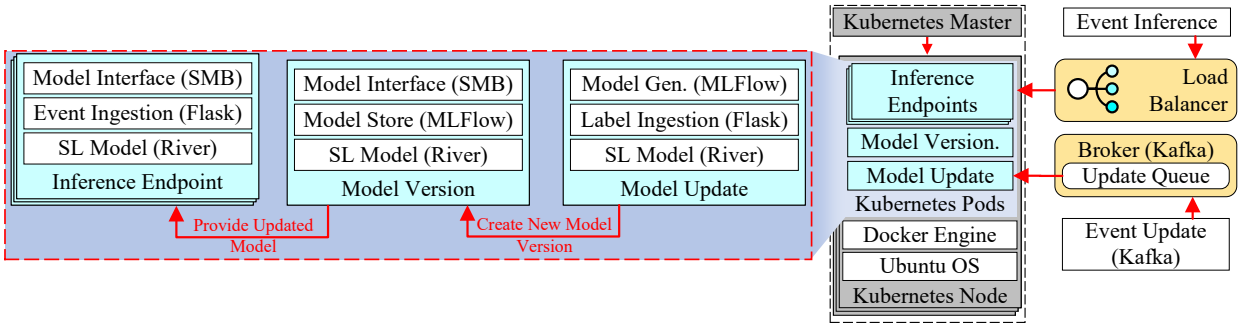


Fig. 4: Prototype overview of our proposed pipeline for implementing the MLOps operation phase on SL-oriented applications.

generating a new model version with each evaluated event. This necessitates efficient serialization for versioning and deserialization to ensure updated models are readily available at the inference endpoints. Addressing these challenges is a challenging task to enable scalable and efficient SL-oriented applications within an MLOps framework.

In response, our proposed architecture introduces a triggering mechanism that manages the execution of model versioning and provisioning tasks (Fig. 2, *Periodically Triggered*). The trigger takes advantage of the characteristic of SL algorithms by examining their decision boundaries before triggering the model versioning task. For example, the Hoeffding Tree algorithm [27] will not split a tree leaf until certain events (the grace period parameter) have been observed. Moreover, even when model modifications are made on an event basis (e.g., Naive Bayes [70]), substantial model changes that affect inference are typically observed only after evaluating a significant number of samples.

Figure 3a overviews our SL update mechanism. The process begins with an input SL model h and a predefined triggering frequency t. Initially, the number of evaluated events i is set to zero. The system then enters a continuous loop, ingesting events from the update queue. Each event x and its associated label y are used for incremental model updates. After each update, the system checks whether the number of processed events exceeds the predefined triggering frequency. A new model version is generated and serialized in the model repository if this condition is met.

Our scheme performs model versioning and deployment when a new model version is created. The versioning requires model serialization and storage, while the deployment sends the new model version to all deployed model inference endpoints in near real-time. This decoupling enables our scheme to provide high inference throughput even when using a low model trigger frequency, which may lead to the frequent generation of new model versions (Pipeline 3a, t). Recalling that the model triggering frequency should be defined based on the operator's discretion, according to the used SL algorithm and required performance (latter evaluated in section VI).

B. Model Inference

MLOps-oriented inference must handle multiple users' requests at scale. However, in traditional SL scenarios, frequent model versioning and deployment can significantly impact inference throughput due to the computational costs of loading new model versions.

We implement the SL inference task to address this challenge as a distributed and decoupled service. In practice, inference is performed by multiple endpoints, each of which performs inference using its loaded SL model. Based on the load of user requests, this approach allows for horizontal scaling of inference tasks. A load balancer manages user requests, efficiently distributing them to the appropriate deployed inference endpoint to process the task (Figure 2, *Inference Pipeline*).

When a new model version is available (see section IV-A), each endpoint halts inference, requests the latest model version from the model versioning module, and resumes inference-related tasks. Given that our scheme relies on a triggering frequency mechanism for model versioning, the computational impact on inference is not significantly degraded. This occurs because fewer SL model versions are generated over time, reducing the model deserialization efforts on the inference side.

Figure 3b provides an overview of our SL update mechanism. The process begins with an input SL model h. The system then enters a continuous loop, where events are ingested for inference from the load balancer. Each ingested event x is processed by the current SL model h, which generates a predicted label y that is subsequently reported. After each inference, the system checks for the availability of a new model version in the model repository. If a new version is detected, it is retrieved and loaded through a deserialization procedure, and the inference task proceeds. This process runs continuously for each deployed inference endpoint.

C. Discussion

The implementation of the MLOps operation phase for SL-oriented applications is a challenging task. Our proposed model is built around two key principles. First, it decouples the inference, versioning, and update processes, enabling incremental SL model updates without compromising the performance of other system components. Second, it implements periodic model versioning and provisioning, which minimizes the number of model versions generated and enhances inference throughput. In practice, we leverage the behavior of SL algorithms, where typically their decision boundaries undergo significant changes only after evaluating a certain number of events. By aligning the versioning process with these natural

adjustments, we reduce the frequency of updates, leading to fewer model versions without affecting the system's accuracy.

V. Prototype

A prototype was developed to validate the proposed architecture and assess the system's applicability in a production environment. As shown in Figure 4, a controlled environment was created to replicate a real-world application. The prototype was implemented as a distributed Python application running as microservices encapsulated in containers, and orchestrated using Kubernetes [10]. The Kubernetes cluster also manages a container running Apache Kafka [58] $v.3.7.0$. This container receives messages with labeled events to be consumed by the update and versioning endpoint. The Microk8s tool [71] $v.1.24$ was used to create and manage the Kubernetes cluster. The prototype comprises three main modules: *Inference Endpoints*, *Model Versioning*, and *Model Update*.

The *Inference Endpoints* are designed to be scalable on demand for executing the model inference task. We use a Kubernetes load balancer that appropriately distributes the load across each endpoint deployment. The endpoint is executed as a Docker container. It loads the selected SL model in River API $v.0.21.2$ [41] format and ingests events to be evaluated using Flask API $v.3.0$ [72]. It periodically checks for updated model versions. In this case, it downloads the updated model over the SMB protocol as it is provided and stored by the *Model Versioning* module.

The *Model Versioning* module stores and serves several SL model versions. It uses the MLFlow API $v.2.16.0$ [7] for river SL model storage. It receives updated model versions generated by the *Model Update* module, stores them, and makes the updated model versions available to the *Inference Endpoint* via the SMB protocol.

Finally, the *Model Update* module is responsible for incremental SL model updates. We deploy a broker using the Kafka API $v.3.7.0$ [58], and continuously publish the events with their previously used labels for inference. Recall that we have two ingestion pipelines, one for the inference task, implemented as a web service, and the other for the update task, implemented as a publish-subscribe model. The module continuously receives updated events by listening to the corresponding Kafka topic. It uses these events to incrementally update the SL model. In addition, it periodically generates a new version of the model (Pipeline 3a, t) for the *Model Version* module.

The proposed architecture addresses the SL challenges by enabling horizontal scaling of inference endpoints, seamless model loading without application restarts, incremental model updates, near-optimal model serialization, and efficient model version management. The system is built as containerized microservices orchestrated by Kubernetes. It is highly flexible and portable, suitable for both on-premises and cloud deployments. The inference engine efficiently loads model updates and scales horizontally, while the model update and versioning modules support incremental updates at a lower computational cost. They manage multiple model versions to ensure rapid model replacement across active endpoints. As a result, the architecture improves the system's ability to adapt to evolving data while maintaining stability and performance.

VI. Evaluation

Our conducted experiments aim to answer the following Research Questions (RQs):

- ***RQ1:*** *What is the baseline accuracy of the SL model in an incremental update scenario?*
- ***RQ2:*** *What is the throughput of our proposed scheme for inference without incremental updates?*
- ***RQ3:*** *What is the impact of the frequency of model versioning on the accuracy and throughput?*
- ***RQ4:*** *How does endpoint parallelism affect throughput and accuracy?*

The next subsections describe the performance of the SL model building aspects.

A. The Datasets and Stream Learning Classifiers

We evaluated our proposal considering both synthetic and realistic datasets, as follows:

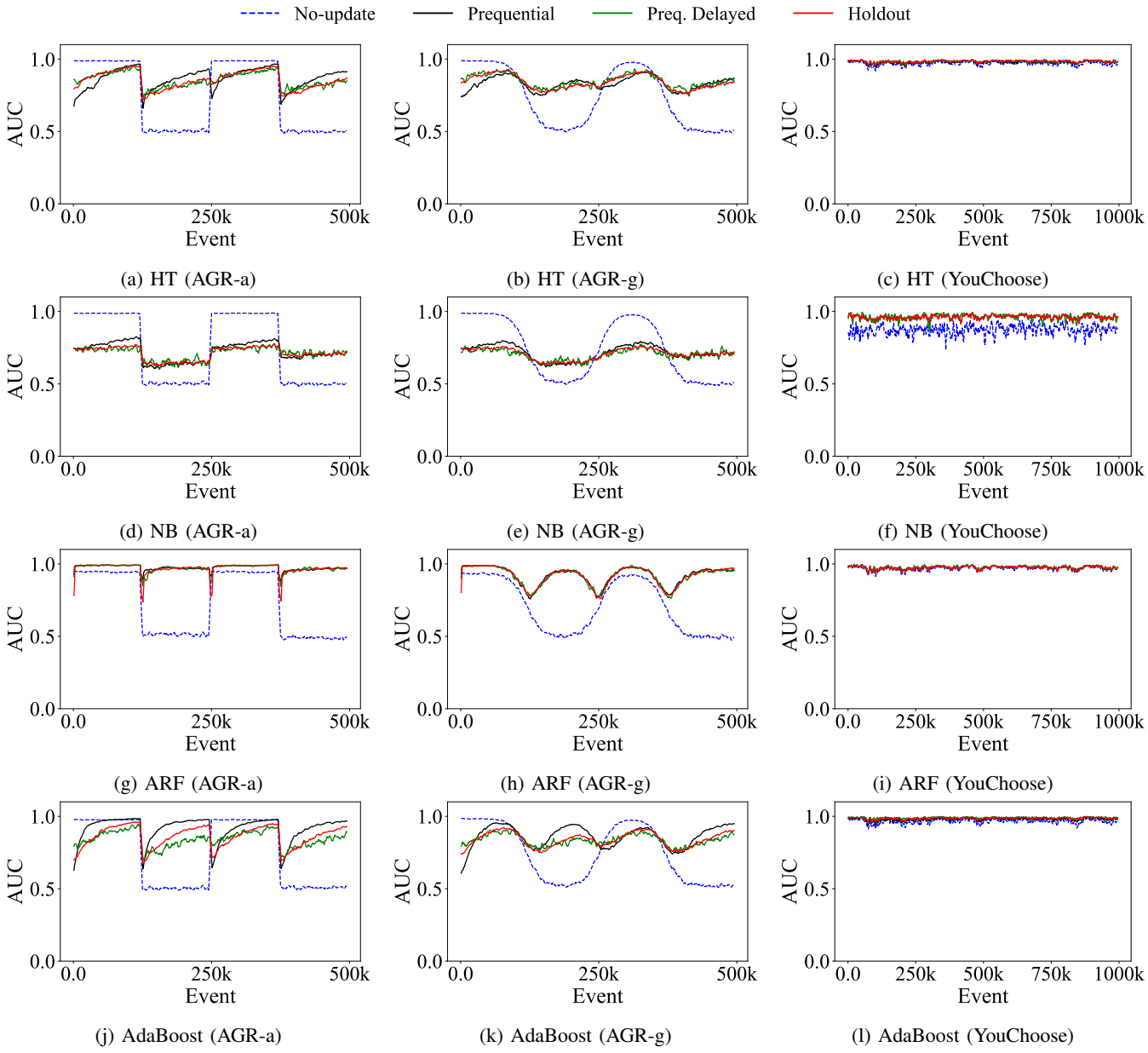
- **AGR-a** and **AGR-g** [73]. A synthesized dataset based on the Agrawal generator. It comprises six nominal and three numerical features for a binary classification task. The dataset includes three abrupt concept drifts for the AGR-a dataset, and three gradual concept drifts for the AGR-g dataset;
- **YouChoose** [74]. A real-world dataset that captures user clicks and purchase events over several months was collected from an online retailer in 2014. It consists of 16 characteristics and a target attribute indicating whether a purchase was made;

We evaluated our scheme executed with 4 widely used SL classifiers, namely Naive Bayes (NB) [70], Adaptive Random Forest (ARF) [11], Hoeffding Tree (HT) [27], and Adaptive Boosting (AdaBoost) [75]. The Gaussian distribution technique was used for the NB. The HT was evaluated with a 200 grace period, information gain as a split criterion, and 100 maximum tree size. The ARF was evaluated with 10 base learners with the ADWIN drift detector, and information gain as the split criterion. The AdaBoost was assessed with 5 HT as the base learner with *gini* as the split criterion. The River API $v.0.21.2$ (see Fig. 4) was used to implement the selected classifiers. The parameters were set empirically, as often made in related works, and no significant differences were observed when varied.

B. The Stream Challenge

Our first experiment aims at answering *RQ1*. It investigates the accuracy performance of the selected SL classifiers with an incremental update implementation. In practice, we establish a baseline detection accuracy before investigating the implementation aspects of our proposed MLOps architecture, as follows (see Fig. 1):

- *No-update*. The classifier is initially trained using the first 1% of the dataset. The resulting model then evaluates incoming samples without performing incremental updates;



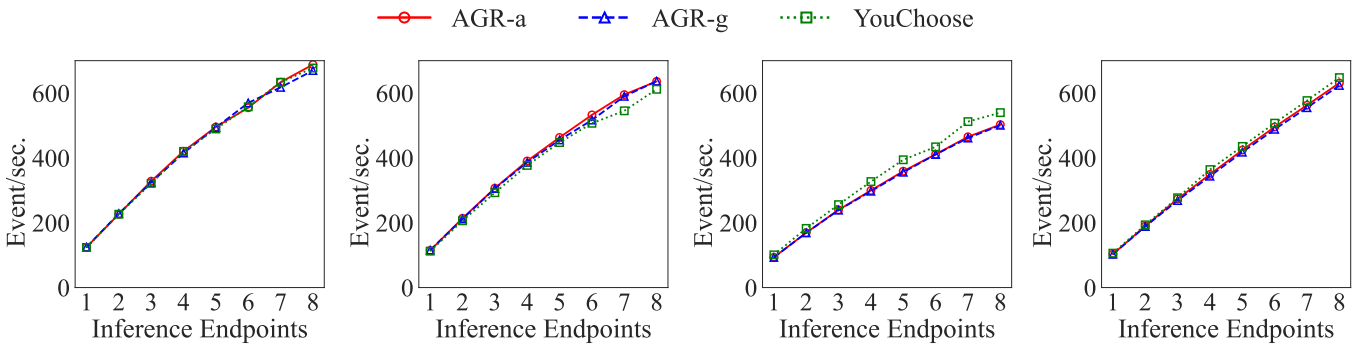
(a) HT (AGR-a) (b) HT (AGR-g) (c) HT (YouChoose)

(d) NB (AGR-a) (e) NB (AGR-g) (f) NB (YouChoose)

(g) ARF (AGR-a) (h) ARF (AGR-g) (i) ARF (YouChoose)

(j) AdaBoost (AGR-a) (k) AdaBoost (AGR-g) (l) AdaBoost (YouChoose)

Fig. 5: Accuracy of selected SL classifiers on common datasets with and without incremental model updates, measured for every thousand-sample interval.



(a) HT (b) NB (c) ARF (d) AdaBoost

Fig. 6: Scalability of the *Inference Endpoint* of our proposed scheme without the application of incremental model updates, measured by the average number of events classified per second.

- *Prequential*. Evaluate each incoming sample, apply it for incremental model updates, and proceed to process the next sample;
- *Holdout*. Process input in batches of 1,000 samples. Each batch is partitioned such that 70% of the samples are allocated for testing, while the remaining 30% are utilized for incremental model updates;
- *Delayed*. Evaluate each incoming sample and apply it for incremental model updates with a delay of $1,000$ samples;

Therefore, we assess the accuracy of the selected classifiers using multiple evaluation setups, both with and without incremental model updates. According to the following equation, we assess the accuracy of the selected classifiers by their Area Under the Curve (AUC) values.

$$\text{AUC} = \int_0^1 \text{TPR}(t)\, d\text{FPR}(t) \tag{1}$$

where the True-Positive Rate (TPR) denotes the ratio of positive events correctly classified, the False-Positive Rate (FPR) denotes the ratio of non-positive events incorrectly classified as positive, and t is the classification threshold set of the classifier. As commonly used in the literature, the AUC values are computed every thousand events. We used AUC because it provides a comprehensive measure of the model's ability to distinguish between positive and negative classes across all possible thresholds simultaneously. Since the classification operation point should be determined at the operator's discretion, AUC allows for an unbiased evaluation by considering both the TPR and FPR, regardless of class distribution.

Figure 5 shows the obtained AUC values for the selected SL classifiers with and without incremental model updates being conducted. It is possible to note that the selected datasets require the execution of model updates for all evaluated SL classifiers, regardless of the utilized model update setup. For example, the HT on AGR-a (Fig. 5a) significantly degrades the measured AUC by an average of 0.13 throughout the evaluation period when no model updates are performed. Conversely, performing incremental model updates using the *prequential*, *holdout*, or *delayed* strategies can accommodate the evolving behavior of the selected datasets. Surprisingly, the selected classifiers achieve similar detection accuracies, irrespective of the underlying dataset and evaluation strategies. This demonstrates that the *delayed* update strategy can be effectively used for incremental model updates without significantly affecting model accuracy. As a result, this evaluation shows the impact that changes in behavior have on the classification performance of selected techniques.

C. Towards a MLOps for Stream Learning

Our second experiment aims to address *RQ2* by investigating the inference throughput of our proposed scheme without applying incremental updates. To this end, we scale the number of deployed *Inference Endpoints* on the deployed Kubernetes pods (see Fig. 4). We set the CPU limit to 0.5 for each docker in use. The experiment aims to investigate whether our implemented prototype can scale appropriately according to the number of deployed inference endpoints and the processing load. The goal is to ensure that our proposed architecture is effectively designed for distributed environments, demonstrating its ability to scale in response to adding new peers.

Figure 6 shows the inference throughput according to the number of deployed endpoints. It can be observed that our proposed model has an improvement in inference throughput as the number of endpoints increases. For example, when the number of endpoints is increased from 1 to 8 on the AGR-a dataset, the inference throughput of the HT model increases from 124 to 688, an increase of 554% (Fig. 6a). Therefore, adding new peers results in an average increase of approximately 71% in inference throughput. This improvement is achieved with minimal variation across different datasets and classifiers. Thus, regardless of the SL application under consideration, our proposed model can effectively scale inference.

Our third experiment aims to answer *RQ3* and investigates how the frequency with which model versions are generated can affect both the accuracy of the inference and the throughput. In practice, we vary the frequency at which model versions are triggered (Pipeline 3a, t). This impacts the frequency of generated new model versions, which can negatively affect inference throughput given the frequent need to deserialize new SL classifiers. We varied the model version triggering frequency (Pipeline 3a, t), from $1,000$ to $2,000$ in intervals of 250 events. The test was executed with various active endpoints for each trigger frequency, from two to eight.

While the number of endpoints used in the test also influences the results, the primary focus of the evaluation is on the frequency of model availability. To achieve such a goal, we first examine the number of queued events on the update Kafka queue across different availability frequencies (Fig. 4, *Update Queue*). Recalling the proposed approach, unlike conventional methods found in the literature, the inference module can load the updated model into the endpoints' memory without recreating it, thus reducing the model update requirements on computational costs. The inference, update, and versioning endpoints also operate in parallel processes, allowing models to be loaded with lower computational overhead when replacing them. However, despite this computational advantage, deploying the model at very short intervals can still significantly impact the system, leading to a decline in predictive performance and system throughput

Figure 7 shows the number of events on the update queue according to our scheme's used model version triggering frequency. An increase in the number of events in the update queue can be observed as the availability frequency parameter and the number of active endpoints in the system vary. However, when the number of endpoints reaches higher values, typically above 6, the queue often stabilizes or decreases for certain frequency parameter values. Further analysis of the results shows that for lightweight models like NB (Figs. 7d, 7e, and 7f) and HT (Figs. 7a, 7b, and 7c), higher availability frequency values (around $2,000$ events) have less impact on the update queue. Sometimes, the queue remains empty even with as many active endpoints. As the availability frequency

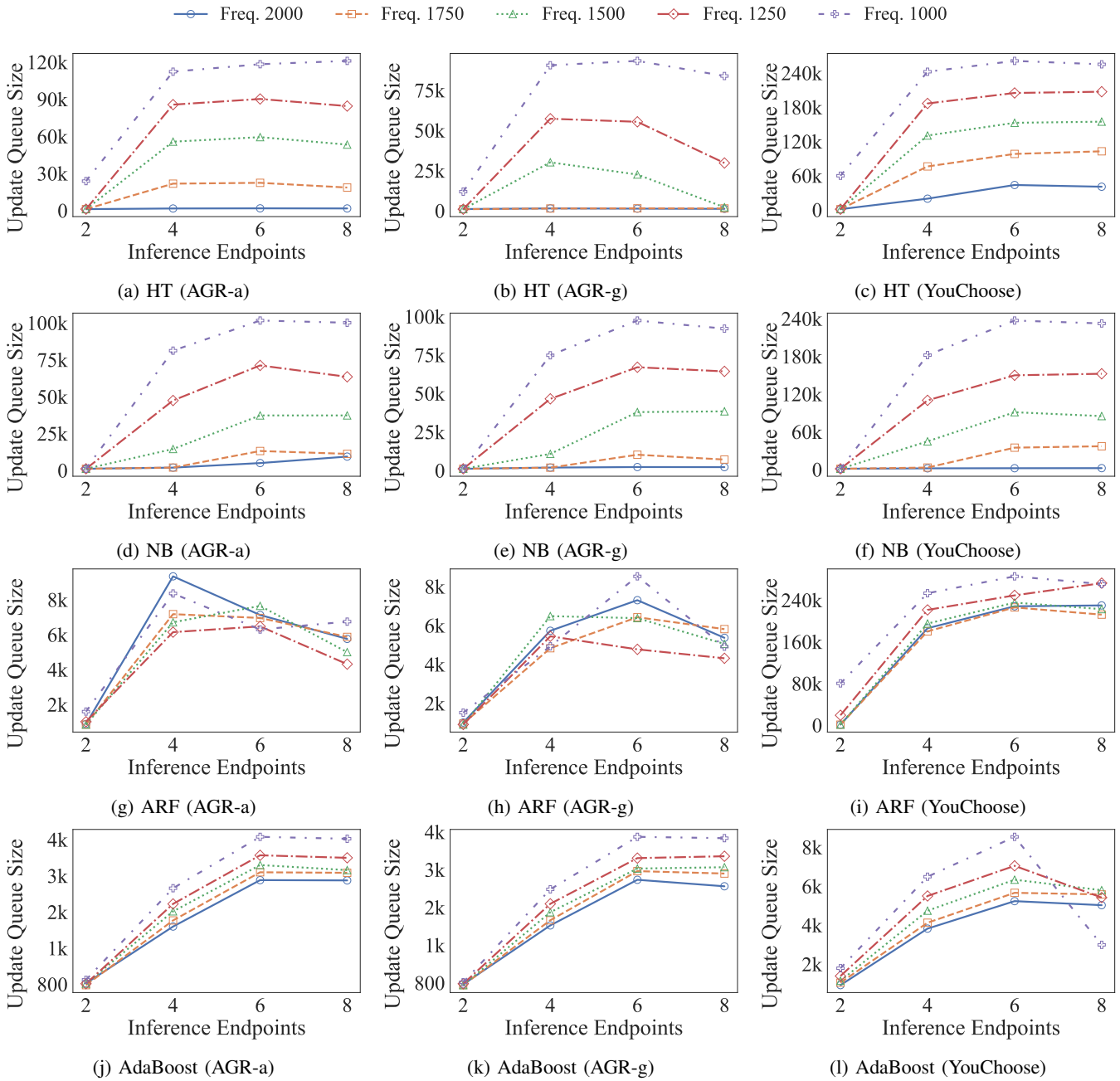


Fig. 7: Number of events in the update queue according to the model version trigger frequency (Pipeline 3a, t).

decreases ($\approx 1,000$ events), a consistent rise in the queue is observed, sometimes exceeding 250 thousand events waiting for consumption. In contrast, the ARF (Figs. 7g, 7h, and 7i) and AdaBoost (Figs. 7j, 7k, and 7l) classifiers does not exhibit such large fluctuations. This is because their inference task is computationally expensive, which slows event consumption during inference. As a result, fewer instances are sent to the update module, leading to smaller queue sizes.

Next, we examine the impact of model availability frequency on system throughput. To achieve this, we measured the number of events consumed per second and the number of available models for each case. Figure 8 shows the event inference throughput, considering the processing of the entire pipeline across all availability frequency values and the number of active endpoints. Higher model version triggering frequency ($\approx 2,000$ events) results in greater consumption of events per second compared to lower values. This occurs because their inference task is computationally expensive, slowing event consumption during inference. Consequently, a similar update queue size is observed regardless of the model versioning frequency, as the inference time is significantly higher than the time required for model serialization at the update endpoint and deserialization at the inference endpoints. As a result, fewer instances are sent to the update module, leading to smaller queue sizes.

Stabilization of the event consumption is observed when the number of active endpoints exceeds 6, similar to the results for the update queue. In practice, for the HT (Fig. 8a, 8b,

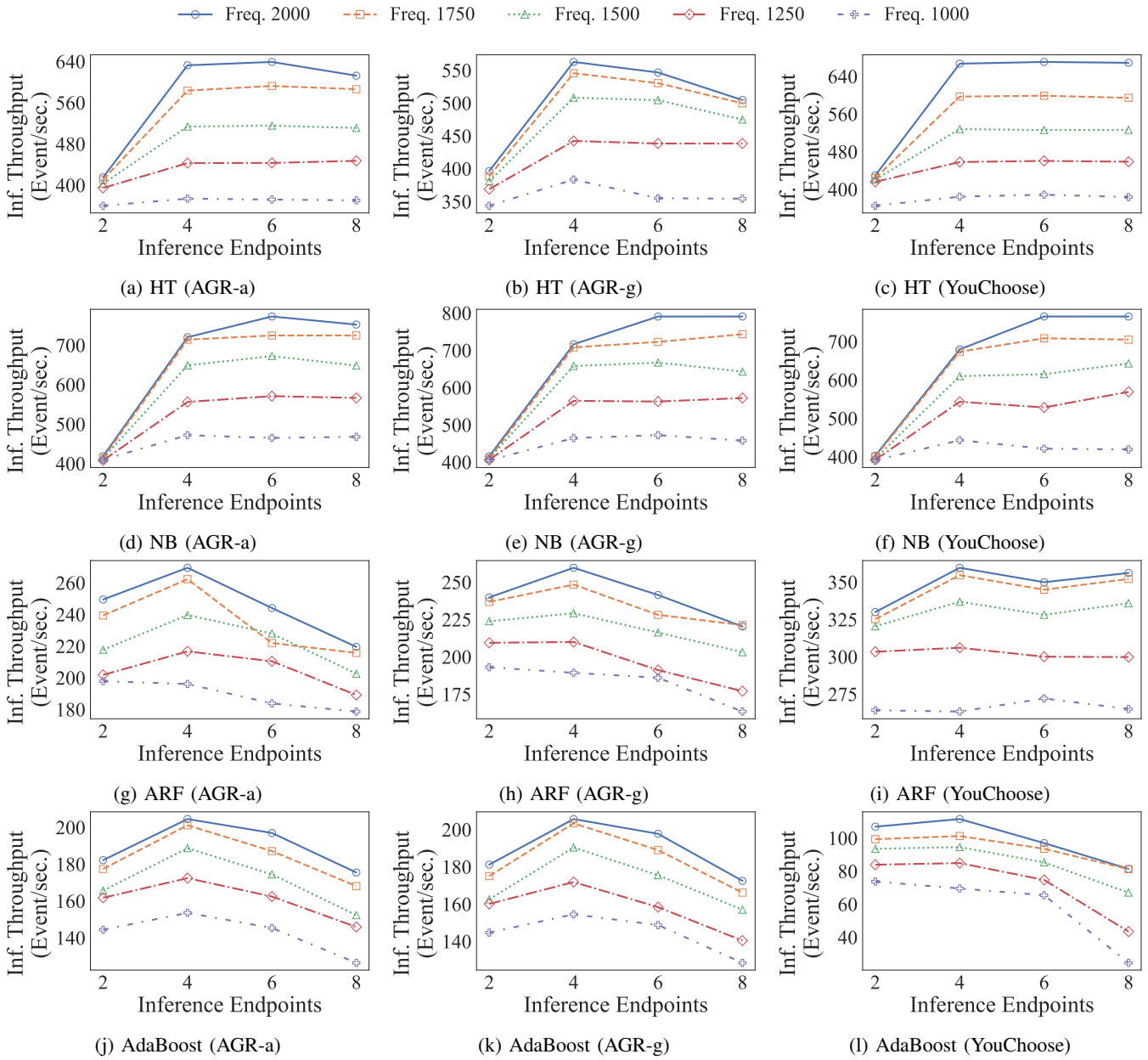


Fig. 8: Inference throughput according to the number of inference endpoints $vs.$ the frequency of triggering the model version (Pipeline 3a, t).

and 8c) and NB (Fig. 8d, 8e, and 8f) learners, the inference throughput exceeds $\approx$ 650 instances per second when the model availability frequency is set to $2,000$ events and the number of active endpoints is 4 or more. As availability frequency decreases, event consumption gradually declines, with a maximum rate of about $\approx$ 480 events per second. In contrast, computational expensive models based on the ARF (Fig. 8g, 8h, and 8i), and AdaBoost (Fig. 8j, 8k, and 8l) exhibit lower event consumption rates due to their higher computation requirements. As a result, the model availability frequency significantly impacts system throughput. The system consumed nearly twice as many instances per second with the $2,000$ events parameter set than with the $1,000$ events configuration, especially with more than 4 active endpoints.

Figure 9 shows the model version frequency according to the number of inference endpoints and the model version triggering. The system demonstrated high throughput, with model versioning occurring every $\approx$ 2.5, even when the availability frequency was set to higher values ($2,000$ events) and the number of active endpoints was kept low (2 endpoints). Due to the significant system overhead in distributed environments, variations in the availability frequency parameter often had no direct impact on the number of models deployed per second. Models based on the ARF (Fig. 9g, 9h, and 9i), and AdaBoost (Fig. 9j, 9k, and 9l) learners were more affected by frequency changes due to their larger size, while lightweight models, such as those based on NB (Fig. 9d, 9e, and 9f) and HT (Fig. 9a, 9b, and 9c), maintained high deployment rates regardless of frequency adjustments.

Consequently, our proposed model can conduct model up-

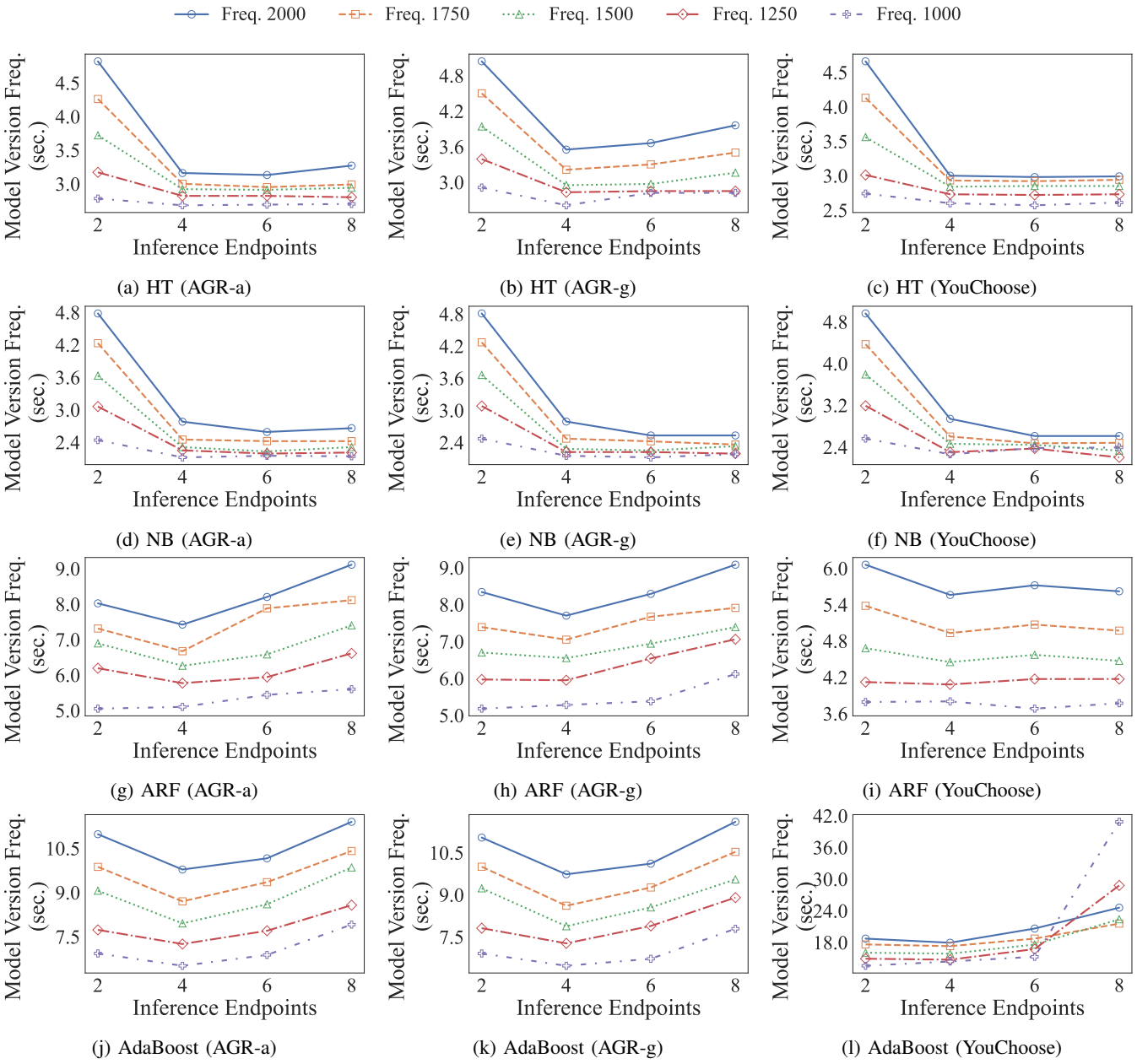
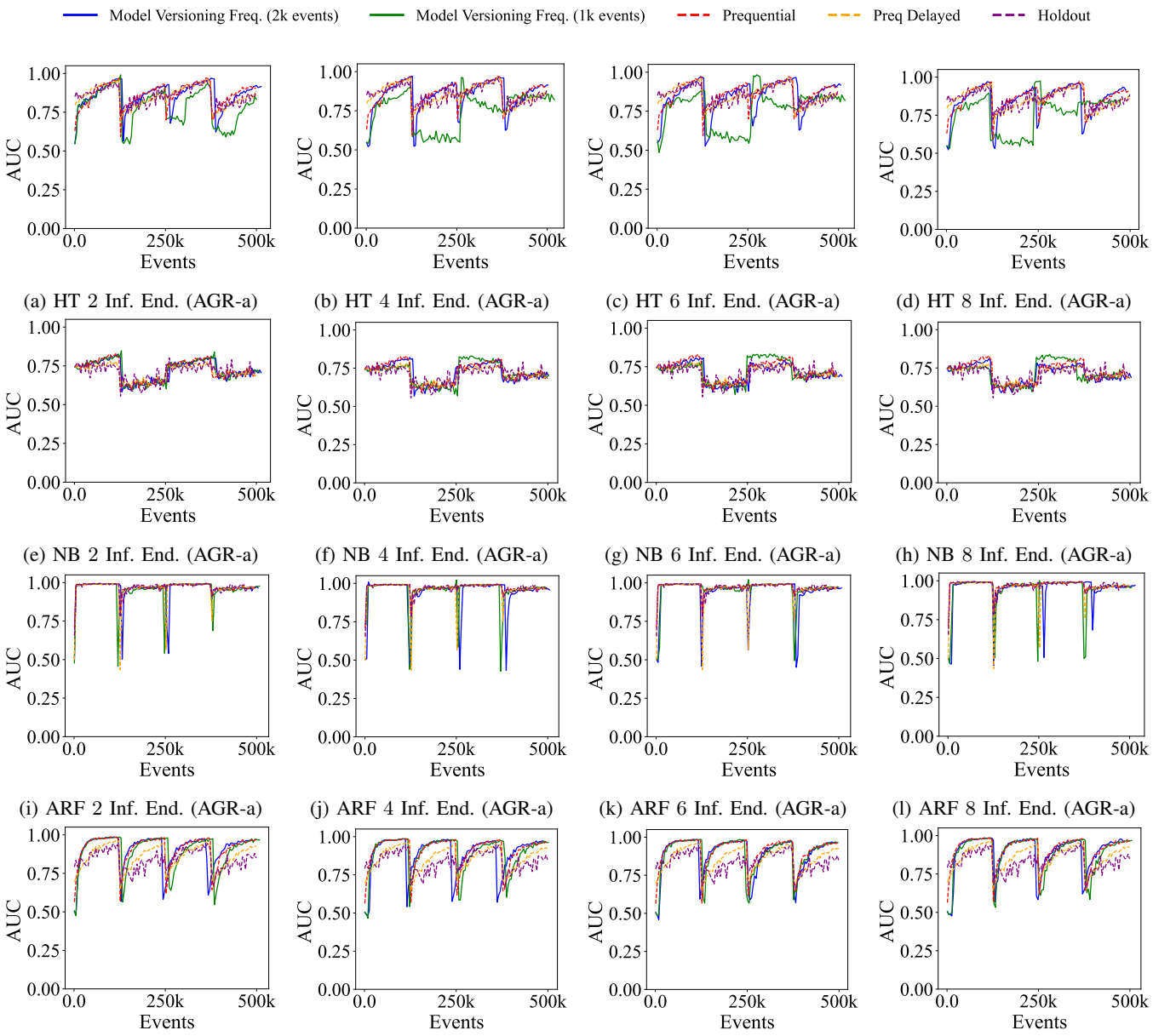


Fig. 9: New model version frequency provision, according to the number of inference endpoints $vs.$ the model version trigger frequency (Pipeline 3a, t).

dates and versioning at a significantly high frequency. In practice, the achieved model versioning frequency can be relaxed according to the considered application dataset and model. For example, the ARF learner can adjust quicker to changes in the environment than other classifiers. As a result, model generation can be conducted less often, which results in a higher inference throughput without degrading the system's accuracy.

Finally, we answer *RQ4* by investigating how each chosen configuration can affect our scheme accuracy. Using the model version triggering frequency parameter, we investigated classifier accuracy as a function of the number of inference endpoints.

Figure 10 shows the accuracy performance of the selected classifiers on the AGR-a dataset as a function of the number of inference endpoints used and the frequency of model versioning. The selected classifiers respond differently to updates due to the different characteristics of each learner. The NB (Fig. 10e, 10f, 10g, and 10h), known for their high elasticity, retain old concepts and adapt minimally to new data. Even with lower availability frequency and maximum active endpoints, the AUC metric remains stable, indicating minimal variation and close alignment with the baseline. The ARF (Fig. 10i, 10j, 10k, and 10l), and AdaBoost (Fig. 10m, 10n, 10o, and 10p) learners, on the other hand, adapts quickly to current data and forgets older concepts. However, its large size slows inference throughput, which reduces the size of the update queue and allows updating with more



(a) HT 2 Inf. End. (AGR-a) (b) HT 4 Inf. End. (AGR-a) (c) HT 6 Inf. End. (AGR-a) (d) HT 8 Inf. End. (AGR-a)

(e) NB 2 Inf. End. (AGR-a) (f) NB 4 Inf. End. (AGR-a) (g) NB 6 Inf. End. (AGR-a) (h) NB 8 Inf. End. (AGR-a)

(i) ARF 2 Inf. End. (AGR-a) (j) ARF 4 Inf. End. (AGR-a) (k) ARF 6 Inf. End. (AGR-a) (l) ARF 8 Inf. End. (AGR-a)

(m) AdaBoost 2 Inf. End. (AGR-a) (n) AdaBoost 4 Inf. End. (AGR-a) (o) AdaBoost 6 Inf. End. (AGR-a) (p) AdaBoost 8 Inf. End. (AGR-a)

Fig. 10: AUC values for each selected classifier on the AGR-a dataset as a function of the number of inference endpoints and the model version trigger threshold. Our proposed scheme achieves similar baseline accuracy while implementing distributed MLOps.

recent data, which helps manage concept drift. Although occasional update queue spikes occur as drift detection mechanisms consume additional resources, system instance consumption normalizes as data concepts stabilize, keeping performance near baseline. Finally, changes in model availability frequency significantly affect the HT models.

Figure 11 further investigates the accuracy distribution of the ARF SL classifier on the AGR-a dataset compared to the selected evaluation setups. The *Prequential* evaluation setup achieves the highest median accuracy, reaching an AUC of 0.9792. However, our proposed scheme demonstrates a similar accuracy distribution to the *Prequential* setup, even when employing a high model versioning frequency. For instance, with a model versioning frequency of one thousand events and eight inference endpoints (Fig. 10l), our model achieves a median AUC of 0.9754, representing only a 0.39% decrease compared to the best-performing evaluation setup. Additionally, the accuracy distribution range remains comparable to the *Prequential* setup, with an interquartile range of 0.0259 versus 0.0249, an increase of merely 4%. It is important to note that these results are obtained while performing near real-time model updates and versioning, along with inference at scale.

D. Discussion

The conducted experiments showed that our proposed scheme enabled the implementation of SL algorithms under

the MLOps framework. In practice, the following remarks were observed.

- ***RQ1***. Incremental model updates, in which the model is updated and then used to predict the next event, allow for rapid adaptation to changes in the data while maintaining satisfactory predictive performance throughout the data flow (Fig. 5). This incremental update behavior is often a must to ensure reliable SL performance;
- ***RQ2***. Scaling the number of inference endpoints significantly increases the inference throughput. However, this increase is not linear due to the distributed system overhead (Fig. 6). Actual performance deviates from the ideal as the number of active endpoints increases, with actual performance averaging 30% less than the ideal for 8 active endpoints. Due to their size and complexity, models based on the ARF learner perform less than their counterparts. The scalability of the inference throughput does not significantly fluctuate according to the used dataset, attesting to the general purpose of our architecture;
- ***RQ3***. Model versioning frequency has an impact on the number of events in the update queue, which significantly degrades other results (Fig. 7). Although system overhead can affect these values, higher model versioning frequencies tend to increase event consumption and the number of updated models per second. Model frequency also has different effects on the predictive performance of models, with each type of learner responding differently to changes in frequency and update queue size. In general, higher values for the model frequency trigger will lead to a better predictive performance than lower values.
- ***RQ4***. Increasing the number of active endpoints directly affects the update queue, inference throughput, and model predictive performance. As the number of endpoints increases, there is a tendency for the update queue to grow significantly. This expansion leads to decreased system throughput and a deterioration in predictive performance. The growing queue can result in outdated models and slower recovery as the time between instance arrival and consumption increases.

The analysis of the results indicates that careful configuration of parameters, such as availability frequency and the number of endpoints, is needed for balancing system throughput, queue size, and model predictive performance. Proper adjustment of these parameters helps to optimize event consumption per second, minimize processing delays, and prevent excessive data accumulation while maintaining model accuracy. Additionally, each learner's intrinsic characteristics and model size significantly affect system behavior. For instance, NB models, which are less sensitive to parameter variations, show more stable performance. Computationally heavier models, such as those based on ARF and AdaBoost, exhibit less performance fluctuation due to parameter changes. In contrast, HT models, with their favorable balance of elasticity and relatively small sizes, are more impacted by parameter variations. It is essential to note that these results were obtained under significant hardware constraints to highlight the main system bottlenecks. Overall, the choice of configuration parameters is fundamental to the system's performance, with optimal results achieved through detailed and customized parameter settings that account for learner characteristics and production environment needs.

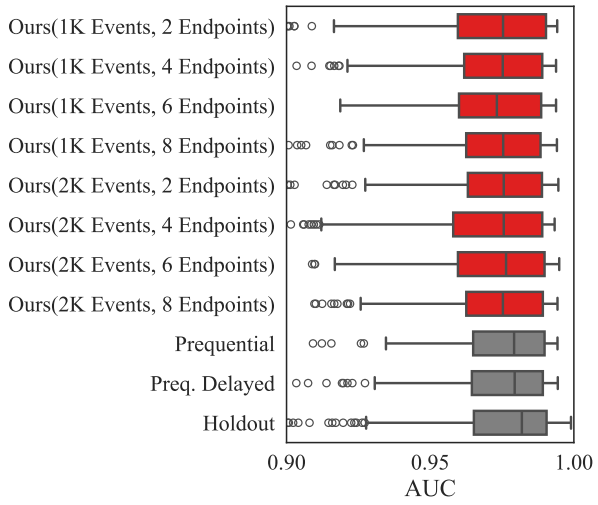


Fig. 11: Accuracy distribution of the ARF classifier on the AGR-a dataset considering multiple architecture configurations vs traditional evaluation setups.

The frequency of model versioning is an important aspect of balancing system performance and accuracy. Operators should define a model versioning triggering frequency that ensures accuracy is maintained in proportion to the number of deployed inference endpoints. However, it is essential to consider the trade-offs of frequent model versioning. While more frequent updates may improve responsiveness to changes in data, they can also negatively affect inference throughput by introducing delays and increasing the load on the model update queue. As a result, the operator should consider finding the operation point that can optimize both accuracy and inference throughput when deploying our proposed architecture.

VII. Conclusion

This paper presents a novel architecture for deploying SL models under the MLOps framework, which incorporates incremental updates and frequent model versioning. The architecture suits various systems, including cloud environments, because it is based on containerized microservices that ensure portability and flexibility. Unlike traditional approaches that periodically update models in batches, our architecture addresses the challenge of continuously adapting to data changes through incremental updates. The implementation provides a scalable inference service that maintains efficiency while addressing key issues such as rapidly updating models, managing multiple model versions, and handling large data streams. Extensive experiments validated the architecture, highlighting the critical role of parameter configuration in the balance between throughput, queue size, and predictive performance. The architecture demonstrated robustness, maintaining low model availability times and high predictive accuracy, even in demanding scenarios

Future work will evaluate the architecture's performance with multiple concurrent models, examine its ability to manage different data flows and model versioning and optimize its adaptability and scalability for more complex situations.

The source code and used datasets are publicly available for download at https://github.com/mgrodrigues001/MLOps-Architecture.

Acknowledgment

This work was partially sponsored by the Brazilian National Council for Scientific and Technological Development (CNPq), grants nº 304990/2021-3, 407879/2023-4, and 302937/2023-4.

References

[1] E. C. P. Neto, S. Dadkhah, S. Sadeghi, H. Molyneaux, and A. A. Ghorbani, "A review of machine learning (ml)-based iot security in healthcare: A dataset perspective," *Computer Communications*, vol. 213, p. 61–77, Jan. 2024. [Online]. Available: http://dx.doi.org/10.1016/j.comcom.2023.11.002

[2] J. A. Pruneski, R. J. Williams, B. U. Nwachukwu, P. N. Ramkumar, A. M. Kiapour, R. K. Martin, J. Karlsson, and A. Pareek, "The development and deployment of machine learning models," *Knee Surgery, Sports Traumatology, Arthroscopy*, vol. 30, no. 12, p. 3917–3923, Sep. 2022. [Online]. Available: http://dx.doi.org/10.1007/s00167-022-07155-4

[3] V. Chaoji, R. Rastogi, and G. Roy, "Machine learning in the real world," *Proceedings of the VLDB Endowment*, vol. 9, no. 13, p. 1597–1600, Sep. 2016. [Online]. Available: http://dx.doi.org/10.14778/3007263.3007318

[4] R. Ashmore, R. Calinescu, and C. Paterson, "Assuring the machine learning lifecycle: Desiderata, methods, and challenges," *ACM Computing Surveys*, vol. 54, no. 5, p. 1–39, May 2021. [Online]. Available: http://dx.doi.org/10.1145/3453444

[5] A. M. Burgueño-Romero, C. Barba-González, and J. F. Aldana-Montes, "Big data-driven mlops workflow for annual high-resolution land cover classification models," *Future Generation Computer Systems*, vol. 163, p. 107499, Feb. 2025. [Online]. Available: http://dx.doi.org/10.1016/j.future.2024.107499

[6] P. P. Shinde and S. Shah, "A review of machine learning and deep learning applications," in *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*. IEEE, Aug. 2018. [Online]. Available: http://dx.doi.org/10.1109/ICCUBEA.2018.8697857

[7] MLflow Project, "An open source platform for the machine learning lifecycle," 2018. [Online]. Available: https://mlflow.org/

[8] Amazon Web Services, Inc, "Amazon sagemaker," 2017. [Online]. Available: https://aws.amazon.com/sagemaker/

[9] Google, "Google cloud ai plataform," 2018. [Online]. Available: https://console.cloud.google.com/marketplace/product/google-cloud-platform/cloud-machine-learning-engine?project=curso-403115

[10] O. Bentaleb, A. S. Z. Belloum, A. Sebaa, and A. El-Maouhab, "Containerization technologies: taxonomies, applications and challenges," *The Journal of Supercomputing*, vol. 78, no. 1, p. 1144–1181, Jun. 2021. [Online]. Available: http://dx.doi.org/10.1007/s11227-021-03914-1

[11] H. M. Gomes, J. P. Barddal, F. Enembreck, and A. Bifet, "A survey on ensemble learning for data stream classification," *ACM Computing Surveys*, vol. 50, no. 2, p. 1–36, Mar. 2017. [Online]. Available: http://dx.doi.org/10.1145/3054925

[12] S. Agrahari and A. K. Singh, "Concept drift detection in data stream mining: A literature review," *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 10, p. 9523–9540, Nov. 2022. [Online]. Available: http://dx.doi.org/10.1016/j.jksuci.2021.11.006

[13] S. R. Upadhyaya, "Parallel approaches to machine learning—a comprehensive survey," *Journal of Parallel and Distributed Computing*, vol. 73, no. 3, p. 284–292, Mar. 2013. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2012.11.001

[14] F. Kiaee and E. Arianyan, "Joint vm and container consolidation with auto-encoder based contribution extraction of decision criteria in edge-cloud environment," *Journal of Network and Computer Applications*, vol. 233, p. 104049, Jan. 2025. [Online]. Available: http://dx.doi.org/10.1016/j.jnca.2024.104049

[15] S. Garg, P. Pundir, G. Rathee, P. Gupta, S. Garg, and S. Ahlawat, "On continuous integration / continuous delivery for automated deployment of machine learning models using mlops," in *2021 IEEE Fourth International Conference on Artificial Intelligence and Knowledge Engineering (AIKE)*. IEEE, Dec. 2021. [Online]. Available: http://dx.doi.org/10.1109/AIKE52691.2021.00010

[16] N. J. Yadwadkar, F. Romero, Q. Li, and C. Kozyrakis, "A case for managed and model-less inference serving," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '19. ACM, May 2019. [Online]. Available: http://dx.doi.org/10.1145/3317550.3321443

[17] M. Barry, J. Montiel, A. Bifet, S. Wadkar, N. Manchev, M. Halford, R. Chiky, S. E. Jaouhari, K. B. Shakman, J. A. Fehaily, F. Le Deit, V.-T. Tran, and E. Guerizec, "Streammlops: Operationalizing online learning for big data streaming amp; real-time applications," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, Apr. 2023. [Online]. Available: http://dx.doi.org/10.1109/ICDE55515.2023.00272

[18] A. L. Suárez-Cetrulo, D. Quintana, and A. Cervantes, "A survey on machine learning for recurring concept drifting data streams," *Expert Systems with Applications*, vol. 213, p. 118934, Mar. 2023. [Online]. Available: http://dx.doi.org/10.1016/j.eswa.2022.118934

[19] A. Bifet, S. Maniu, J. Qian, G. Tian, C. He, and W. Fan, "Streamdm: Advanced data mining in spark streaming," in *2015 IEEE International Conference on Data Mining Workshop (ICDMW)*. IEEE, Nov. 2015. [Online]. Available: http://dx.doi.org/10.1109/ICDMW.2015.140

[20] M. Komisarek, M. Choraś, R. Kozik, and M. Pawlicki, "Real-time stream processing tool for detecting suspicious network patterns using machine learning," in *Proceedings of the 15th International Conference on Availability, Reliability and Security*, ser. ARES 2020. ACM, Aug. 2020. [Online]. Available: http://dx.doi.org/10.1145/3407023.3409189

[21] A. Paleyes, R.-G. Urma, and N. D. Lawrence, "Challenges in deploying machine learning: A survey of case studies," *ACM Computing Surveys*, vol. 55, no. 6, p. 1–29, Dec. 2022. [Online]. Available: http://dx.doi.org/10.1145/3533378

[22] I. L. Markov, H. Wang, N. S. Kasturi, S. Singh, M. R. Garrard, Y. Huang, S. W. C. Yuen, S. Tran, Z. Wang, I. Glotov, T. Gupta, P. Chen, B. Huang, X. Xie, M. Belkin, S. Uryasev, S. Howie, E. Bakshy, and N. Zhou, "Looper: An end-to-end ml platform for product decisions," in *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, ser. KDD '22, vol. 28. ACM, Aug. 2022, p. 3513–3523. [Online]. Available: http://dx.doi.org/10.1145/3534678.3539059

[23] J. Lu, A. Liu, F. Dong, F. Gu, J. Gama, and G. Zhang, "Learning under concept drift: A review," *IEEE Transactions on Knowledge and Data Engineering*, p. 1–1, 2018. [Online]. Available: http://dx.doi.org/10.1109/TKDE.2018.2876857

[24] J. P. Barddal, H. M. Gomes, F. Enembreck, and B. Pfahringer, "A survey on feature drift adaptation: Definition, benchmark, challenges and future directions," *Journal of Systems and Software*, vol. 127, p. 278–294, May 2017. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2016.07.005

[25] P. M. Gonçalves, S. G. de Carvalho Santos, R. S. Barros, and D. C. Vieira, "A comparative study on concept drift detectors," *Expert Systems with Applications*, vol. 41, no. 18, p. 8144–8156, Dec. 2014. [Online]. Available: http://dx.doi.org/10.1016/j.eswa.2014.07.019

[26] A. R. Moya, B. Veloso, J. Gama, and S. Ventura, "Improving hyperparameter self-tuning for data streams by adapting an evolutionary approach," *Data Mining and Knowledge Discovery*, vol. 38, no. 3, p. 1289–1315, Dec. 2023. [Online]. Available: http://dx.doi.org/10.1007/s10618-023-00997-7

[27] F. Banar, A. Tabatabaei, and M. Saleh, "Stream data classification with hoeffding tree: An ensemble learning approach," in *2023 9th International Conference on Web Research (ICWR)*. IEEE, May 2023. [Online]. Available: http://dx.doi.org/10.1109/ICWR57742.2023.10139228

[28] J. Gama, R. Sebastião, and P. P. Rodrigues, "On evaluating stream learning algorithms," *Machine Learning*, vol. 90, no. 3, p. 317–346, Oct. 2012. [Online]. Available: http://dx.doi.org/10.1007/s10994-012-5320-9

[29] H. M. Gomes, J. Read, A. Bifet, J. P. Barddal, and J. Gama, "Machine learning for streaming data: state of the art, challenges, and opportunities," *ACM SIGKDD Explorations Newsletter*, vol. 21, no. 2, p. 6–22, Nov. 2019. [Online]. Available: http://dx.doi.org/10.1145/3373464.3373470

[30] S. Anandan, M. Bogoevici, G. Renfro, I. Gopinathan, and P. Peralta, "Spring xd: a modular distributed stream and batch processing system," in *Proceedings of the 9th ACM International Conference on Distributed*

Event-Based Systems, ser. DEBS '15. ACM, Jun. 2015, p. 217–225. [Online]. Available: http://dx.doi.org/10.1145/2675743.2771879
[31] The Apache Software Foundation, "Apache airflow," 2014. [Online]. Available: https://airflow.apache.org/
[32] The Kubeflow Authors, "Kubeflow," 2018. [Online]. Available: https://www.kubeflow.org/
[33] Seldon Technologies Limited, "Seldon," 2014. [Online]. Available: https://www.seldon.io/
[34] Microsoft Corporation, "Azure machine learning," 2018. [Online]. Available: https://azure.microsoft.com/
[35] The University of Waikato, "Moa - massive online analysis," 2010. [Online]. Available: https://moa.cms.waikato.ac.nz/
[36] Apache Software Foundation, "Samoa - scalable advanced massive online analysis," 2014. [Online]. Available: https://incubator.apache.org/projects/samoa.html
[37] ——, "Storm," 2014. [Online]. Available: https://storm.apache.org/
[38] ——, "Samza," 2018. [Online]. Available: https://samza.apache.org/
[39] Jacob Montiel and Jesse Read and Albert Bifet and Talel Abdessalem, "Scikit-multiflow: A multi-output streaming framework," *Journal of Machine Learning Research*, vol. 19, no. 72, pp. 1–5, 2018. [Online]. Available: http://jmlr.org/papers/v19/18-251.html
[40] Python Software Foundation, "Creme," 2020. [Online]. Available: https://pypi.org/project/creme/
[41] Montiel, Jacob and Halford, Max and Mastelini, Saulo Martiello and Bolmier, Geoffrey and Sourty, Raphael and Vaysse, Robin and Zouitine, Adil and Gomes, Heitor Murilo and Read, Jesse and Abdessalem, Talel and others, "River: machine learning for streaming data in python," 2021. [Online]. Available: https://riverml.xyz/
[42] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
[43] W. Jamil, N.-C. Duong, W. Wang, C. Mansouri, S. Mohamad, and A. Bouchachia, "Scalable online learning for flink: Solma library," in *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*, ser. ECSA '18, vol. 58. ACM, Sep. 2018, p. 1–4. [Online]. Available: http://dx.doi.org/10.1145/3241403.3241438
[44] Apache Software Foundation, "Apache flink: Stateful computations over data streams," 2014. [Online]. Available: https://flink.apache.org/
[45] D. Xu, D. Wu, X. Xu, L. Zhu, and L. Bass, "Making real time data analytics available as a service," in *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures*, ser. CompArch '15. ACM, May 2015, p. 73–82. [Online]. Available: http://dx.doi.org/10.1145/2737182.2737186
[46] Apache Software Foundation, "Apache spark," 2013. [Online]. Available: https://spark.apache.org/
[47] ——, "Mllib: Machine learning library," 2014. [Online]. Available: https://spark.apache.org/mllib/
[48] R. Minon, J. Diaz-de Arcaya, A. I. Torre-Bastida, G. Zarate, and A. Moreno-Fernandez-de Leceta, "Mlpacker: A unified software tool for packaging and deploying atomic and distributed analytic pipelines," in *2022 7th International Conference on Smart and Sustainable Technologies (SpliTech)*, vol. 20. IEEE, Jul. 2022, p. 1–6. [Online]. Available: http://dx.doi.org/10.23919/SpliTech55088.2022.9854211
[49] R. Fayos-Jordan, S. Felici-Castell, J. Segura-Garcia, J. Lopez-Ballester, and M. Cobos, "Performance comparison of container orchestration platforms with low cost devices in the fog, assisting internet of things applications," *Journal of Network and Computer Applications*, vol. 169, p. 102788, Nov. 2020. [Online]. Available: http://dx.doi.org/10.1016/j.jnca.2020.102788
[50] R. Ramanath, K. Salomatin, J. D. Gee, K. Talanine, O. Dalal, G. Polatkan, S. Smoot, and D. Kumar, "Lambda learner: Fast incremental learning on data streams," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery amp; Data Mining*, ser. KDD '21. ACM, Aug. 2021. [Online]. Available: http://dx.doi.org/10.1145/3447548.3467172
[51] F. Carcillo, A. Dal Pozzolo, Y.-A. Le Borgne, O. Caelen, Y. Mazzer, and G. Bontempi, "Scarff: A scalable framework for streaming credit card fraud detection with spark," *Information Fusion*, vol. 41, p. 182–194, May 2018. [Online]. Available: http://dx.doi.org/10.1016/j.inffus.2017.09.005
[52] J. Kranjc, R. Orač, V. Podpečan, N. Lavrač, and M. Robnik-Šikonja, "Clowdflows: Online workflows for distributed big data mining," *Future Generation Computer Systems*, vol. 68, p. 38–58, Mar. 2017. [Online]. Available: http://dx.doi.org/10.1016/j.future.2016.07.018
[53] S. Garcia-Rodriguez, M. Alshaer, and C. Gouy-Pailler, "Streamer: A powerful framework for continuous learning in data streams," in *Proceedings of the 29th ACM International Conference on Information amp; Knowledge Management*, ser. CIKM '20. ACM, Oct. 2020. [Online]. Available: http://dx.doi.org/10.1145/3340531.3417427
[54] C. Martín, P. Langendoerfer, P. S. Zarrin, M. Díaz, and B. Rubio, "Kafka-ml: Connecting the data stream with ml/ai frameworks," *Future Generation Computer Systems*, vol. 126, p. 15–33, Jan. 2022. [Online]. Available: http://dx.doi.org/10.1016/j.future.2021.07.037
[55] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: a low-latency online prediction serving system," in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'17. USA: USENIX Association, 2017, p. 613–627.
[56] D. Baylor, E. Breck, H.-T. Cheng, N. Fiedel, C. Y. Foo, Z. Haque, S. Haykal, M. Ispir, V. Jain, L. Koc, C. Y. Koo, L. Lew, C. Mewald, A. N. Modi, N. Polyzotis, S. Ramesh, S. Roy, S. E. Whang, M. Wicke, J. Wilkiewicz, X. Zhang, and M. Zinkevich, "Tfx: A tensorflow-based production-scale machine learning platform," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '17. ACM, Aug. 2017. [Online]. Available: http://dx.doi.org/10.1145/3097983.3098021
[57] A. Pareek, B. Zhang, and B. Khaladkar, "A demonstration of striim a streaming integration and intelligence platform," in *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems*, ser. DEBS '19. ACM, Jun. 2019, p. 236–239. [Online]. Available: http://dx.doi.org/10.1145/3328905.3332519
[58] Apache Software Foundation, "Apache kafka," 2011. [Online]. Available: https://kafka.apache.org/
[59] Elastic, "Elasticsearch," 2010. [Online]. Available: https://www.elastic.co/elasticsearch/
[60] ——, "Kibana," 2013. [Online]. Available: https://www.elastic.co/kibana/
[61] Apache Software Foundation, "Hadoop distributed file system (hdfs)," 2006. [Online]. Available: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html
[62] Pivotal Software, "Rabbitmq," 2007. [Online]. Available: https://www.rabbitmq.com/
[63] Apache Software Foundation, "Apache hadoop," 2006. [Online]. Available: https://hadoop.apache.org/
[64] Redis Labs, "Redis," 2009. [Online]. Available: https://redis.io/
[65] Apache Software Foundation, "Apache zookeeper," 2010. [Online]. Available: https://zookeeper.apache.org/
[66] InfluxData, "Influxdb," 2013. [Online]. Available: https://www.influxdata.com/products/influxdb/
[67] Google Brain Team, "Tensorflow: An open-source machine learning framework," 2015. [Online]. Available: https://www.tensorflow.org/
[68] PyTorch Foundation, "Pytorch: An open-source machine learning library," 2016. [Online]. Available: https://pytorch.org/
[69] P. Horchulhack, E. K. Viegas, A. O. Santin, F. V. Ramos, and P. Tedeschi, "Detection of quality of service degradation on multi-tenant containerized services," *Journal of Network and Computer Applications*, vol. 224, p. 103839, Apr. 2024. [Online]. Available: http://dx.doi.org/10.1016/j.jnca.2024.103839
[70] F.-J. Yang, "An implementation of naive bayes classifier," in *2018 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, Dec. 2018. [Online]. Available: http://dx.doi.org/10.1109/CSCI46756.2018.00065
[71] Canonical Ltd., "Microk8s: Lightweight kubernetes for developers and devops," 2018. [Online]. Available: https://microk8s.io/
[72] Pallets Projects, "Flask: A python microframework," 2010. [Online]. Available: https://flask.palletsprojects.com/
[73] R. Agrawal, T. Imielinski, and A. Swami, "Database mining: a performance perspective," *IEEE Transactions on Knowledge and Data Engineering*, vol. 5, no. 6, p. 914–925, Dec. 1993. [Online]. Available: http://dx.doi.org/10.1109/69.250074
[74] N. Asghar, "Yelp dataset challenge: Review rating prediction," 2016. [Online]. Available: https://arxiv.org/abs/1605.05362
[75] N. Oza, "Online bagging and boosting," in *2005 IEEE International Conference on Systems, Man and Cybernetics*, vol. 3. IEEE, p. 2340–2345. [Online]. Available: http://dx.doi.org/10.1109/ICSMC.2005.1571498

Miguel G. Rodrigues is a Computer Science MSc. candidate at the Pontifical Catholic University of Paraná (PUCPR). He received his bachelor's degree in Big Data and Data Science from PUCPR in 2021. His research includes stream learning applications, Big Data, Distributed Systems, and MLOps.

Eduardo K. Viegas received his BS in computer science in 2013, the MSC in computer science in 2016 from PUCPR, and the Ph.D. degree from PUCPR in 2018. He is an associate professor of the Graduate Program in Computer Science (PPGIa). His research interests include machine learning, network analytics, and computer security.

Altair O. Santin received the BS degree in Computer Engineering from the PUCPR in 1992, the M.Sc. degree from UTFPR in 1996, and the Ph.D. degree from UFSC in 2004. He is a full professor of the Graduate Program in Computer Science (PPGIa) and head of the Security & Privacy Lab (SecPLab) at PUCPR. He is an IEEE, ACM, and Brazilian Computer Society member. https://orcid.org/0000-0002-2341-2177.

Fabricio Enembreck holds a degree in Computer Science from the State University of Ponta Grossa (1997), a master's degree in Computer Science from the Pontifical Catholic University of Paraná (1999), and a PhD in Information and Systems Technologies from the Universite de Technologie de Compiegne (2003). He is currently an Assistant Professor at the Pontifical Catholic University of Paraná. He has experience in the area of Computer Science, with an emphasis on Computer Systems. He works mainly on the following topics: assistant agents, autonomous agents, adaptive autonomous agents, distributed artificial intelligence, multi-agent systems, and machine learning.