Crayfish: Navigating the Labyrinth of Machine Learning Inference in Stream Processing Systems

Sonia Horchidan
KTH Royal Institute of Technology
sfhor@kth.se

Po Hao Chen
Brown University
pch@brown.edu

Emmanouil Kritharakis
Boston University
ekrithar@bu.edu

Paris Carbone
KTH Royal Institute of Technology
parisc@kth.se

Vasiliki Kalavri
Boston University
vkalavri@bu.edu

ABSTRACT

As Machine Learning predictions are increasingly being used in business analytics pipelines, integrating stream processing with model serving has become a common data engineering task. Despite their synergies, separate software stacks typically handle streaming analytics and model serving. Systems for data stream management do not support ML inference out-of-the-box, while model-serving frameworks have limited functionality for continuous data transformations, windowing, and other streaming tasks. As a result, developers are left with a design space dilemma whose trade-offs are not well understood. This paper presents Crayfish, an extensible benchmarking framework that facilitates designing and executing comprehensive evaluation studies of streaming inference pipelines. We demonstrate the capabilities of Crayfish by studying four data processing systems, three embedded libraries, three external serving frameworks, and two pre-trained models. Our results prove the necessity of a standardized benchmarking framework and show that (1) even for serving tools in the same category, the performance can vary greatly and, sometimes, defy intuition, (2) GPU accelerators can show compelling improvements for the serving task, but the improvement varies across tools, and (3) serving alternatives can achieve significantly different performance, depending on the stream processors they are integrated with.

1 INTRODUCTION

A plethora of data management issues in Machine Learning (ML) application scenarios have recently concerned our community [32, 41], including data cleaning and quality verification [31, 40, 44, 47], in-database learning [35, 55], optimizing ML over relational data [7, 34], learning on private data [21, 27], and model tuning and lifecycle management [1, 37], among others. Nonetheless, little attention has been paid to the intricacies involved in the rising use of ML applications on streaming data. Existing studies have merely focused on model training in this context [36]; yet, the fundamental complexities of combining data stream processing technologies with ML inference tools have been greatly overlooked. In this paper, we review the state of affairs in streaming ML inference and study the problem of model serving within data stream processing pipelines.

Despite growing needs for integrating ML predictions into online analytics pipelines [2, 8, 17], no canonical methodology exists today for developing such applications. On one hand, modern stream processing systems (SPSs), such as Apache Flink [9] and Kafka Streams [43], provide advanced functionality for online data transformations, aggregation, and windowing but lack native support for model lifecycle management and hardware acceleration (e.g., GPU, SIMD), which is paramount when dealing with frequent vectorized ML operations [53]. On the other hand, ML serving frameworks, such as TensorFlow Serving [39] and TorchServe [11], offer limited features for data transformations and have no support for streaming computations. As a result, developers are left with two main design choices whose trade-offs are not well understood. They can embed ML models into stream processing operators to enable the creation of end-to-end pipelines but with high engineering and maintenance costs for inference-critical features such as autoscaling, model versioning, and hardware acceleration. Alternatively, they can instruct the SPS to forward inference requests to external serving frameworks at the cost of increased complexity, additional resources, and potentially weaker fault-tolerance guarantees.

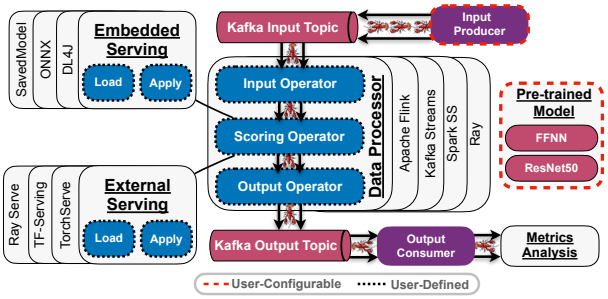

Figure 1: Crayfish Overview.

Unfortunately, evaluating these alternative design choices is not straightforward, as users need to develop, deploy, and test multiple combinations of complex tools to find the most suitable solution for their use case. The different architectures of existing SPSs and the obscure performance characteristics of the serving approaches only exacerbate the problem. For instance, even though most modern SPS designs have converged to common dataflow programming abstractions [3, 6], their underlying execution engines vary considerably in the way they handle batching, asynchronous requests, backpressure, and state management [14, 20, 51]. Understanding the performance trade-offs between embedded and external model serving designs is also challenging without extensive experimentation. Embedding models in a streaming operator enables local inference calls but requires interoperability libraries, such as DeepLearning4j [30] and ONNX (Open Neural Network Exchange) [16, 28], which rely on performance-costly foreign function interfaces. Conversely, external serving frameworks expose models via REST-like APIs

that may also introduce overheads. Existing performance studies and benchmarking tools have only considered data stream workloads and ML serving in isolation.

Our paper addresses this problem by proposing `Crayfish`, a new benchmark framework for the systematic evaluation of ML inference in stream processing systems, and presents the first comprehensive experimental study conducted using `Crayfish`. Further, `Crayfish` can serve as testing grounds during the model fine-tuning phase, such that performance metrics can be optimized together with accuracy measurements. Figure 1 shows an overview of `Crayfish`'s capabilities. We carefully design the framework to be extensible so developers can easily integrate their SPS, serving tool, and pre-trained model of choice. At the moment of writing, `Crayfish` already supports: (1) four popular (stream) data processing systems, namely Apache Flink, Apache Spark, Kafka Streams, and Ray, (2) three interoperability libraries (ONNX, DL4J, and SavedModel), and (3) three external serving tools (TensorFlow Serving, TorchServe, and Ray Serve). Moreover, we provide a complete toolkit for automating the deployment of `Crayfish` in the cloud and quickly assessing performance results, including data connectors and generators, model loaders, Docker containers, and a metrics collection component. We release the `Crayfish`'s source code alongside the setup and evaluation configuration details as open source[1].

Our contributions. We introduce `Crayfish` in §3. Next, in §4, we design and conduct the first quantitative analysis of popular tools for serving pre-trained ML models in production-level stream processing frameworks and present the results in §5, where we provide an in-depth analysis of the trade-offs of the chosen methods. Following the description of our systematic evaluation, we discuss the implications of the takeaways with respect to the data management problems incurred by the task of streaming model serving in §6. Furthermore, in §7, we list the features worth adopting when designing new streaming systems that enable ML inference and highlight open research challenges.

Summary of major findings. The key findings of our experimental study can be summarized as follows:

- There is significant performance variation among serving frameworks of the same type (i.e., embedded and external) for the same SPS.
- There is no clear performance dichotomy between embedded and external approaches for the same SPS. That is, external serving can achieve lower latency than embedded designs under some conditions.
- All examined configurations experience benefits from GPU acceleration during inference, albeit to varying extents.
- A given serving framework may exhibit considerably different performance depending on the SPS it is integrated with.

2 INFERENCE FOR STREAM PROCESSORS

This section identifies the options available to integrate model-serving tasks into SPSs. Furthermore, we motivate the necessity of a standardized benchmarking tool.

2.1 Embedded and External Serving

Preliminary research described two dominant directions for model serving when interfacing with event-based stream processing systems based on the available integration layer type [25]: *embedded* and *external* serving. These alternatives cover many use cases and can serve as building blocks for complex scenarios. By studying these two alternatives in isolation, practitioners can gain valuable insights to inform the design of complex, more intricate use cases and deployment pipelines.

[1]https://github.com/soniahorchidan/crayfish23

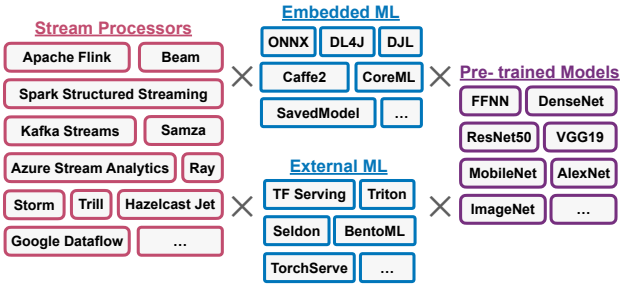

Figure 2: Example of configuration alternatives to compare performance when serving image classification models over data streams.

Embedded serving lets the stream processor manage the entire model serving lifecycle. Typically, the pre-trained model is loaded into the managed state of an operator in the system and the inference runs embedded in the same process as the hosting SPS. However, an interoperability library (such as the ones described in §3.4.2) is required to address the potential incompatibility between stream processors and pre-trained models, since the former are commonly implemented using JVM-based languages. This library should provide primitives for model loading and inference, whereas the SPS is responsible for providing auxiliary production features such as scalability and monitoring.

External serving approaches delegate model serving operations to a dedicated inference service executed as a standalone process. The stream processor submits input data points over the network to the inference service and receives the output of the inference typically via a REST API. Managing and scaling the inference lifecycle is, therefore, operated by the specialized inference service. We delve into external serving approaches in §3.4.3.

2.2 Motivation

The motivation behind the need for a benchmarking framework is two-fold. Firstly, it is unfeasible to test all possible combinations of stream processors and serving alternatives for a set of pre-trained models, yet essential to determine the optimal performance. Secondly, fine-tuning a given ML model to strike a good balance between accuracy and expected latency is challenging but is essential in many real-world applications. We now discuss the two motivating scenarios.

2.2.1 Possible Combinations. A configuration includes selecting an SPS and a model serving tool of choice. Given a number of pre-trained models M to use, the possible configurations can be $SP \times (E + X) \times M$ where SP is the number of available stream processors, E is the number of embedded serving tools, and X is the number of considered external servers. Consider the example in Figure 2, which shows the streaming configurations for serving a number of popular image classification models with roughly 12 candidate stream processors, 6 embedded machine learning (ML) libraries, and 5 external tools. This amounts to a staggering 924 performance tests to integrate and implement, which is nearly infeasible in terms of engineering complexity. Designing accurate benchmarking software is challenging for most system developers, especially when several systems are

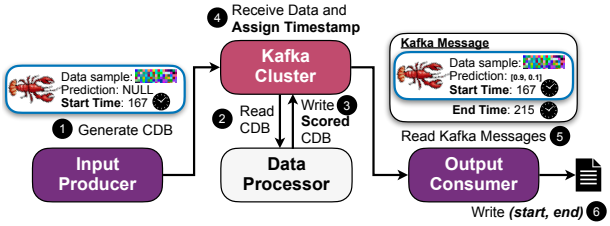

Figure 3: `Crayfish` data generation and timestamps recording. CDB Stands for CrayfishDataBatch.

involved in the process. In this case, a benchmarking framework can help quickly identify the most suitable combination of tools, given a set of latency/throughput constraints.

2.2.2 The latency-accuracy trade-off. In situations demanding real-time predictions for critical decision-making, such as many Internet of Things (IoT) applications, the challenge lies in finding the right balance between ensuring accurate predictions and meeting stringent low latency requirements [5]. The standard ML training process has traditionally prioritized the pursuit of peak accuracy, often sidelining considerations related to latency until the model preparation phase is completed. Here, a benchmarking framework can close the gap by simulating the target production environment and quantifying the expected inference latency. Such a tool can empower data scientists to efficiently iterate and fine-tune their models, ensuring that they not only achieve accuracy benchmarks but also conform to latency constraints.

3 THE CRAYFISH FRAMEWORK

The `Crayfish` framework is designed to provide a standardized way to evaluate and compare the performance of different serving configurations deployed in a chosen stream processing system in terms of metrics such as latency, throughput, and scalability. Similarly to the literature, the serving logic in `Crayfish` is defined as a function determined solely by the inputs and the chosen pre-trained model [33]. This section describes the design and implementation details of `Crayfish`.

3.1 Architecture

Figure 1 shows the main components of `Crayfish`. All measurements in `Crayfish` are orchestrated using an event-based execution, from an input to an output message log. To facilitate this, `Crayfish` makes use of an input workload producer component, which feeds event streams to a data processor component through a message broker. The data processor contains all system-specific implementations of model serving, including a selection of serving alternative configurations covering both embedded and external inference arrangements and pre-trained models. All scored entries arrive at an output consumer component through the message broker where `Crayfish` logs all performance measurements.

Each benchmark in `Crayfish` is composed of discrete units of computation, representing fixed batches of scoring requests. A *CrayfishDataBatch* contains a batch of data points alongside the creation timestamp, which is used in computing end-to-end latencies. The latency computation process is described in detail in §3.3. `Crayfish` uses JSON serialization throughout the data pipeline for simplicity and flexibility.

The *input producer* component can be configured to (1) generate synthetic input streams according to user-defined specifications or (2) read real datasets. The configurable input rate is designed to simulate workloads with constant rate or periodic bursts. The description of these parameters is included in Table 1. The input producer writes each generated *CrayfishDataBatch* to the *Kafka Input Topic* as one batch at a time.

Table 1: Configuration parameters for `Crayfish` experiments.

Notation	Description
isz	Shape of the input data points generated.
bsz	The number of data points generated in one event.
ir	Constant input rate used in the data generation (events/s).
bd	Burst Duration (s)†.
tbb	Time Between Bursts (s)†.
mp	Number of workers used for inference.

† Configured only for input rate generation with periodic bursts.

Next, the *data processor* component refers to the combination of the stream processor and serving tool chosen for the benchmark, which is also referred to as the system under test (SUT) throughout this paper. For extensibility, the data processor in `Crayfish` uses an adapter pattern, allowing for system-specific implementation for each of its operators. There are three main operators: an input operator (i.e., source) for data reads, a scoring operator that performs inference, and an output operator (i.e., the sink) for data writes. The *embedded server* allows to load and apply the pre-trained model via an interoperability library, while the *external server* provides abstractions for HTTP/gRPC-based requests to specialized serving microservices.

The *output consumer* component defines a Kafka consumer responsible for reading the data from the *Kafka Output Topic* and extracting the end-to-end latency per *CrayfishDataBatch*. When the experiments are completed, the *metrics analyzer* component can be invoked to produce performance statistics.

The SUT's components are user-defined. Therefore, we showcase `Crayfish`' capabilities by implementing adaptors for a list of stream processors and serving alternatives.

3.2 Core Abstractions

`Crayfish` provides a set of core programming abstractions to extend the benchmark with more systems and libraries of choice. These interfaces are depicted in Figure 1 as user-defined components. Furthermore, the system can be configured to test pre-trained models of choice (shown as user-configurable in Figure 1).

Stream Processors: `Crayfish` provides an interface to extend the framework functionality with streaming frameworks of choice. Any event-based system that can declare its computation as a Directed Acyclic Graph (DAG) fulfills the requirements of the Crayfish processor. The main operators are the following: 1) *I*: `inputOp` implements the stream ingestion logic from a Kafka topic. 2) *S/E*: `scoringOp` takes the output of `inputOp` as input and executes the serving logic via a flatmap-like operation. `Crayfish` treats embedded (*S*) and external serving separately (*E*), allowing for the stream processor to specify different behavior for each. 3) *O*: The `outputOp` method should accept the scored data points as input and write data to Kafka topics. Furthermore, given that `Crayfish` aims to evaluate scalability considerations, any system implementation of the core abstraction must specify how to establish the parallelism of the streaming computation.

Serving Tools & Models: To ensure compatibility with the new serving tools, `Crayfish` expects libraries to provide the implementation of two methods: `load`, which specifies how the pre-trained model is to be loaded into memory, and `apply`, which obtains a prediction, given a `CrayfishDataBatch` object and a model. Loading a pre-trained model and performing inference

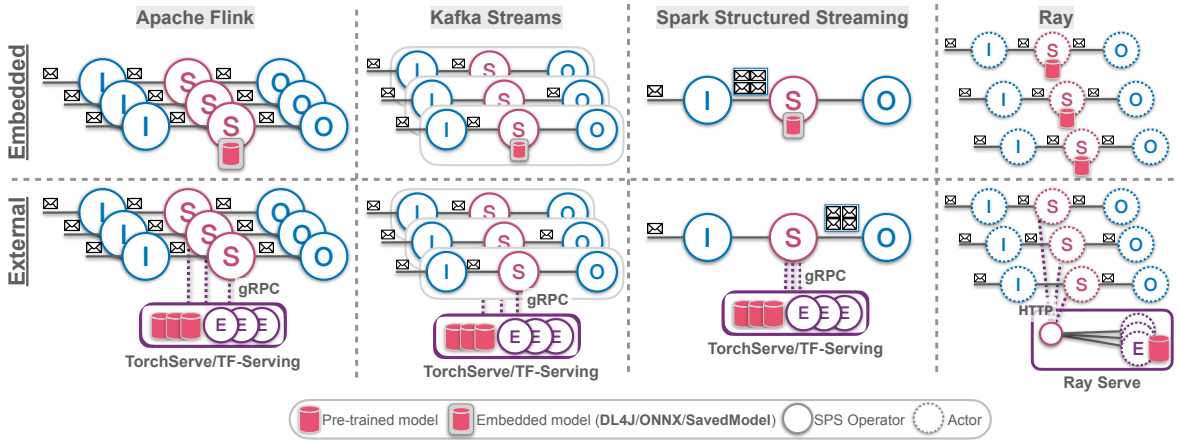

Figure 4: An illustrative overview of all dataflow system task configurations for model serving across systems, depicting how the `Crayfish` tasks are arranged (I: input, S:serving, E: external serving, O:output, model placement).

given input data of a predefined type are minimal functionalities typically part of any model-serving interface. Most existing applications adhere to these assumptions; therefore, the provided abstractions do not hinder the applicability of `Crayfish` to other serving tools. Finally, to support models other than the default ones in `Crayfish`, users can indicate the format and location of any stored model they wish to test via configuration files.

For concrete examples of how to extend `Crayfish` with additional SPSs, model serving tools, and pre-trained models, we invite the readers to consult our public repository.

3.3 Metrics Collection

In this section, we outline `Crayfish`'s metrics collection process, which is depicted in Figure 3. When evaluating the performance of streaming tasks, *throughput* and *end-to-end latency* are traditionally used as the primary metrics. `Crayfish` records two timestamps for each (batch of) generated data point(s). The first timestamp is the *start time*, which is defined as the local clock time of the generation and is recorded by the input producer component. This timestamp is recorded prior to the data being written to the Kafka Input Topic (Step ❶). The second timestamp is the *end time* and is collected on the Kafka side per message (Step ❺). This timestamp is the local time of the broker when the message is appended to the output topic (Kafka's `LogAppendTime` configuration). By using these two timestamps, which are persistently written in step (Step ❻), `Crayfish` computes the time spent by the `Crayfish` data batch in the system.

3.4 Supported Frameworks and Libraries

`Crayfish` currently supports a non-exhaustive list of tools and frameworks curated to cover various applications and simultaneously showcase that the benchmark is general and extensible. This section discusses tools, configurations, and implementation details, which are also depicted in Figure 4. In total, `Crayfish` supports at the moment four stream processors, three embedded libraries, three external ones, and two pre-trained models. The number of combinations tested in our evaluation goes up to 48.

3.4.1 Stream Processing Systems. SPSs are systems purposefully designed to ingest, transform, and apply computations on events that get ingested continuously. In `Crayfish`, SPSs are different implementations of the *data processor* component.

Apache Flink. Apache Flink [9] is a full-fledged distributed stream processing engine for continuous data processing. The Flink adapter in `Crayfish` follows closely the interface described by the *data processor* component. The DAG comprises three operators: a Kafka source, a `map` operator responsible for model scoring, and a Kafka sink. In the case of embedded serving, the map operator loads the pre-trained model in memory using an interoperability library before the streaming job begins. For external serving, the scoring operator sends gRPC requests to an external endpoint and waits for the prediction. We chose Apache Flink as a representative push-pull event-based SPS, which, by default, enables the push-based mechanism. Therefore, data is pushed to the streaming engine as soon as it becomes available. Moreover, as depicted in Figure 4, Flink employs processing pipelining techniques by overlapping the processing of multiple stages to minimize the time taken for data to flow through each step of the DAG. As shown in the figure, scaling up in Apache Flink can be achieved by setting the default parallelism of the DAG, which involves assigning a copy of the dataflow graph to each available worker, which processes a partition/shard of the stream. Alternatively, one can set the operator-level parallelism, which assigns individual operator tasks of the dataflow graph to workers, providing more granular control over the resources allocated to each stage.

Kafka Streams. Kafka Streams [43] is a Java library provided by the Apache Kafka project designed for stream processing computations. Therefore, it uses Kafka topics as the communication channels between different stages of the streaming application. Similarly to Flink, the Kafka Streams `Crayfish` adapter is built as a graph of three chained operators. The source and sink act as Kafka consumers and producers, respectively. We use a `transform` operator to load the model into memory at initialization time and perform the scoring, which operates similarly to the corresponding Flink operator described above. We chose Kafka Streams as a representative pull-based event-based system, where events need to go through the whole processing DAG before requesting a new one for ingestion, as can be noted in Figure 4. Kafka Streams achieves vertical scalability by increasing the number of partitions on the stream application topics.

Spark Structured Streaming. Spark Structured Streaming (Spark SS) [4] offers stream processing capabilities on top of Apache Spark's SQL engine. Unlike the two frameworks described above, Spark SS's computation unit is a micro-batch, a difference noted in Figure 4. At the moment of writing this paper, Spark SS also

offers a continuous processing alternative meant to offer event-based processing. However, since this feature is experimental [45], we decided to employ the micro-batch operation mode. The Spark SS `Crayfish` adapter implements the three interface methods similar to the Flink and Kafka Streams adapters by using specialized abstractions for interfacing with the Kafka topics and a `map` operator for scoring. Spark SS operates on data in micro-batches; therefore, scaling up involves parallelizing the execution of each micro-batch. This consists of splitting ingested data batches into chunks and executing them sequentially by multiple workers. We set the job trigger interval to the minimum possible and use the default append mode configuration to minimize overheads. By default, Spark SS adopts a push-based ingestion policy similar to the one described above for Apache Flink.

3.4.2 Interoperability Libraries. In this section, we will discuss the specific libraries that were integrated by `Crayfish`'s serving tool configuration capabilities. All the serving tools offer APIs for loading and applying the model, thus offering a one-to-one mapping to the interface methods of the `CrayfishModel` interface. General-purpose interoperability libraries such as Py4J or Jython were not considered for this purpose, as they are not specifically designed for model scoring.

DeepLearning4j. The DeepLearning4j (DL4J) [30] library offers capabilities beyond the scope of inference, as it is built for end-to-end deep learning solutions on the JVM. However, for this study, we test the Keras model import functionality using the models stored in H5 format. DeepLearning4j was selected for comparison due to a tight Java integration.

ONNX. The goal of the Open Neural Network Exchange (ONNX) is to offer a standard format to represent ML models across different frameworks [16]. We use the ONNX Runtime in conjunction with models stored in native ONNX format. ONNX was chosen for this study due to its extensive interoperability capabilities.

SavedModel. TensorFlow's solution for the platform-independent deployment of TensorFlow models, SavedModel [50] is a file format and serialization protocol for TensorFlow models. We selected SavedModel as a specialized embedded serving tool that optimizes for one specific format.

3.4.3 Specialized Frameworks. Specialized serving services are tools that wrap pre-trained models into microservices and feature REST-like APIs for model inference, management, versioning, and monitoring in production environments. When used together with SPSs, the latter only require knowledge about the endpoint exposed for inference but are oblivious to ML inference lifecycle changes, such as scaling up or down or model changes. We chose the TensorFlow and PyTorch specialized solutions for this study due to their popularity in the ML community.

TensorFlow Serving. TensorFlow's specialized solution that offers a cloud-based or on-prem system for serving ML models, TensorFlow Serving [39] exposes both REST and `gRPC` APIs for interacting with the served models, facilitating integration with client applications written in different programming languages. We used the gRPC API in this study, as depicted in Figure 4. Scaling up the deployment was implemented by setting the maximum number of threads that can be used to process events concurrently. TensorFlow Serving's inference uses models saved in the SavedModel format.

TorchServe. The PyTorch alternative to TensorFlow Serving, TorchServe [11] features similar functionalities and APIs as the former. Distinctively, it allows users to write additional wrapper code for the inference through Python handlers. TorchServe can serve models obtained with different frameworks and libraries, such as PyTorch, TensorFlow, ONNX, or `scikit-learn`. We used native PyTorch models and the gRPC API for this study, as per Figure 4. Scaling up was achieved by adjusting the number of worker processes used for inference.

3.4.4 Actor-Based Systems. So far, we have considered only stream processing systems. However, we also chose to include Ray, a general-purpose, Python-based distributed computing framework that can support ML-powered applications at scale [38]. Contrary to the SPSs described in §3.4.1, Ray is based on the actor programming paradigm. We implement the Ray adapter using Ray actors for operators to build a processing pipeline reminiscent of the dataflow graphs used in SPSs. As a result, the Ray implementation includes an input actor type that consumes data from Kafka topics, a scoring actor type, and an output actor type that writes data to the output topic. In the case of embedded serving, the scoring actor loads a pre-trained model and applies it to new events. As Ray is a Python-based system, no interoperability library is required to interface with the pre-trained models. Scaling up is handled by spawning actors of each type manually. Regarding external serving, we let the scoring actor handle the HTTP-based communication with an external serving service. Particularly, Ray features Ray Serve, a dedicated library for external model serving [49], which will be employed in our analysis. Figure 4 illustrates the two deployment schemes for Ray.

Ray Serve. Ray Serve enables developers to create and deploy highly scalable, high-performance ML models [49]. Similar to the external serving tools described in §3.4.3, Ray Serve allows users to send inference queries over HTTP. We used this API, and not gRPC for Ray Serve, because the support for the latter is marked as experimental in the documentation [48]. Vertical scalability is achieved by increasing the number of execution replicas.

3.5 Design Decisions

This section provides an overview and discussion of the core decisions regarding the benchmark's assumptions and architecture.

Model Storage. External serving alternatives manage the model storage through the microservice responsible for the inference task. In contrast, for embedded alternatives, `Crayfish` stores the model in memory for several reasons. First, we standardize the comparison among different stream processors since each system may handle persistent storage differently. This study does not aim to compare stream processors in terms of state I/O. Secondly, state management in SPSs usually targets persisting intermediate computation results for recovery and reconfiguration. Since we assume models are immutable and only used for inference, there is no need to pay the overhead of using the state management layer.

Producer-level Batching. `Crayfish` operates on batches of data points as units of computation (i.e., the *CrayfishDataBatch*) throughout the whole pipeline to avoid introducing extra overheads into the latency of the stream processors. This implies that the stream processor of choice will process a `Crayfish` batch as a single event. Therefore, another layer of batching may be applied at the level of the stream processor. We consider this choice equitable because we are solely interested in measuring the performance of the inference; thus, we limit the functionality of the data processor to this task.

Table 2: Specifications for the pre-trained models.

		FNN	*ResNet50*
Input Size		28 x 28	224 x 224 x 3
Output Size		10x1	1000x1
Parameters Number		28 K	23 M
Model Size	ONNX	113 KB	97 MB
	SavedModel	508 KB	101 MB
	Torch	115 KB	98 MB
	H5	133 KB	98 MB

SUT Separation. We adopt an approach similar to Karimov et al. [29] and separate the benchmarking driver from the SUT to provide a level playing field, as various SPSs may have distinct definitions of latency and throughput. To this end, `Crayfish` measures the *start* and *end times* at the data generator and message broker components, outside the scope of the inference adapters. This design decision ensures that the measurements collection logic is decoupled from the logic of the adapters, therefore guaranteeing the correctness of the measurements regardless of user-defined behavior. The `Crayfish` users can, therefore, implement new adapters by focusing solely on the inference task while the framework handles measurements. In our experiments, we opted to execute the components on separate machines. We ensured that the same hosts were re-used across experiments, that they are located in the same LAN, and feature adequate clock synchronization using network time protocol services of GCP (Google Cloud Platform) with sub-millisecond accuracy (detailed in §4.2).

Message Brokers. We have designed our benchmark with Apache Kafka as a publish-subscribe messaging system for two reasons: (1) to decouple the input generation and measurements analysis from the SUT, and (2) to closely imitate real-world situations where such systems are commonly used and require persistent and reliable input sources. This decision aligns with decisions found in the literature [52]. Before performing our experiments, we verify that our Kafka deployment and configuration do not create a bottleneck by confirming that the maximum arrival rate of our experiments can be achieved by the Kafka cluster.

Fair Resource Allocation for Inference. When scaling up, more resources are allocated to the SUT to observe its ability to handle larger workloads. The embedded inference task is performed by the streaming operator thread; there is no control over its allocated resources, and the parallelism can only be set at the level of the serving stage of the pipeline. Accordingly, the resources available to the embedded inference cannot be assigned in isolation to ensure a one-to-one comparison with the external approaches. When comparing embedded and external alternatives, there is no guarantee that provisioning the embedded option with the same total resources used by the SPS and the external serving tool would yield comparable results. Defining "equivalent amounts" is non-trivial since, contrary to external serving tools, embedded serving tools share their resources with the SPS, thus, leading to unfair comparisons. For this reason, our experiments provide more resources in total for the SUT running external serving than the embedded alternatives. `Crayfish` is a suitable tool to investigate this problem; however, extensively evaluating the space goes beyond the scope of this paper.

4 EVALUATION METHODOLOGY

We have conducted an extensive experimental analysis of the frameworks and model serving tools supported by `Crayfish`. The evaluation serves as a showcase of the capabilities of the benchmarking framework. Specifically, we structured section §5 to answer the following research questions.

RQ1: For a given SPS, how do different serving tools compare in terms of latency and throughput or different input rates?

RQ2: How does GPU acceleration influence the end-to-end latency of the serving task on streaming data?

RQ3: How do modern SPSs differ in terms of both features and performance with respect to model inference?

4.1 Workload Design

The benchmarking process is carried out on a per-experiment basis. The configuration parameters can be found in Table 1. We designed three pre-configured workload scenarios focused on throughput and latency, respectively. First, the *open loop* scenario is meant to measure the SUT throughput accurately. The producer sends requests at a predefined fixed input rate (which can be configured). This way, we identify the maximum rate that can be handled by the processor (i.e., sustainable throughput). Furthermore, we monitor the performance of the SUT under high input rates and increasing allocated resources, which, in `Crayfish`, are controlled by the *mp* parameter. Next, the *closed loop* scenario records the SUT latency under low input rates, such that the end-to-end latency is dominated by the inference time. The aim is to measure the inference latency per batch under optimal circumstances while minimizing any additional delays. Lastly, we monitor the performance of the SUT when it comes to periodic bursts of data that exceed the sustainable throughput (ST) of the given configuration. We test the ability of the SUT to recover by measuring the time interval required for the latency to stabilize after a burst. The generator produces input at 110% of the ST during the bursts and at 70% otherwise.

Machine Learning Task. The examples presented in this study are derived from traditional image classification tasks. However, as the focus is on the inference phase, the evaluation is independent of the specific ML model or task. As detailed in §3.2, `Crayfish` can be expanded to explore different ML models, provided they follow the consistent inference pattern of having one input operator, one scoring operator, and one output operator. This setup covers a wide variety of ML tasks.

Synthetic Input Data. We opted for using synthetic data across all experiments, although `Crayfish` is configurable to read real datasets from files stored on disk. The latency of the inference task depends on the size of input events and the model, with data content being irrelevant, thus justifying the decision. Moreover, the data distribution does not impact the results of this study, as inputs are uniformly partitioned across workers. Therefore, our results are relevant for real datasets where the size of the data point matches the dimensions we set for the synthetic data. The data generator is general enough to cover a wide range of ML models, as it produces tensor-like data of user-defined size and shape. If the user aims to test, for instance, the performance of their Convolutional Neural Network (CNN), they can configure the framework to generate 2D or 3D data points, as we did in our evaluation. Similarly, for testing Recurrent Neural Networks (RNN), the generator can be configured to yield sequence-like random data. Autoencoders can also be benchmarked with `Crayfish` to test the performance of producing compact representations. We perform experiments with both constant and bursty rates to stress the SUT, as well as analyze its performance under conditions mimicking realistic scenarios.

Table 3: Stream Processor Configuration Parameters.

Stream Processor	Parameter	Value
Apache Flink 1.15.2	JobManager count	1
	TaskManager count	1
	JobManager Memory	2 GB
	TaskManager Memory	120 GB
	TaskManager Task Slots count	60
Spark Structured Streaming 3.3.2	Driver Count	1
	Executor Instances Count	1
	Driver Memory	8 GB†
	Executor Memory	120 GB
	Executor Cores	60
Kafka Streams 3.2.3	Instances Count	1
	Java Memory	120 GB

† The Spark Structured Streaming Driver has more memory allocated to mitigate Out Of Memory errors.

Pre-trained Models. We choose two diverse image classification models to showcase the impact of factors that can influence the performance of the serving approaches, such as model size and input/output sizes. The first model, *FFNN*, is trained on the Fashion MNIST dataset [54] and is the smallest in our study, having 28K parameters. The input data is represented as images of size `28x28`, and the output is a vector of 10 elements, representing the probabilities of the image belonging to one of ten classes. *FFNN* is a fully-connected neural network with three hidden layers, each consisting of 32 neurons and utilizing a `ReLU` activation function. *ResNet50*, on the other hand, is trained on the ImageNet dataset [15], accepts `224x224x3` images as input, and outputs vectors of probabilities for 1000 classes. *ResNet50*'s network architecture contains 23M parameters and is defined in its corresponding paper [22]. We implemented all models using TensorFlow or PyTorch and converted them into the formats chosen for this study. The chosen models are outlined in Table 2.

4.2 Environment

We conducted experiments using a cluster of 9 VMs deployed on Google Cloud Platform (GCP). Each `Crayfish` component runs on a separate machine. The SPS is deployed on a single VM throughout our evaluation; nonetheless, the design of `Crayfish` does not limit the data processor to single-node deployments and can, therefore, be used to evaluate its performance on a cluster. The network bandwidth is 1 Gbps, and the machines are synchronized using Google's internal NTP service. The average ping time to send one packet as large as one FFNN input data point (3 KB) is 0.945 ms, whereas to send 64 KB is 1.565 ms. All the VMs were equipped with `Intel(R) Xeon(R) CPU @ 2.20GHz` processors and had the following specifications: (1) the 4 Kafka Brokers had 4 vCPUs and 15 GB RAM each, (2) the VM running Zookeeper had 2 vCPUs and 8 GB RAM, (3) the input producer VM was configured with 4 vCPUs and 15 GB RAM, (4) the VM running the data processor component had 64 vCPUs and 240 GB RAM, and (5) the VM running the external serving service has 16 vCPUs and 60 GB RAM. We equip the data processor and external serving VMs with `NVIDIA T4` GPUs for the experiments testing inference enabled by hardware acceleration.

We generate 1M measurements per experiment. However, in the interest of time and due to the large number of tested combinations, we also set a timeout of 15 minutes, which sets a hard limit for the termination frame of each experiment. We discarded the first 25% of the measurements to eliminate the impact of system warmup. We run each experiment twice and report the averages and standard deviations.

4.3 SUT Configuration

The systems and frameworks were evaluated mostly with their default configurations to provide a fair comparison. Performance optimization and tuning are highly dependent on the use case and are, therefore, outside the scope of this work. However, we level the field by allocating roughly the same resources for each SPS and scoring alternative.

Resources. For the Kafka cluster, we created 32 partitions per topic and used `LogAppendTime` to measure the *end* timestamp of the inference task. We tested the maximum throughput of a no-op inference task using this configuration to ensure that the Kafka cluster did not hinder our throughput measurements. We also increased the maximum request size to `50 MB` to enable sending large messages for the latency experiments. All of the stream processors run on one compute node. Each streaming framework had equal resources allocated, with the task executors getting a total of 120 GB and 60 cores. We run Apache Flink and Spark Structured Streaming in cluster mode. Flink is configured with one Job Manager and one Task Manager. Spark Structured Streaming uses a similar configuration. Kafka Streams also runs with one instance. The configurations are listed in Table 3.

Scaling up. We set the default parallelism for Flink, Kafka Streams, and Spark Structured Streaming according to the *mp* parameter of each experiment. As for Ray, to simulate a similar set-up, we manually spawn *mp* input actors, *mp* scoring actors, and *mp* output actors and forward the data using a one-to-one mapping from one actor to the next.

Hardware Acceleration. We used the default model-level optimizations and configurations for the serving tools, except for inference CPU parallelization, where we used one thread for inter- and intra-operator parallelism in all embedded and external serving tools that supported this optimization. For some experiments, we enable GPU-based processing.

Network Calls. All external calls were executed as blocking for all SPSs. We decided not to use Apache Flink's asynchronous I/O operations [19] for external serving to ensure a fair comparison with the other stream processors that do not offer native support. As detailed in §3.4, gRPC was used for communication between the SPS and the external model servers in all experiments, except in the case of Ray Serve, where we used HTTP.

5 EXPERIMENTAL RESULTS

We now present the results of our evaluation. We showcase a selection of stream processors and serving tool combinations and discuss the most interesting insights. The throughput measurements are expressed in events per second, where one event can contain a batch, while the latency is measured in milliseconds.

5.1 Comparing Serving Alternatives

To address **RQ1**, we assess the performance of the selected inference tools. We use Apache Flink as the host SPS.

5.1.1 Throughput. The inference results for the *FFNN* and *ResNet50* models are presented in Table 4. The embedded models exhibit significantly superior throughput, with ONNX and SavedModel achieving a throughput of more than 1200 events per second for the *FFNN* model. Among the tested external serving approaches, TF-Serving performs better than TorchServe, maintaining an input rate of 617 req/s, almost three times higher. We

Table 4: Throughput comparison of the serving tools using Apache Flink as stream processor of choice ($bsz = 1$ events/s, $mp = 1$). (e) stands for embedded, while (x) represents external alternatives.

Model	*FFNN*					*ResNet50*		
Model Server	DL4J (e)	ONNX (e)	SavedModel (e)	TorchServe (x)	TF-Serving (x)	ONNX (e)	TorchServe (x)	TF-Serving (x)
Throughput (events/s)	787.53	1373.07	1289.68	225.09	617.2	2.85	0.91	2.62

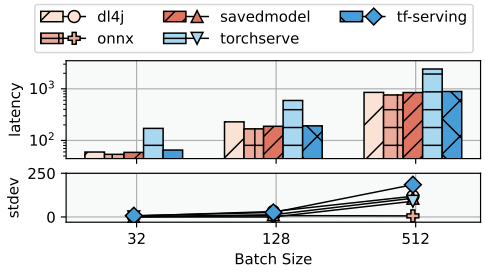

Figure 5: End-to-end request latency average measurements (ms/batch) for Apache Flink and increasing batch sizes using the *FFNN* model ($ir = 1$ event/s, $mp = 1$).

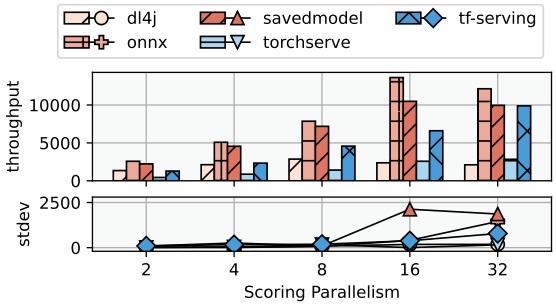

Figure 6: Vertical scalability results measured in requests/sec for the considered model serving tools using Apache Flink as SPS and *FFNN* as the pre-trained model ($ir = 30$k events/s, $bsz = 1$).

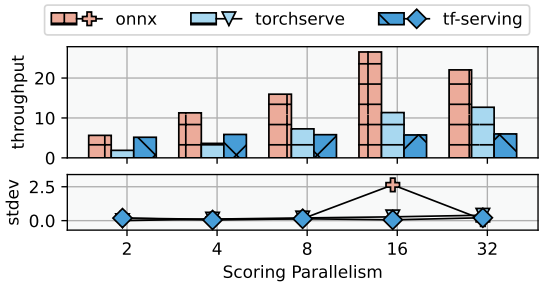

Figure 7: Vertical scalability results of considered model scoring alternatives using Apache Flink as SPS and *ResNet50* as the pre-trained model ($ir = 256$ events/s, $bsz = 1$).

attribute this to the off-the-shelf usage of CPU optimizations, as also noted in the literature [25]. Next, of note, the performance variation across serving tools of the same type is significant. For instance, DL4J's measured throughput is 42.6% lower than SavedModel. Furthermore, the gap in performance between TF-Serving and DL4J is below 200 events per second, hinting that a highly optimized external server can achieve comparable results to embedded options. Contrary to intuition, the embedded

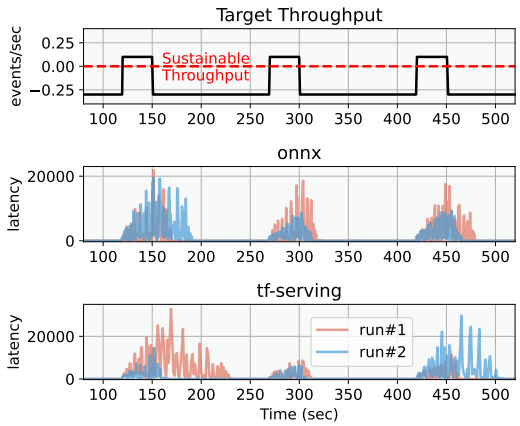

Figure 8: Bursty workload results of considered model scoring alternatives using Apache Flink as SPS and *FFNN* as the pre-trained model ($bsz = 1$, $mp = 1$, $bd = 30$s, $tbb = 120$s). Performance is shown for the first 3 bursts of the run, excluding the warmup.

options outperform the external servers, despite having the inference computation share the same resources as the stream processor. Next, we compare the throughput of a selection of serving alternatives using the *ResNet50* model. In contrast, the results show that less than 3 events per second can be sustained for all the evaluated serving tools. Additionally, we highlight that ONNX sustains an ingestion rate similar to TF-Serving; therefore, the choice between embedded and external in this case is not straightforward when serving large models.

5.1.2 Latency. We now evaluate the end-to-end latency for increasing input sizes in the *closed-loop* scenario for the *FFNN* model. As before, the performance of a single inference task is assessed, and thus, the default parallelism in Flink is set to 1. The latencies of serving batch sizes up to 512 are presented in Figure 5. The embedded options demonstrate similar performance outcomes, with end-to-end serving latencies. Surprisingly, in the case of external serving, the latency achieved by TF-Serving is comparable to the embedded alternatives and, in some cases, even lower. For instance, TF-Serving exhibits an average inference latency of 191 milliseconds for $bsz = 128$, while DL4J and SavedModel show respective latencies of 229 ms and 188 ms. This finding enforces the conclusion that, despite hosting it on a remote machine, which requires extra communication calls, an optimized external server can deliver lower latencies than storing and loading the model in memory close to the stream processor. Lastly, we observe that the standard deviation obtained among the runs is higher for larger batch sizes, but the latency remains stable for batch sizes 32 and 126.

5.1.3 Scaling Up. Figures 6 and 7 present the outcomes of scaling up the inference task for the *FFNN* and *ResNet50* models, respectively. We first analyze the results of the former model. DL4J attains a maximum throughput of 2.8k events per second for parallelism 8. At best, ONNX achieves a maximum throughput of up to 13.6k events/s with $mp = 16$. SavedModel's best throughput is also achieved at parallelism 16, at 10.4k events/s. Next,

Table 5: Throughput comparison for the *FFNN* model and the tested SPSs ($bsz = 1$ req/s, $mp = 1$). (*e*) stands for embedded, while (*x*) represents external alternatives.

Stream Processor	Apache Flink		Kafka Streams		Spark Structured Streaming		Ray	
Model Server	ONNX (*e*)	TF-Serving (*x*)	ONNX (*e*)	TF-Serving (*x*)	ONNX (*e*)	TF-Serving (*x*)	ONNX (*e*)	TF-Serving (*x*)
Throughput (events/s)	1373.07	617.2	2054.21	702.12	4044.99	3924.49	157.4	122.44

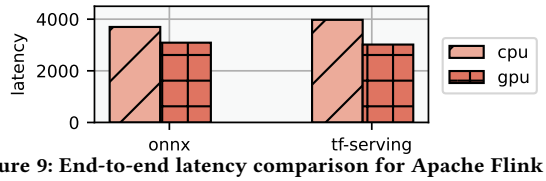

Figure 9: End-to-end latency comparison for Apache Flink and the *ResNet50* model ($ir = 0.2$ events/s, $mp = 1$, $bsz = 8$).

we focus on the external servers. TF-Serving attains a maximum throughput of 9.8k events per second among the external options, while TorchServe's performance peaks at 2.8k events/s. Interestingly, TF-Serving surpasses DL4J in our tests. We also notice that the tools achieve maximum arrival rates at different threading options. Both ONNX and TF-Serving leverage data-parallel execution in Flink and scale up to parallelism 16, while DL4J stops scaling up after parallelism 8. This trend is not followed by external tools, whose performance consistently increases when adding more resources to the inference task, showing that sharing resources between stream processing and model inference can be detrimental to scalability. Lastly, the embedded tools showcase higher standard deviation among different runs, especially for high parallelism levels. Distinctively, SavedModel showed a standard deviation of around 2300 events per second with parallelism 16. We now discuss the scalability capabilities with respect to the inference of the *ResNet50* model. ONNX and TorchServe show the same behaviors as observed for the *FFNN* model. However, in this case, TensorFlow Serving shows negligible performance increases when scaling up. Although TorchServe is outperformed by TF Serving in low-resource scenarios, it surpasses its direct competitor after parallelism 8. We also note that ONNX obtains the maximum standard deviation noted at parallelism 16 but is negligible.

5.1.4 Periodic Bursts. In this section, we investigate the impact of bursty workloads on various serving tools, focusing specifically on ONNX and TensorFlow Serving as representative respective examples. Our analysis targets serving frameworks since the performance of stream processors in similar scenarios has been assessed in previous studies [52]. We test the systems under bursts of 30 seconds that stress the SUT, with 2-minute windows in between having target throughput lower than the sustainable throughput measured in §5.1. Figure 8 illustrates the observed behavior of the systems for the two runs tested. The best recovery time achieved by the systems is 41.37 seconds for ONNX and 34.16 for TensorFlow Serving, showcasing the potential of the latter to recover faster after bursts. However, we note significant variations in performance between consecutive bursts, especially for the external approach, in all our runs. The average recovery time found for ONNX across runs was 46.52 seconds, while TensorFlow Serving recovered on average in 56.15 seconds. Therefore, our experiment shows that TensorFlow Serving can recover faster than ONNX but shows higher variation from one recovery to the next, whereas ONNX demonstrates a comparatively more stable performance trend. A deeper analysis is required to understand the reasons behind this behavior.

Takeaways:
(1) *The performance difference between different serving tools in the same category is significant.*
(2) *ONNX achieves the highest throughput and lowest inference latency,* followed closely by SavedModel.
(3) Despite being an external serving tool, *TensorFlow Serving can be faster than some embedded alternatives.*
(4) *Embedded options face scalability challenges,* as they share resources with the SPS.
(5) *Larger models narrow the performance gap among serving tools.*
(6) *TensorFlow Serving recovers faster than ONNX in bursty workloads* but with higher variation.

5.2 GPU Acceleration

We now present the results of the experiments targeting **RQ2**. We show the results of enabling GPU acceleration for ONNX and TensorFlow Serving and compare them against their CPU-only alternative. We will refer to these as `onnx-cpu`, `onnx-gpu`, `tf-serving-cpu`, and `tf-serving-gpu`. To ensure meaningful comparisons, we focused on testing *ResNet50* model, which is the biggest in our study and accepts larger inputs, to simulate a scenario where the data transfer overhead to the GPU is justified. We use Flink as the SPS of choice and use the same *closed loop* experiment design: we set the input rate to emit one event every 5 seconds, the default parallelism to 1, and the batch size to 8.

Our findings, depicted in Figure 9, reveal an improvement of 16.4% for ONNX in terms of latency per batch, going from 3698ms for `onnx-cpu` to 3089 for `onnx-gpu`. When it comes to the external server, TensorFlow Serving achieved a reduction in inference latency of 24.1%, from an average of 3974ms to 3016ms, hinting that specialized ML serving tools could potentially show more benefits from hardware accelerators. Furthermore, since TensorFlow Serving is an external approach, we note that inference acceleration could potentially amortize the overhead introduced by external network communications. In our experiments, we see that `tf-serving-gpu` shows marginally lower latency than `onnx-gpu`, although we observed a high standard deviation between consecutive runs for the latter, of up to 500ms. In contrast, the other combinations show a maximum deviation of 90ms. `tf-serving-gpu` also outperforms `onnx-cpu`, achieving an improvement of 18.4%.

Takeaways:
(1) *TensorFlow Serving shows great improvements in latency when enabling GPU acceleration.*
(2) *ONNX benefits from GPU accelerators,* albeit to a lesser degree.
(3) *GPU-based external servers can outperform embedded options and alleviate the costs of network exhanges.*

5.3 Stream Processors Comparison

Next, we aim to answer **RQ3** by studying the performance of a selection of SPSs. We set ONNX and TensorFlow Serving as representative embedded and external serving tools, respectively.

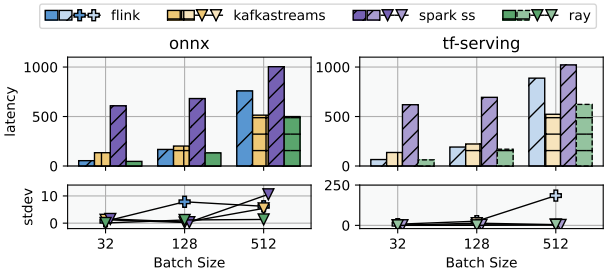

Figure 10: End-to-end latency comparison of the SPSs using the *FFNN* model (ir = 1 event/s, mp = 1)[2].

5.3.1 Throughput. Depicted in Table 5, the results indicate varying performance outcomes across the assessed SPSs. Kafka Streams achieves considerably higher throughput than Flink for both the embedded and external servers, showcasing a boost of 49.6% for ONNX and 13.7% for TF-Serving. This increase can potentially be attributed to better compatibility with the message broker chosen for the data storage source. Next, Spark SS achieved the highest throughput among the tested SPSs, with respective values of around 4000 events per second for both ONNX and TF-Serving. This result is not surprising since Spark SS is designed to achieve high throughput by applying the computation in mini-batches (as detailed in §3.4.1) but at the cost of latency. For instance, we find that using ONNX, the *FFNN* model, and (ir = 512 events/s, bsz = 1, mr = 1), on average, one event is served in 16.25 ms in Kafka Streams, but in 290.78 ms in Spark SS. Interestingly, using Spark SS maximizes the performance of TF-Serving and makes the performance difference between the embedded and external approaches almost imperceptible. Lastly, Ray exhibits the lowest throughput among the tested systems in both serving scenarios but sustains similar arrival rates for embedded and external serving. Further investigation is required to identify the bottlenecks in the Ray pipelines.

5.3.2 Latency. Figure 10 shows the end-to-end latency for increasing batch sizes under low input rates. Among SPS, Flink achieves the lowest latency for batches of size 32 and 128, but is outperformed by Kafka Streams for batch size 512. This behavior is consistent for both the embedded and external inference alternatives. The buffering and waiting times in Flink's push-based model, coupled with a default parallelism of 1, potentially contribute to increased latency to achieve high throughput, particularly for large input records when those do not align with Flink's buffer quota. In contrast, Kafka Streams does not have built-in mechanisms to split large objects, which, under very low input rates, could be beneficial to minimize transfer latency. Spark SS exhibits the highest latency across the board due to overheads introduced by micro-batching. Ray also achieves comparable, or sometimes even lower latency measurements than the SPSs. For instance, using TensorFlow Serving, Ray scores an event of 128 data points in 169.7 ms, while Flink completes the inference task in 167.44 ms, on average. Furthermore, Ray shows a competitive advantage even in the case of external serving, despite using HTTP instead of gRPC for communication. Lastly, we note higher measured standard deviations, especially for bigger batch sizes.

5.3.3 Scaling up. In Figure 11, we plot the vertical scalability experiment results across the selected streaming engines. Spark SS achieves high maximum throughputs of roughly 23k events per second for most configurations, but the performance does not seem to be improved by scaling up the used resources. Further investigation is required to uncover the root cause. However, we note the high maximum throughput of 10.2k events/s achieved by Spark SS with TensorFlow Serving and parallelism 2. When comparing the same configuration but in combination with the other stream processors, we note that they achieve considerably lower performance, with Kafka Streams showcasing throughput 7.2 times lower than the one of Spark SS, hinting that the TensorFlow Serving instance was not saturated and that the performance bottlenecks come from the SPS side. Next, Kafka Streams yields good performance scalability metrics in both embedded and external settings, peaking at 23k events/s using ONNX and scoring parallelism equal to 16 while showing consistent performance increase when increasing the parallelism. Flink's performance shows similar tendencies but achieves a maximum throughput of 13k events/s using ONNX and 9.8 k events/s with TensorFlow Serving. This could be attributed to Kafka Streams' pull-based network model, as detailed in §3.4.1, and its tight integration with the input/output Kafka message queues. Kafka Streams' pull-based approach can potentially distribute work across multiple threads more efficiently by fetching data directly from Kafka partitions when needed. In contrast, Flink's push-based model and buffering strategies can lead to network congestion, which may limit its ability to scale throughput effectively with more threads. Further, Ray peaks at 1.2k events per second for ONNX and only 455.44 events/s for the external counterpart. The latter could be justified by a design choice we found in Ray Serve, where a single HTTP Proxy can be deployed per physical node (as depicted in Figure 4). As its role is to forward requests to inference replicas, it can potentially hinder the prospects of vertical scalability. Lastly, we note the higher standard deviations across runs with higher parallelism, which is consistent with the scalability experiments we showcased in §5.1.

Takeaways:

(1) *Ray shows the lowest end-to-end inference times, but also the lowest maximum sustained throughput.*
(2) *Flink shows low inference times for smaller batches, but loses its competitive advantage in favor of Kafka Streams for bigger batches.*
(3) *Spark Structured Streaming's micro-batching mechanism helps saturate the external servers and achieve the best optimal throughput,* making it more suitable in combination with external approaches.
(4) *All the tested SPSs's performance increases when scaling vertically, except for Spark Structured Streaming.*

6 DATA MANAGEMENT CONSIDERATIONS

Integrating ML inference into stream processing pipelines is intrinsically a data management problem. Based on our experience conducting the experimental analysis, we noticed that the impact of data transfer overheads and resource allocation are key problems that influence the end-to-end performance of ML inference in this context. In this section, we discuss these challenges and further explore the overhead introduced by `Crayfish` to understand its implications on the overall system performance.

6.1 SPS Resource Allocation

In data stream processing pipelines that use ML inference, optimizing solely the inference task may not lead to the desired outcomes. While allocating adequate resources for inference is crucial, it is essential to recognize that other components within

[2]Ray is depicted with dotted lines for the `tf-serving` case because it is not using TensorFlow Serving, but simulating it using Ray Serve.

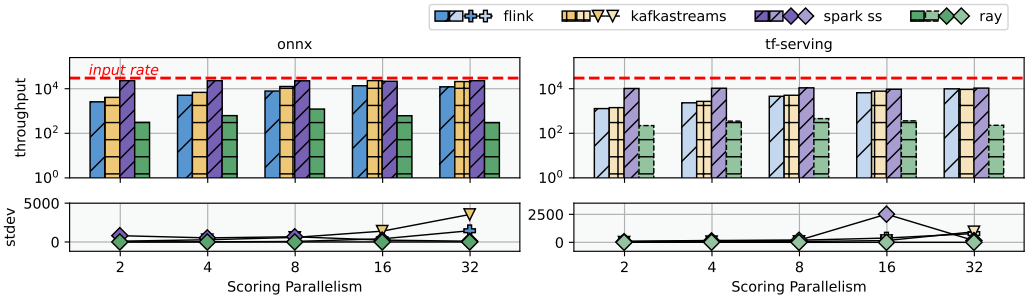

Figure 11: Vertical scalability results of the selected SPSs and serving tools using the FFNN model (ir = 30k events/s, bsz = 1) [2].

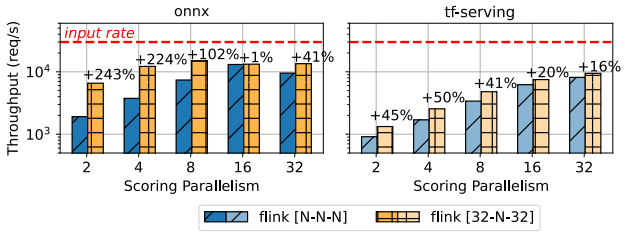

Figure 12: Parallelism performance for Apache Flink and the FFNN model. `[A-B-C]` defines a Flink DAG with parallelism set to A on the input operator, B on the scoring operator, and C on the out operator. In `flink[32-N-32]` we disable operator chaining.

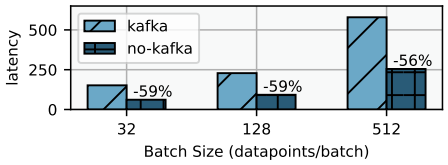

Figure 13: Overhead introduced by `Crayfish` (`kafka`) in terms of end-to-end latency when compared to a self-contained, equivalent standalone Apache Flink implementation (`no-kafka`). The experiments used the ONNX serving tool (ir = 1 events/s, mp = 1).

the pipeline, such as upstream or downstream operators like sources or sinks, can substantially impact the overall performance. These components may introduce complexities and bottlenecks, potentially outweighing the computational overhead of the scoring task. Therefore, to achieve optimal performance in the inference task, it becomes imperative to scale these upstream or downstream operators independently, ensuring that the entire pipeline is efficiently managed and the performance of the inference task is maximized without suffering from overhead that is not essential to model serving.

To validate the argument above, we conducted an experiment using Apache Flink with two distinct pipelines to explore the decoupling of the parallelism level of the reading and writing operations from that of the scoring task and devise two distinct Flink pipelines. The first alternative, `flink[N-N-N]`, employs the default parallelism configuration of N data-parallel task, as used in prior experiments. The second scenario, denoted as `flink[32-N-32]`, uses operator parallelism and matches the parallelism of the source and sink to the number of partitions in the Kafka topic (i.e., 32), therefore allocating more resources to reading and writing. The scoring operator is the only component in the pipeline that is scaled up. We now discuss the results of serving the *FFNN* model using ONNX, but also show the results of inference using TF-Serving, which exhibit similar trends. Firstly, when it comes to the maximum throughput of one scoring task, `flink[N-N-N]` achieves 1393.07 events per second, as reported in §5.1.1. However, we find that `flink[32-N-32]` achieves an input rate around 3.8 times higher, reaching up to 5373.15 req/s. Furthermore, Figure 12 illustrates the results of an experiment testing the vertical scalability of these alternative parallelism configurations. The findings indicate that `flink[N-N-N]` achieves consistently lower performance due to resource constraints in the reading and writing operations, suggesting that these may serve as bottlenecks in a chained pipeline. Accordingly, operator-level parallelism proves advantageous for model scoring, as it maximizes the utilization of the scoring tool. This outcome strongly supports the intuition that independently scaling operators within the data pipeline yields better results, emphasizing the significance of benchmarking and optimizing end-to-end pipelines rather than solely focusing on the inference task.

6.2 Measuring the Kafka Overhead

We now quantify the overhead introduced in `Crayfish` by having Kafka handle the input and output feeds. We devised a standalone Flink pipeline that infers using the *FFNN* model in ONNX format. This pipeline is also responsible for input data generation and recording the output timestamps. We use operator-level parallelism and set the same configurations for both the standalone (`kafka`) and the `Crayfish` adapter (`no-kafka`). In throughput measurements, `Crayfish` achieved up to 5506.67 events/s, whereas the standalone implementation exhibited an ST of 5506.67 events/s. Therefore, `Crayfish` incurred a throughput overhead as low as 2.42% by introducing Kafka in the pipeline. At the same time, `Crayfish` also displayed extra latency, as it can be noted in Figure 13. This is expected since multiple communication hops are needed between multiple systems. To that end, we observed up to 59% lower latency in the standalone configuration compared to the one with Kafka-based logging. However, we argue that introducing realistic latency overheads is beneficial in measuring the performance of the inference task over streaming data since, in production-level systems, it is rarely the case that an SPS will operate in isolation.

7 DISCUSSION

In this section, we first discuss the features worth adopting when designing new SPSs equipped with inference capabilities based on insights obtained in our empirical evaluation. Next, we highlight the open research questions in this area.

7.1 Designing New Systems

External Inference Tools for Decoupled Scalability. If scalability is a hard requirement, external serving systems offer a more

suitable solution, according to our study. We showed that sharing resources between an SPS and an embedded serving tool can curb the system's performance when scaling up. The advantage of external systems lies in their decoupled scalability, allowing for independent adjustments without even requiring redeployment of the SPS. Furthermore, this separation facilitates a more nuanced understanding of system performance with respect to the resources allocated to the serving tool. At the opposite pole, the scalability gains are difficult to grasp in systems combining SPSs and embedded serving tools, and allocating resources to scale up the serving task is not trivial.

Micro-batching Support for External Servers. We found in §5.3.3 that the event-based SPSs, Kafka Streams and Apache Flink, showed significantly lower throughput than micro-batch-based systems (i.e., Spark Structured Streaming) when utilized in conjunction with external inference tools, in situations where the resources allocated to the stream processor are low. This suggests that batching the inference requests is crucial for maximizing the performance of the external serving tool.

Operator-level Parallelism. In Section 6.1, we meticulously analyze the potential bottlenecks arising from upstream and/or downstream tasks, emphasizing the imperative of operator-level scaling to optimize the performance of the inference task.

7.2 Open Research Questions

Despite the shown competitive performance of the embedded options in specific cases, we believe that external serving is currently the more attractive alternative. First, embedded serving requires custom effort and rare interdisciplinary knowledge of ML and event-based system internals. Furthermore, the execution model of current dataflow systems is rather restrictive in support of capabilities essential for model serving in production. That includes the lack of model management, auto-scaling, state sharing, multi-model serving, and native hardware acceleration capabilities for inference, features natively supported by most external alternatives. These considerations are crucial for many industries that require flexibility to deploy and serve thousands of models of different sizes concurrently, each with different deployment time, re-deployment periodicity, and lifespan [26].

Nevertheless, despite these advantages, modern streaming services prioritize tight integration and processing guarantees, like fault tolerance and exactly-once processing, which are not ensured with external interfacing. Hence, looking forward, we identify the prospects of a close integration supported between external serving tools and stream processors. Core challenges in achieving this include decentralized dataflow execution [46], hardware resource management and storage for hybrid relational and vectorized operations, as well as enriching existing event-based programming libraries with ML semantics.

8 RELATED WORK

Except for a short study that quantitatively compares serving alternatives using Apache Flink [25], to the best of our knowledge, this paper is the first work that offers both a benchmarking framework and an extensive performance analysis of selected alternatives for stream processing, serving tools, and representative pre-trained models. In this section, we outline related efforts.

Benchmarking Streaming Frameworks. As the demand for real-time data processing has grown, so has the need for thorough evaluations of SPSs. Providing a taxonomy for such studies is outside the scope of this work; we refer the reader to recent results that provide general-purpose performance evaluations [10, 23, 29, 52]. Similarly to our study, these evaluations focus on measuring throughput and latency. When it comes to benchmarking tools, we mention efforts such as ESPBench [24] or Yahoo! Streaming Benchmark (YSB) [10], which are configurable and offer APIs to plug in new workloads, or evaluation metrics.

Machine Learning Benchmarks. Most efforts towards benchmarking ML solutions focus on optimizing models' performance on respective hardware. Distinctively, we highlight the MLPerf Inference Benchmark, a community effort that proposes a standardized benchmark suite for measuring the performance of ML inference systems [42]. The benchmark suite includes a range of workloads and models across several domains, including image classification, object detection, and natural language processing. While streaming data can be used as input for the model serving task, MLPerf is not specifically designed to evaluate the performance of SPSs. Moreover, MLPerf does not consider external alternatives such as TensorFlow Serving.

Unifying Directions. Finally, we acknowledge the attempts made toward consolidating the Python ecosystem with modern stream processors. Presently, numerous SPSs provide Python APIs that may alleviate the necessity for interoperation libraries for embedded inference. For instance, Apache Flink offers a specialized ML solution through Flink ML [18], comprising a number of Flink-based operators for standard ML algorithms. Notably, at the moment of writing this paper, Flink ML does not offer deep learning capabilities. Subsequently, InferLine [12] and Clipper [13] are versatile systems that support inference with plugged-in models and advanced capabilities, including online learning or adaptive batching. They feature RPC-type interfaces analogous to the external alternatives considered in this research.

9 CONCLUSION AND FUTURE WORK

In this work, we presented `Crayfish`, an extensible framework to benchmark model serving alternatives over streaming data. `Crayfish` can aid developers in choosing the option most suitable for their use case and quickly iterating and optimizing the inference task in their chosen stream processor. Among possible prospects for future work, we highlight the non-trivial problem of independent scaling and resource management for model serving in conjunction with stream processing with the goal of analyzing optimal non-uniform allocations of resources between the stream processor and the serving tool. Lastly, `Crayfish` could be extended to add support for models with unique requirements, such as Graph Neural Networks, which, besides access to a stored pre-trained model, require reading historical data in the form of a k-hop neighborhood of a node.

ACKNOWLEDGMENTS

The authors would like to thank Jun-Wei Liu, Sophie Zhang, and Achilleas Stefanidis for their contributions. This work was supported by the Wallenberg Foundation (WASP), the Google Cloud Research Credits program, and the National Academic Infrastructure for Supercomputing in Sweden (NAISS), partially funded by the Swedish Research Council through grant agreement no. 2022-06725. E. Kritharakis is supported by the Onassis Scholarship [Scholarship ID: F ZR 030/1-2021/2022]. V. Kalavri's work is supported by the National Science Foundation under Grant No. 2237193, a Red Hat Collaboratory Research Incubation Award (ID:2023-01-RH03), and a Google DAPA Award.

REFERENCES

[1] Leonel Aguilar, David Dao, Shaoduo Gan, Nezihe Merve Gurel, Nora Hollenstein, Jiawei Jiang, Bojan Karlas, Thomas Lemmin, Tian Li, Yang Li, et al. 2021. Ease. ML: A lifecycle management system for MLDev and MLOps. *Proc. of Innovative Data Systems Research* (2021).

[2] Adnan Akbar, Abdullah Khan, Francois Carrez, and Klaus Moessner. 2017. Predictive analytics for complex IoT data streams. *IEEE Internet of Things Journal* 4, 5 (2017), 1571–1582.

[3] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. 2015. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. (2015).

[4] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 601–613. https://doi.org/10.1145/3183713.3190664

[5] Colby R. Banbury, Chuteng Zhou, Igor Fedorov, Ramon Matas Navarro, Urmish Thakker, Dibakar Gope, Vijay Janapa Reddi, Matthew Mattina, and Paul N. Whatmough. 2021. MicroNets: Neural Network Architectures for Deploying TinyML Applications on Commodity Microcontrollers. In *Proceedings of Machine Learning and Systems 2021, MLSys 2021, virtual, April 5-9, 2021*, Alex Smola, Alex Dimakis, and Ion Stoica (Eds.). mlsys.org. https://proceedings.mlsys.org/paper/2021/hash/a3c65c2974270fd093ee8a9bf8ae7d0b-Abstract.html

[6] Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, and Kenneth Knowles. 2019. One SQL to rule them all-an efficient and syntactically idiomatic approach to management of streams and tables. In *Proceedings of the 2019 International Conference on Management of Data*. 1757–1772.

[7] Riccardo Cappuzzo, Paolo Papotti, and Saravanan Thirumuruganathan. 2020. Creating Embeddings of Heterogeneous Relational Datasets for Data Integration Tasks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1335–1349. https://doi.org/10.1145/3318464.3389742

[8] Paris Carbone, Marios Fragkoulis, Vasiliki Kalavri, and Asterios Katsifodimos. 2020. Beyond analytics: The evolution of stream processing systems. In *Proceedings of the 2020 ACM SIGMOD international conference on Management of data*. 2651–2658.

[9] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38.

[10] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Peng, and Paul Poulosky. 2016. Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2016, Chicago, IL, USA, May 23-27, 2016*. IEEE Computer Society, 1789–1792. https://doi.org/10.1109/IPDPSW.2016.138

[11] PyTorch Serve Contributors. 2023. TorchServe. https://pytorch.org/serve/. [accessed 11-July-2023].

[12] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. 2020. InferLine: latency-aware provisioning and scaling for prediction serving pipelines. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi (Eds.). ACM, 477–491. https://doi.org/10.1145/3419111.3421285

[13] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, Aditya Akella and Jon Howell (Eds.). USENIX Association, 613–627. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw

[14] Miyuru Dayarathna and Srinath Perera. 2018. Recent advancements in event processing. *ACM Computing Surveys (CSUR)* 51, 2 (2018), 1–36.

[15] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009), 20-25 June 2009, Miami, Florida, USA*. IEEE Computer Society, 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[16] ONNX Runtime developers. 2023. ONNX Runtime. https://onnxruntime.ai/. [accessed 10-July-2023].

[17] Javier Duarte, Nhan Tran, Ben Hawks, Christian Herwig, Jules Muhizi, Shvetank Prakash, and Vijay Janapa Reddi. 2022. FastML Science Benchmarks: Accelerating Real-Time Scientific Edge Machine Learning. arXiv:cs.LG/2207.07958

[18] Apache Flink. 2021. Flink ML Documentation. https://nightlies.apache.org/flink/flink-ml-docs-stable/. [accessed 10-July-2023].

[19] Apache Flink. 2023. Asynchronous I/O in Apache Flink. https://nightlies.apache.org/flink/flink-docs-master/docs/dev/datastream/operators/asyncio/. [accessed 10-July-2023].

[20] Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. 2020. A survey on the evolution of stream processing systems. *arXiv preprint arXiv:2008.00842* (2020).

[21] Fangcheng Fu, Huanran Xue, Yong Cheng, Yangyu Tao, and Bin Cui. 2022. Blindfl: Vertical federated machine learning without peeking into your data. In *Proceedings of the 2022 International Conference on Management of Data*. 1316–1330.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[23] Sören Henning and Wilhelm Hasselbring. 2023. Benchmarking scalability of stream processing frameworks deployed as event-driven microservices in the cloud. *CoRR* abs/2303.11088 (2023).

[24] Guenter Hesse, Christoph Matthies, Michael Perscheid, Matthias Uflacker, and Hasso Plattner. 2021. ESPBench: The Enterprise Stream Processing Benchmark. In *ICPE '21: ACM/SPEC International Conference on Performance Engineering, Virtual Event, France, April 19-21, 2021*, Johann Bourcier, Zhen Ming (Jack) Jiang, Cor-Paul Bezemer, Vittorio Cortellessa, Daniele Di Pompeo, and Ana Lucia Varbanescu (Eds.). ACM, 201–212. https://doi.org/10.1145/3427921.3450242

[25] Sonia Horchidan, Emmanouil Kritharakis, Vasiliki Kalavri, and Paris Carbone. 2022. Evaluating Model Serving Strategies over Streaming Data. In *Proceedings of the Sixth Workshop on Data Management for End-To-End Machine Learning (DEEM '22)*. Association for Computing Machinery, New York, NY, USA, Article 4, 5 pages. https://doi.org/10.1145/3533028.3533308

[26] Chip Huyen. 2022. *Designing Machine Learning Systems.* " O'Reilly Media, Inc.".

[27] Nick Hynes, David Dao, David Yan, Raymond Cheng, and Dawn Song. 2018. A demonstration of sterling: a privacy-preserving data marketplace. *Proceedings of the VLDB Endowment* 11, 12 (2018), 2086–2089.

[28] Colin Jermain. [n.d.]. *Machine Learning Inference in Flink with ONNX*. Vervetica. https://www.youtube.com/watch?v=6g167GXFb4A&ab_channel=Ververica

[29] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking Distributed Stream Data Processing Systems. In *ICDE*. IEEE Computer Society, 1507–1518.

[30] Konduit K.K. 2023. Deeplearning4j. https://deeplearning4j.konduit.ai/. [accessed 10-July-2023].

[31] Sanjay Krishnan, Michael J Franklin, Ken Goldberg, Jiannan Wang, and Eugene Wu. 2016. Activeclean: An interactive data cleaning framework for modern machine learning. In *Proceedings of the 2016 International Conference on Management of Data*. 2117–2120.

[32] Arun Kumar, Matthias Boehm, and Jun Yang. 2017. Data management in machine learning: Challenges, techniques, and systems. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1717–1722.

[33] Valliappa Lakshmanan, Sara Robinson, and Michael Munn. 2020. *Machine learning design patterns*. O'Reilly Media.

[34] Side Li, Lingjiao Chen, and Arun Kumar. 2019. Enabling and Optimizing Non-Linear Feature Interactions in Factorized Linear Algebra. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1571–1588. https://doi.org/10.1145/3299869.3319878

[35] Xupeng Li, Bin Cui, Yiru Chen, Wentao Wu, and Ce Zhang. 2017. Mlog: Towards declarative in-database machine learning. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1933–1936.

[36] Yiming Li, Yanyan Shen, and Lei Chen. 2022. Camel: Managing Data for Efficient Stream Learning. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1271–1285. https://doi.org/10.1145/3514221.3517836

[37] Yaliang Li, Zhen Wang, Yuexiang Xie, Bolin Ding, Kai Zeng, and Ce Zhang. 2021. AutoML: From Methodology to Application. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management (CIKM '21)*. Association for Computing Machinery, New York, NY, USA, 4853–4856. https://doi.org/10.1145/3459637.3483279

[38] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 561–577. https://www.usenix.org/conference/osdi18/presentation/nishihara

[39] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. TensorFlow-Serving: Flexible, High-Performance ML Serving. *CoRR* abs/1712.06139 (2017). arXiv:1712.06139 http://arxiv.org/abs/1712.06139

[40] Jose Picado, John Davis, Arash Termehchy, and Ga Young Lee. 2020. Learning Over Dirty Data Without Cleaning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1301–1316. https://doi.org/10.1145/3318464.3389708

[41] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. 2017. Data management challenges in production machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1723–1726.

[42] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. 2020. MLPerf Inference Benchmark. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*. IEEE, 446–459. https://doi.org/10.1109/ISCA45697.2020.00045

[43] Matthias J. Sax, Guozhang Wang, Matthias Weidlich, and Johann-Christoph Freytag. 2018. Streams and Tables: Two Sides of the Same Coin. In *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics, BIRTE 2018, Rio de Janeiro, Brazil, August 27, 2018*, Malú Castellanos, Panos K. Chrysanthis, Badrish Chandramouli, and Shimin Chen (Eds.). ACM, 1:1–1:10. https://doi.org/10.1145/3242153.3242155

[44] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Biessmann, and Andreas Grafberger. 2018. Automating large-scale data quality verification. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1781–1794.

[45] Apache Spark. [n.d.]. *Structured Streaming Programming Guide*. Retrieved July 10, 2023 from https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#structured-streaming-programming-guide

[46] Jonas Spenger, Paris Carbone, and Philipp Haller. 2022. Portals: An Extension of Dataflow Streaming for Stateful Serverless. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2022, Auckland, New Zealand, December 8-10, 2022*, Christophe Scholliers and Jeremy Singer (Eds.). ACM, 153–171. https://doi.org/10.1145/3563835.3567664

[47] Ki Hyun Tae, Yuji Roh, Young Hun Oh, Hyunsu Kim, and Steven Euijong Whang. 2019. Data cleaning for accurate, fair, and robust models: A big data-AI integration approach. In *Proceedings of the 3rd International Workshop on Data Management for End-to-End Machine Learning*. 1–4.

[48] Ray Team. 2023. Experimental Direct Ingress. https://docs.ray.io/en/latest/serve/direct-ingress.html. [accessed 12-July-2023].

[49] Ray Team. 2023. Ray Serve: Scalable and Programmable Serving. https://docs.ray.io/en/latest/serve/index.html. [accessed 11-July-2023].

[50] TensorFlow. 2023. Using the SavedModel format. https://www.tensorflow.org/guide/saved_model. [accessed 10-July-2023].

[51] Quoc-Cuong To, Juan Soto, and Volker Markl. 2018. A survey of state management in big data processing systems. *The VLDB Journal* 27, 6 (2018), 847–872.

[52] Giselle van Dongen and Dirk Van den Poel. 2020. Evaluation of Stream Processing Frameworks. *IEEE Trans. Parallel Distributed Syst.* 31, 8 (2020), 1845–1858.

[53] Maria Xekalaki, Juan Fumero, Athanasios Stratikopoulos, Katerina Doka, Christos Katsakioris, Constantinos Bitsakos, Nectarios Koziris, and Christos Kotselidis. 2022. Enabling Transparent Acceleration of Big Data Frameworks Using Heterogeneous Hardware. *Proceedings of the VLDB Endowment* 15, 13 (2022), 3869–3882.

[54] Han Xiao, Kashif Rasul, and Roland Vollgraf. 2017. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. *CoRR* abs/1708.07747 (2017). arXiv:1708.07747 http://arxiv.org/abs/1708.07747

[55] Lijie Xu, Shuang Qiu, Binhang Yuan, Jiawei Jiang, Cedric Renggli, Shaoduo Gan, Kaan Kara, Guoliang Li, Ji Liu, Wentao Wu, Jieping Ye, and Ce Zhang. 2022. In-Database Machine Learning with CorgiPile: Stochastic Gradient Descent without Full Data Shuffle *(SIGMOD '22)*. New York, NY, USA, 1286–1300. https://doi.org/10.1145/3514221.3526150