SensiX++: Bringing MLOps and Multi-tenant Model Serving to Sensory Edge Devices

CHULHONG MIN, AKHIL MATHUR, UTKU GÜNAY ACER, ALESSANDRO MONTANARI, and FAHIM KAWSAR, Nokia Bell Labs, UK

We present SensiX++, a multi-tenant runtime for adaptive model execution with integrated MLOps on edge devices, e.g., a camera, a microphone, or IoT sensors. SensiX++ operates on two fundamental principles: highly modular componentisation to externalise data operations with clear abstractions and document-centric manifestation for system-wide orchestration. First, a data coordinator manages the lifecycle of sensors and serves models with correct data through automated transformations. Next, a resource-aware model server executes multiple models in isolation through model abstraction, pipeline automation, and feature sharing. An adaptive scheduler then orchestrates the best-effort executions of multiple models across heterogeneous accelerators, balancing latency and throughput. Finally, microservices with REST APIs serve synthesised model predictions, system statistics, and continuous deployment. Collectively, these components enable SensiX++ to serve multiple models efficiently with fine-grained control on edge devices while minimising data operation redundancy, managing data and device heterogeneity, and reducing resource contention. We benchmark SensiX++ with 10 different vision and acoustics models across various multi-tenant configurations on different edge accelerators (Jetson AGX and Coral TPU) designed for sensory devices. We report on the overall throughput and quantified benefits of various automation components of SensiX++ and demonstrate its efficacy in significantly reducing operational complexity and lowering the effort to deploy, upgrade, reconfigure, and serve embedded models on edge devices.

CCS Concepts: • **Computer systems organization** → **Embedded software**; *Redundancy*; Special purpose systems;

Additional Key Words and Phrases: MLOps, multi-tenancy, model serving, edge

ACM Reference format:
Chulhong Min, Akhil Mathur, Utku Günay Acer, Alessandro Montanari, and Fahim Kawsar. 2023. SensiX++: Bringing MLOps and Multi-tenant Model Serving to Sensory Edge Devices. *ACM Trans. Embedd. Comput. Syst.* 22, 6, Article 98 (November 2023), 27 pages.
https://doi.org/10.1145/3617507

1 INTRODUCTION

The recent emergence of edge accelerators has radically transformed the analytical capabilities of low-power and low-cost sensory edge devices such as cameras, microphones, or IoT devices. These edge devices can now perform cloud-scale, processing-intensive **machine learning (ML)**

Authors' address: C. Min, A. Mathur, A. Montanari, and F. Kawsar, Nokia Bell Labs, 21 J J Thomson Avenue, Cambridge, CB3 0FA, UK; e-mails: chulhong.min@nokia-bell-labs.com, akhil.mathur@nokia-bell-labs.com, alessandro.montanari@nokia-bell-labs.com, fahim.kawsar@nokia-bell-labs.com; U. G. Acer, Nokia Bell Labs, Copernicuslaan 50, 2018 Antwerpen, Belgium; e-mail: utku_gunay.acer@nokia-bell-labs.com.

1539-9087/2023/11-ART98 $15.00
https://doi.org/10.1145/3617507

inferences locally and eliminate the need to send a large amount of privacy-sensitive data to remote data centres, thereby offering benefits in efficiency, speed, availability, and privacy [49, 59]. This possibility has uncovered a new era of affordable **artificial intelligence (AI)** applications, from personal assistants to recommendation systems to video surveillance available through various edge devices [10, 23, 32, 35, 37, 48, 53, 63].

Today, most ML capabilities embedded in edge devices are limited to inference tasks, i.e., the point at which all the data-driven learning accumulated during training in a data centre is deployed on real-world data to infer a prediction. This task entails (i) preparing the data acquired from a real-world sensor (e.g., pixels from an image sensor or waveforms from a microphone) to a compatible input format, (ii) executing the model within a model-specific framework (e.g., TensorFlow or PyTorch), and (iii) serving the inferences through well-defined interfaces (e.g., REST APIs). Some applications might demand these devices to store inferences to support historical queries and offer provisioning schemes for deployment. Collectively, these operations represent a subset of **Machine Learning Operations (MLOps)** on edge devices.

We have seen significant efforts from academia and industry to offer toolkits to facilitate and accelerate the execution of ML models on edge devices [21, 22, 25, 34, 38, 41, 44, 45, 50, 56]. However, they lack this end-to-end view. For instance, these solutions require expensive and manual intervention to prepare and transform input data. Also, if done inadequately, the model's performance suffers in the new operating environment due to data and device heterogeneity [17, 40, 42–44]. Furthermore, multi-tenancy support is only available in cloud-scale model-serving systems (e.g., [20, 46]). Thus, sensory edge devices are often constrained to provide a specific, pre-defined ML task, and the chance for utilising those devices as general-purpose ML compute devices is accordingly limited. By overcoming these challenges, we envision unleashing a new paradigm of *software-defined sensors*. For instance, a software-defined camera can perform multiple and differential inference tasks assisted by automated MLOps to serve various applications simultaneously or on-demand with zero reliance on distant clouds. Such abilities will transform today's dumb and hard-coded sensory edge devices into dynamic, re-configurable, and intelligent computing platforms.

To this end, we present SensiX++, a multi-tenant model-serving system with integrated MLOps for software-defined sensory edge devices. SensiX++ reduces operational complexity, minimises redundant data operations,[1] eliminates manual intervention, and achieves two crucial properties: high inference throughput and low inference latency. SensiX++ follows a modular system design and applies declarative abstraction principles across its various components, enabling different models and respective pipelines to be managed and configured automatically on different accelerators (Section 4.2). In addition, SensiX++ externalises system-wide data operations away from model execution, thereby enabling multiple models to share identical data transformation and featurisation pipelines.

SensiX++ consists of four main components. First, *a data coordinator* manages the lifecycle of sensors and serves models with correct input data through automated transformations, i.e., resolution scaling, sample scaling, or dynamic encoding (Section 5.1). Second, a *resource-aware model server* executes multiple models in isolation through model abstraction and pipeline automation (Section 5.3). This component dynamically constructs model-specific pipeline containers with appropriate frameworks (TensorFlow, TensorFlow Lite, or PyTorch), considering available processing resources while meeting latency and throughput requirements. This component also leverages a novel feature-sharing functionality that enables multiple models to share a common

[1]Different ML models often use the same data operations before feeding the data to task-specific classifiers, e.g., Mel-filterbank generation for audio signals and frequency-domain feature generation for inertial sensor data.

featurisation pipeline, when possible. The third component of SensiX++ is *a low-overhead scheduler* that orchestrates the best-effort executions of multi-tenant models across heterogeneous accelerators, balancing latency and throughput by applying multiple optimisation heuristics (Section 5.2). Finally, the fourth component of SensiX++ encompasses a set of microservices with REST APIs to serve synthesised inferences and continuous deployment (Section 5.4).

We implemented SensiX++ in Python and NodeJS, deployed it on Jetson AGX and Coral TPU as representative edge accelerators, and added support for widely used embedded ML frameworks, such as TensorFlow, TensorRT, PyTorch, and so forth. Given the dramatic cost reduction of these boards, in conjunction with increasingly cheap compute storage, we expect them to soon feature in commodity scale edge devices, such as a smart camera or a smart speaker. We evaluate SensiX++ with 10 different vision and acoustics models across various multi-tenant configurations and demonstrate that SensiX++ offers balanced throughput and latency in a best-effort manner. We also quantify various components of SensiX++ to prove its efficacy in reducing operational complexities in serving multiple models. In summary, our contributions to this article include the following:

— We present SensiX++, the first-of-its-kind framework that serves multi-tenant models with automated MLOps on sensory edge devices.
— We devise a novel declarative abstraction mechanism for system-wide orchestration, automated data operation, and pipeline construction for heterogeneous models.
— We identify the technical challenges to serve multi-tenant models on edge devices. Then, we devise a set of novel techniques achieved through adaptive scheduling, feature caching, and shared data operation to reduce and bound latency while maximising throughput, and we prototype the end-to-end, integrated system.
— We demonstrate the efficiency and efficacy of the system through extensive experiments.

2 RELATED WORK

SensiX++ is designed for serving multiple models with automated MLOps on sensory edge devices. In this section, we review related research in these two areas.

2.1 Serving Models on Edge Devices

Unlike model training that happens offline, inference usually serves applications directly and needs to be resource efficient. Naturally, this requirement has drawn significant attention and many techniques have been proposed including architecture scaling [13, 22, 34, 47, 50], model compression and quantisation [16, 24, 25, 38], pruning [36, 39, 64], accelerator-aware compilation [12, 51, 56], model partitioning [31, 33, 62], logic-based inference algorithms [14, 15], and system support for IoT devices [11, 28, 55].

While models that run on SensiX++ can benefit from these techniques to meet latency or accuracy targets, the core model-serving capabilities of SensiX++ are independent of this aspect. In addition, model efficiency alone is insufficient to meet system-wide performance in a multi-tenant setting. SensiX++ is designed to make systematic decisions concerning how to run various models with different runtime and performance constraints on edge devices to offer predictable throughput and latency. A few research studies that come close to these objectives are NestDNN [21], HiveMind [52], DeepEye [41], and BAND [30]. In NestDNN [21], Fang and his colleagues transform multiple mobile vision models into a single, multi-capacity model consisting of a set of descent models with graceful performance degradation to serve continuous mobile vision applications. In a related effort, although cloud-scale, Narayanan et al. proposed HiveMind [52] that compiles multiple models into a single optimised model by performing cross-model operator fusion and sharing

I/O across models to optimise GPU parallelisation. In [41], Mathur and his colleagues present an execution framework that carefully segregates and schedules different computational layers of multiple models. These research efforts offer an excellent foundation for our work. However, they rely on accessing and modifying model architecture and weights. Jeong et al. proposed BAND [30] that coordinates multi-DNN inferences on heterogeneous mobile processors by partitioning the input model into a set of subgraphs and scheduling subgraphs from multiple models in a holistic way. Instead, SensiX++ takes a more practical and model-agnostic execution approach. SensiX++ treats each model as a black box operating on top of its native runtime framework in a container. These resource-aware containers are then dynamically scheduled with varying heuristics considering system status and individual performance targets.

There has been considerable prior work in multi-tenant model serving at a cloud scale. For instance, Velox [19] and Clipper [20] developed at UC Berkeley utilise a decoupled and layered design with useful abstractions to interpose models on top of different runtime frameworks to build low-latency serving systems. TensorFlow Serving [46] is from Google for serving TensorFlow models, and SageMaker [8] from Amazon is a more general-purpose platform to prepare, build, train, and deploy ML models. These vertically integrated frameworks essentially offer a containerised environment with micro-services to execute respective models. SensiX++ is inspired by these systems, their model abstractions, and scheduling optimisations. However, in contrast to these systems, SensiX++ supports a wide range of ML models and frameworks, resource-aware scheduling, automated data transformation and pipeline, all at edge scale.

2.2 Automated MLOps on Edge Devices

MLOps are a relatively new area of data engineering, and so far, most of the attention is focused on automated model training and primarily driven by the industry due to the challenges in running ML systems in the real world [54, 60]. For instance, KubeFlow [5] operates on top of Kubernetes and aims to simplify the deployment and orchestration of ML pipelines and workflows. Google's TensorFlow Extended [46] has similar objectives and integrates all components of an ML pipeline to reduce time to production. Uber's Michelangelo [7] is designed to build and deploy ML services in an internal ML-as-a-service platform. There are several other initiatives such as BentoML [1] and ElectrifAI [3] that automate and orchestrate ML workflows to serve in a production environment. All of these platforms essentially aim to reduce the transition time from development to production for ML systems, and in many ways, automate different workflows to simplify and scale these systems. However, these systems operate in a cloud environment, and we are yet to see their entrance to edge devices. SensiX++ is a first-of-its-kind system for bringing these MLOps capabilities to edge devices. SensiX++ borrows many concepts from these systems; for instance, automated data transformation, dynamic pipeline, or declarative abstraction to automate system-wide orchestration while contextualising them with careful assessment of resource constraints and performance targets.

3 SENSIX++: DESIGN PRINCIPLES

The primary objective of SensiX++ is to bring cloud-scale MLOps and multi-tenant model serving to sensory edge devices. To this end, we have designed SensiX++ with the following four design principles.

- **Declarative abstraction for system orchestration:** The diversity of ML models, underlying frameworks, and input variabilities, poses a significant challenge in building a multi-tenant serving solution. We argue that the first step to addressing this challenge is to devise a declarative abstraction that defines the model and explains its runtime requirement and

I/O dynamics. In SensiX++, we apply such declarative abstraction in a manifest script that acts as the glue to bring various system components together with systematic orchestration.

— **Resource-aware model isolation:** Every ML model is unique with implicit assumptions on data distributions and underlying library support. It is essential to keep these dependencies intact while running this model in the real world to maintain accuracy and robustness. However, this maintenance becomes highly complicated in a multi-tenant setting unless we use appropriate abstractions and isolation. In SensiX++, we apply model abstraction and process isolation by running each model in a separate container that embodies all runtime dependencies. We create these containers to run on different available processors to avoid resource contention through careful and periodic assessment of the system load and demands.

— **Externalisation of data operations:** Most model-serving systems today assume that model-specific input is available. This assumption demands significant manual operations in a multi-tenant production environment and is particularly cumbersome for edge devices. In SensiX++, we externalise all data operations pertained to a model, including data acquisition, data transformation, and featurisation. Furthermore, we leverage our declarative abstraction mechanism to automate and optimise these data operations and simultaneously serve multiple models, reducing operation redundancy by sharing and removing manual interventions.

— **Fast and direct access for deployment and query:** Today's edge devices are essentially all-streaming, dumb data producers. By embedding ML capability, we are slowly turning them into intelligent perception units. However, they still rely on the remote clouds for I/O and serving queries to applications. In SensiX++, we bring cloud-scale query service directly on the edge device to serve real-time queries and manage model deployment, thereby offering benefits in speed and availability. SensiX++ maintains a set of micro-services to operationalise these aspects and leverages its model-serving component for continuous deployment without any downtime.

4 SENSIX++: OVERVIEW

We begin by offering an overview of SensiX++ and its various components. In Section 5, we cover each of these components, underlying challenges, and technical details.

4.1 SensiX++ Components

Figure 1 shows the overall system architecture of SensiX++ composed of four main components:

— **Data Coordinator:** This component manages the sensor lifecycle and reads the sensor data in a unified manner through its sensor controller. It uses a transformation coordinator to transform sensor data into different formats meeting the requirements of different models (Section 5.1).

— **Adaptive Scheduler:** This component is responsible for system-wide orchestration. This component decides the execution schedule of multiple models balancing throughput and latency and informs the data requirement of each model, e.g., sensor type, sampling rate, and resolution, to the data coordinator (Section 5.2).

— **Model Server:** This component manages the execution of different models and creates model-specific containers meeting all runtime requirements, i.e., framework and libraries. These containers are then assigned to different processors (CPU, Mobile GPU, or TPUs) by the adaptive scheduler. This component maintains a featurisation coordination container that caches feature pipelines to serve multiple models requiring identical features (Section 5.3).

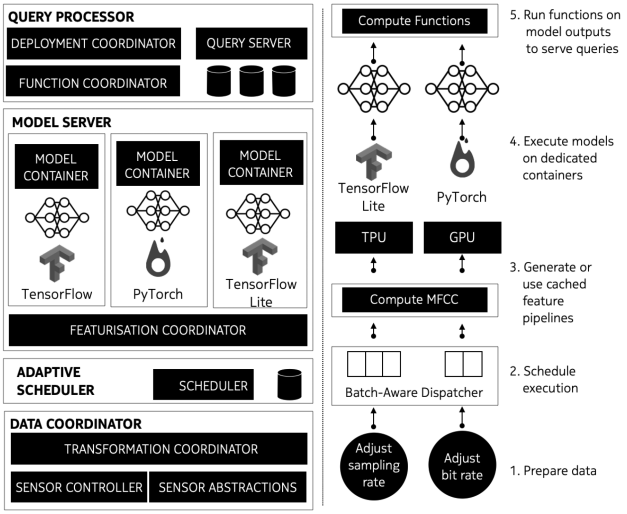


Fig. 1. System architecture (left) and an example operational flow for multiple acoustic models (right).

— **Query Processor:** This component is responsible for external interfaces. It maintains a deployment coordinator to receive model packages and deploy them on SensiX++ using a model server and adaptive scheduler. It also maintains a function coordinator that operates on model outputs to produce synthesised query responses served through its query server. Additionally, this component maintains small storage of model outputs to serve historical queries (Section 5.4).

We illustrate the various MLOps performed by these components to serve multiple models in Figure 1 (right) for a generic acoustic modelling task. Next, we discuss the declarative abstraction mechanism of SensiX++ that acts as the glue to facilitate the interplay between these components.

4.2 Declarative Abstraction for System Orchestration

An important design consideration for SensiX++ is to support the efficient execution of ML models on edge devices irrespective of their model architectures, runtime inference frameworks, and library dependencies. The challenge in achieving this vision is that an ML model is often released as a black box and does not contain metadata regarding its input format (e.g., RGB or YUV images), library dependencies, featurisation pipelines, and so on. Gaining visibility in these aspects of a model is critical to enable many of the system optimisations in SensiX++ and to achieve significant gains in inference throughput and latency.

In SensiX++, we require model developers to specify metadata about their model through a declarative abstraction mechanism using a *model manifest file*. Figure 2 shows the template of a model manifest file. It contains information about the input requirements of the model (e.g., image resolution and encoding), the inference framework (e.g., TensorFlow 2.0, PyTorch), names of the processors with which the model is compatible (e.g., CPU, GPU, TPU), and a unique global identifier for the feature extraction pipeline (more details in Section 5.3). Further, the manifest file also describes the library dependencies to run the model, a set of processor-specific weights of the neural network model, and an inference script that interprets the numeric outputs of the model (e.g., softmax probabilities) and converts them to human-readable classes. The developers can also specify the latency constraint as a performance requirement (more details in Section 5.2).

```
{
    "name": "Model Name",
    "vision_input": {
        "color": "encoding_type",
        "resolution": [ "height", "width" ]
    },
    "supported_processors": "cpu,gpu,tpu",
    "featuratisation_pipeline_id": "unique_pipeline_id",
    "featuratisation_pipeline_description": "URL to the source code of the
            featurization pipeline",
    "dependencies": {
        "packages": ["package 1", "package 2"],
        "weights": {
            "gpu": {
                "url": "URL for GPU model weights",
                "framework": "tensorflow2"
            },
            "tpu": {
                "url": "URL for TPU model weights",
                "framework": "tensorflow-lite"
            }
        },
        "inference": "inference_pipeline.py"
    },
    "performance_metrics": {
        "inference_latency": "time_in_milliseconds"
    }
}
```

Fig. 2. Template of a model manifest file.

We expect that the proposed declarative abstraction will significantly lower application developers' burden due to its simplicity. Also, it requires developers to provide less information than today's ML frameworks. The manifest file needs information only about the model (which is also needed for today's ML frameworks), but does not require the information to support different execution environments, e.g., handling input variabilities and addressing resource conflict from multiple models. In the coming sections, we will explain how this meta-information about the model is exploited by various components in SensiX++ to design novel system-level optimisations.

5 SENSIX++: COMPONENT DESCRIPTION

5.1 Data Coordinator

The lowest layer of SensiX++ is the *Data Coordinator*, which interfaces with sensors connected to a host device decoupling data production from data consumption (i.e., ML models) and represents the first aspects of MLOps to enable the automatic deployment of diverse ML models.

At a conceptual level, the data coordinator deals with *sensor capabilities* on one side and *model requirements* on the other. The sensor capabilities are intrinsic characteristics of the sensors. For example, the capabilities of a camera include sampling rate, resolution, and colour space (e.g., RGB or YUV). Since SensiX++ is designed to coordinate the execution of diverse models, not tailored to any specific hardware, the data coordinator layer needs to support models with different sensor data requirements. The model requirements define the expectations each model has on its input data to complete the computation and produce a valid inference result. Any deviation from these expectations will result in the failure of the model to execute or in severe degradation of its output quality (e.g., recognition accuracy).

In the multi-tenant environment considered in this work, it is very likely to have a mismatch between sensor capabilities and model requirements, especially when a set of models needs data from a smaller set of sensors. The challenge for the data coordinator is to resolve the mismatch between sensor capabilities and model requirements automatically and most efficiently, ensuring that all the models running are fed with the appropriate sensor data without introducing excessive latency and overhead for the host device. The mismatch resolution also needs to be transparent to model developers who are unaware of the sensors available on the hosts where their models will be deployed but provide the requirements via the manifest file.

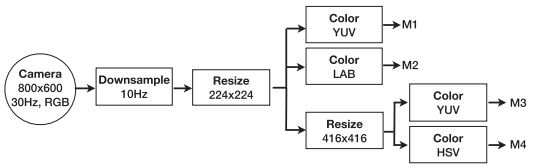


Fig. 3. Shared pipeline to serve four models.

Multi-model Sensor Data Transformation. Given the diverse and potentially incompatible requirements of the different models, the sensor-model mismatch cannot be resolved only by re-configuring the underlying sensors. Hence, SensiX++ proposes a two-stage approach where the system first carefully configures the sensors to produce data as similar as possible to the model requirements and, second, creates a shared pipeline to take care of the remaining transformations for each model.

- **Stage 1**: During the first stage, the data coordinator uses the information stored in each model's manifest file to find a common sensor configuration valid for all models. For example, if two vision models require images at 224 × 224 pixels and 416 × 416 pixels, respectively, this layer configures the camera to output images at 416 × 416 resolution or bigger if that resolution is not available. This avoids sub-optimal sensor configurations that might load the host device unnecessarily given the current models running (e.g., capturing 4k images when models only use 400 × 400 images).
- **Stage 2**: The second stage involves the creation of a pipeline to transform each sensor sample and finally meet the models' requirements. Instead of building an individual pipeline for each model, the data coordinator creates a shared pipeline for all models. This pipeline is manifested as a graph of transformations; Figure 3 depicts an example. In this example, four models need to be fed with images from the camera at 10 Hz, but they all require a different combination of resolution and colour space. By sharing the down-sampling and resizing operations across all models, the system avoids their repetition in the case when there is an individual pipeline for each model. We show in Section 6.3 how this results in significant savings in terms of CPU and memory load.

When building the shared pipeline, the data coordinator places transformations that reduce the number of samples or the size of each sample early in the pipeline (e.g., down-sampling or resizing) in order to reduce the operations performed at a later stage. Considering the example pipeline in Figure 3, the down-sampling to 10 Hz is the first operation in order to discard images that are not necessary as soon as possible, and hence save computation later in the pipeline. Similarly, for microphone data, it might be beneficial to reduce the sampling rate sooner to perform later operations on fewer data. Currently, we provide three transformations for cameras and three for microphones, but the data coordinator can easily be extended to include additional ones. For images, SensiX++ can resize their resolution, modify their colour space, or reduce their sampling rate (i.e., frames per second). For the microphone, the system can reduce the sampling rate, modify the bit depth of each sample, and aggregate samples in different windows (e.g., 1 s, 2 s).

The components of the data coordinator that implement these functionalities are shown in Figure 1. The Sensor Abstractions hide the details of each sensor and expose a uniform view to the upper layers of the system. The Sensor Controller manages and coordinates multiple instances of sensors, and the Transformation Coordinator selects the appropriate sensor configurations and runs the shared pipeline. Collectively, the interplay between these components leads to completely automated data management for serving models, which is an integrated part of MLOps.

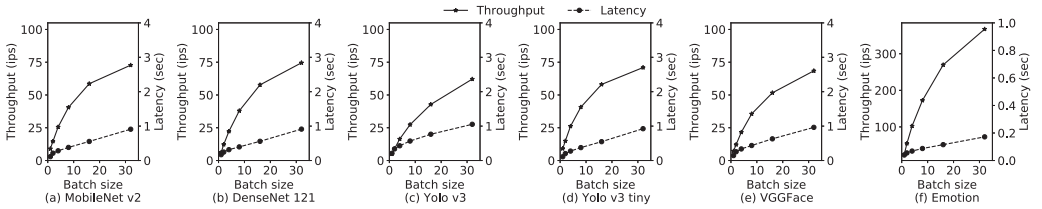


Fig. 4. Tradeoff between throughput (# of queries per second) and latency (sec).

In the next section, we present the Adaptive Scheduler layer, which sits on top of the Data Coordinator and is responsible for optimising the models' execution schedule.

5.2 Adaptive Scheduler

Due to the dynamic, unpredictable nature of deployment environments, it is infeasible for model-serving systems to guarantee the exact performance that models have at development time. This becomes more critical at the edge because it is almost infeasible to extend the resource availability on the fly dynamically. Thus, we design SensiX++ (1) to allow models to specify their latency requirements, rather than specifying an exact preferred throughput, and (2) to provide *best-effort* performance while meeting those latency budgets.

For a given set of input models, a key to improving their inference throughput without modifying models is to leverage batch processing. Batch processing enables fast and efficient computation by allowing the internal framework to exploit data-parallel processing, thereby increasing the number of inferences that can be computed per unit of time. However, the downside of batch processing is that it has a higher latency than doing a single inference. Figure 4 shows the relationships between the batch size, throughput, and latency for different models (details about the models are in Table 1).[2] Here, we define latency as the time from the data acquisition to the generation of a final inference output. Interestingly, the model throughput significantly increases even with a small increase in the batch size. For example, MobileNet v2 can be executed 9.1 times per second if the inference is continuously executed with a batch size of 1 (which is a common practice for real-time applications). By increasing the batch size to 4, the throughput is expected to increase to 26.7 inferences per second (about three times), but the maximum latency increases only from 0.11 to 0.26 seconds; here, the maximum latency is defined as the latency of the first sample in a batch including the queueing time (i.e., the time it takes the data sample to stay in the queue).

Our explorative study in Figure 4 indicates an important principle of providing best-effort inference performance. When a model's latency requirement is given in its manifest file, a straightforward method would be to profile the relationship between batch size and latency for the model, and select the maximum batch size for which the expected latency is less than the model's latency requirement. However, such a profile-based static decision does not work well in multi-tenant environments due to resource contention (especially, memory contention) between concurrent models. Figure 5 shows latency profiles of MobileNet v2 and VGGFace when they are running alone (w/o resource contention) and together with the other two vision models (w/ resource contention). The contention increases the latency, thereby changing the optimal batch size. For example, the optimal batch size of MobileNet v2 for the latency requirement of 300 ms changes from 4 (w/o contention) to 2 (w/ contention).

[2]Note that we do not argue that the model throughput and latency generally have a linear relationship to the batch size. The optimisation strategy of batch processing can differ depending on the accelerator architecture and framework implementation. We leave the modelling of the relationship between throughput/latency and batch size as future work.

Table 1. List of Models

	Task	Model	Framework
Vision	Image classification	MobileNet v2 [58]	TensorFlow v2
		MobileNet v2 (TPU) [58]	Coral TPU
		DenseNet 121 [27]	TensorFlow v2
		DenseNet 169 [27]	TensorFlow v2
		ResNet 50 v2 [26]	TensorFlow v2
		Inception v3 [61]	TensorFlow v2
	Object detection	Yolo v3 [57]	TensorFlow v2
		TinyYolo v3 [57]	TensorFlow v2
		MobileNet SSD v2 [58]	Coral TPU
	Face recognition	VGGFace (Senet50) [18]	TensorFlow v1
Audio	Emotion recognition	Emotion [4]	PyTorch
	Sound classification	YamNet [9]	TensorFlow Lite
	Keyword spotting	Res-8 [2]	Coral TPU

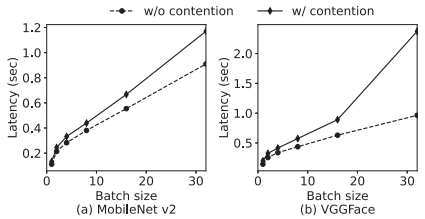


Fig. 5. Latency comparison w/ and w/o contention.

These results have two important implications. First, it is very costly to profile models by considering the contention effect because all possible combinations of concurrent models need to be addressed. Second, although possible, the profile is still inaccurate due to resource contention with other processes and daemons running on the device.

Adaptive Scheduling. To address the aforementioned challenge, we devise a lightweight, adaptive scheduler. The key idea is to monitor the runtime latency of each model and adjust the batch size dynamically. More specifically, the scheduler employs the additive increase/decrease method, inspired by TCP congestion control. Simply speaking, for each model, the scheduler monitors the end-to-end runtime latency of every inference and increases the batch size if the runtime latency is below the latency requirement. Similarly, it decreases the batch size if the runtime latency exceeds the requirement. Then, based on the selected batch size and runtime latency, the scheduler estimates the expected throughput (as *batch_size* / *runtime_latency*) and passes it to the data coordinator with the sampling rate of the corresponding sensor. Then, at runtime, the transformed data in the data coordinator is stored and maintained in the queue of the adaptive scheduler. The batch-aware dispatcher monitors the data queue and triggers the execution of the corresponding model if the number of samples in the queue reaches the batch size. It discards the samples if their queue time exceeds the latency requirement to avoid unnecessary processing.

Two practical issues in realising the scheduler are to determine (a) how often to change the batch size and (b) when to stop increasing the batch size if it is expected to reach the optimal size. First, the frequent change would enable the fast saturation to the optimal batch size, but also incur additional costs (e.g., pipeline re-organisation in data coordinator) and make an inaccurate decision due to the observation of small samples. Second, a naive implementation of the additive increase/decrease method would repeat to decrease and increase the batch size if it exceeds the optimal one, thereby wasting system resources.

ALGORITHM 1: Adaptive Scheduler for Multi-tenant Model Serving

```
Initialize: latencyRecords ← {}
for each model m do
    if m is not in latencyRecords then
        latencyRecords[m] ← {batchSize : 1, latencies : {}}
    end if
end for
while system is running do
    for each model m do
        batchSize ← latencyRecords[m]['batchSize'] + 1
        runtimeLatency ← measureLatency(m, batchSize)
        latencies ← getLatencies(m, batchSize)
        latencies.append(runtimeLatency)
        avgRuntimeLatency ← calculateAverage(latencies)
        if avgRuntimeLatency < LatencyReq then
            latencyRecords[m]['batchSize']+ = 1
        else if avgRuntimeLatency > LatencyReq then
            latencyRecords[m]['batchSize'] = max(0, latencyRecords[m]['batchSize'] − 1)
        end if
        estimatedThroughput ← batchSize/avgRuntimeLatency
        passToDataCoordinator(m, estimatedThroughput, samplingRate)
    end for
end while
```

To address these issues, the scheduler keeps track of the runtime latency and makes an informed decision based on the recent trend, not from a single, last observation. More specifically, it maintains $rl(m, b)$, where $rl()$ returns the average runtime latency of recent inferences for given model m, and batch size b. Then, to make a decision for the batch size of b, the scheduler compares the latency requirement with the expected latency of the increased batch size $rl(m, b+1)$, not the current latency ($rl(m, b)$). This enables the scheduler to efficiently spot the optimal batch size while minimising the increasing/decreasing trials. Algorithm 1 shows the detailed algorithm of the adaptive scheduling operation.

5.3 Model Server

In this section, we discuss how SensiX++ enables and optimises the deployment of multiple models on edge devices.

System-aware model containers. As the ML ecosystem on edge devices is evolving, there is a wide diversity of ML frameworks, feature extraction libraries, and hardware-specific models. For example, ML models could be developed using different versions of TensorFlow, PyTorch, and TensorFlow Lite, which may use feature extraction routines provided by OpenCV, Librosa, and NumPy, and could be designed to execute on specialised hardware (e.g., TPUs). This challenge of heterogeneity in the ML landscape is further compounded when multiple models run on the same edge device.

To achieve process isolation and avoid library or framework conflicts between multiple models, SensiX++ places each model in a separate Docker container. Although such container-based model deployments have been proposed earlier [20], SensiX++'s key novelty is that the process of creating

ALGORITHM 2: Container Creator

```
for each model m do
    modelVariants ← getModelVariants(m)
    P ← getAllAvailableProcessors()
    while true do
        P_min ← selectProcessor(P)
        modelVariant ← getModelVariantAvailableOnProcessor(modelVariants, P_min)
        if modelVariant is not None then
            createContainer(modelVariant, P_min)
            break
        else
            P ← P \ {P_min}
        end if
    end while
end for
```

ALGORITHM 3: Execution Coordinator

```
Initialize: workloadDictionary ← {}
for each processor p do
    workloadDictionary[p] ← getInferenceWorkload(p, k)
end for
```

docker containers and model execution pipelines is completely aware of the system state and is designed to keep multi-tenancy at its core.

In a multi-tenant system, various models compete to access the underlying hardware resources. For instance, due to the benefits associated with GPU acceleration, all model developers may individually want to run their inference pipeline on the GPU, by creating docker containers with GPU support. This can, however, congest the GPU and require constant paging-in and paging-out of parameters of different models from the GPU memory, leading to higher inference latency and lower inference throughput for each model.

SensiX++ addresses this resource contention challenge through a simple yet effective solution. In the model manifest file shown in Figure 2, model developers specify the hardware-specific variants of their models. For example, for an *object detection* task, a developer may provide a MobileNet v2 model (that runs on a GPU), a MobileNet v2-TensorflowLite variant (that runs on a CPU), and a MobileNet v2-TPU variant (that runs on a Coral Edge TPU). At the time of model deployment, the *Container Creator* component first interacts with the *Execution Coordinator* and obtains the current inference workload on each processor (e.g., CPU, GPU, Edge TPU) for the next k seconds, as described in Algorithms 2 and 3, respectively. Thereafter, it selects the processor P_min with the least inference workload as shown in Equation (1) and checks if the model weights provided by the developer are compatible with P_min. If yes, SensiX++ creates a container, specific to that processor (e.g., Edge TPU) by downloading the TPU-specific weights and inference pipeline provided by the developer as well as TPU-specific libraries. If the model developer has not provided a model compatible with P_min, SensiX++ looks for the next available processor in the system, and creates a model container specific to it.

$$selectProcessor(P) = \arg\min_{p \in P}(workloadDictionary[p]). \tag{1}$$

In effect, our proposed approach of system-aware container creation provides an early-stage load balancing and minimises resource contention in a multi-model system right before the models are about to be deployed.

Feature Caching. Despite the recent emphasis on end-to-end deep learning on raw data, many ML models still use dedicated feature extraction pipelines to generate features from the raw data and feed them to the ML model, e.g., Mel-filterbank generation for audio signals and frequency-domain feature generation for inertial sensor data. Interestingly, different models often use the same feature extraction pipelines before feeding the data to task-specific classifiers. This presents a clear system optimisation opportunity by caching the features generated by one model and reusing them for other models, thereby saving redundant feature computations.

A major challenge here is that ML models do not yet have a standardised way of specifying feature pipelines used by them. This implies that it is impossible to know if two models share the same pipeline, without doing a thorough code analysis. As a solution, SensiX++ envisions and proposes that feature pipelines are assigned a unique identifier based on the sequence of operations and parameters used in the pipeline. For example, the process of generating Mel-filterbank features from raw audio data involves decomposing the audio signal in frames, applying Discrete Fourier Transform on each frame to obtain its frequency-domain power spectrum, and re-scaling the power spectrum to the Mel scale. This sequence of steps and the parameters used in each step collectively constitute the feature extraction pipeline.

In SensiX++, model developers can specify a unique identifier for this feature extraction pipeline in the model manifest file. This proposal has parallels with how neural network architectures are assigned unique identifiers; for instance, *MobileNet v2* is simply an identifier for the collection of pre-defined computational layers with a fixed set of parameters (e.g., number and size of convolutional kernels, stride length). By adding similar metadata for feature extraction pipelines, we can facilitate the development of feature caching mechanisms across models that use the same pipeline.

Implementation of Feature Caching. SensiX++ performs feature extraction and caching in a dedicated docker container known as the *Featurisation Coordinator* as shown in Figure 1. When an inference request for a model X is triggered, the Featurisation Coordinator receives sensor data from the Execution Coordinator and executes the X's feature extraction pipeline on it. The output features are then fed to X's container for computing the inferences. At the same time, if another model Y registered on SensiX++ is using the same feature extraction pipeline, these output features computed for X are cached in memory and passed on to Y when it needs to compute an inference on the same data sample(s). This approach adds minimal overhead to the inference pipeline associated with caching and retrieving the output features. However, this overhead is negligible in comparison to the gains achieved by skipping the redundant feature extraction.

5.4 Query Processor

In this section, we discuss how SensiX++ interfaces with external applications, and handles their queries.

Function Coordinator. SensiX++ allows the execution of functions that process the outcome from model containers. These functions may serve several purposes, including generating higher-order analytics and annotations for the sensor data using predictions from one or more models or other functions. SensiX++ dedicates a container to facilitate the execution of such post-processing functions applying microservice principles, i.e., a certain function is executed when a particular request arrives. In our case, developers provide such functions during the deployment phase using two files. A *codelet* file provides a number of executable functions that

```python
def count(**kwargs):
    detected_objects = kwargs.get('detected_objects')# Get detected objects
            from the model
    count = 0
    for obj in detected_objects:
        if obj.get('class_name') == kwargs.get('object_to_count'): # Check if
                the detected model is the desired object
            count += 1
    return count

def draw_boxes(**kwargs):
    image = kwargs.get('original_image') # Get the original image
    detected_objects = kwargs.get('detected_objects');# Get detected objects
    for obj in detected_objects:
        if obj.get('class_name') in kwargs.get('objects_to_draw'):
            bbox = obj.get('bounding_box') #Get the bounding box
            cv2.rectangle(image, bbox[0:1], bbox[2:3], 'red', 2) # Draw the
                    bounding box
            cv2.putText(image, obj.get('class_name'), (bbox[0], bbox[1] - 5),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.5, 'red') # Write the label
    return image
```

Fig. 6. A code snippet for functions in the codelet.

use the outcome from one or more other functions and/or models. For each of these functions, a *functions* file lists models and/or other functions, whose outputs are necessary for execution, as well as the type of its outcome. Using this file, we follow an event-triggered approach where the execution of a function is prompted by the completion of these entities in the *functions* file.

Query Server and Data Store. To serve query requests from users and applications, SensiX++ includes a microservice with a number of API endpoints. SensiX++ can serve a single response or a stream. The APIs can be summarised below:

- `/models`: Get the list of models/
- `/functions`: Get the information about available post-processing functions/
- `/inference/:type/:function_id`: Get the outcome of a post-processing function with the given query type (single or stream).

In order to serve historical queries, SensiX++ maintains a LevelDB data store [6], and the query server leverages this data store to serve queries opportunistically. As the other system components already maintain continuous and fresh pipeline execution from capturing sensor data to post-processing functions and inserting the results into the database, the queries are responded to from the data store. This way, even if there are multiple queries with the same request, the response needs to be computed just once.

Deployment Coordinator. SensiX++ provides a document-based deployment mechanism to model and application developers through an external POST API (`/deploy`). This API accepts an archive of files to define the deployment, composed of a *manifest*, a *codelet* file to add functions to the post-processing container as shown in Figure 6, and a *functions* file that is used to create a chain of execution for post-processing functions. The manifest file lists a number of model manifests as explained in Section 4.2. This file is dispatched to the scheduler to create model containers. On the other hand, the codelet and functions files are forwarded to the post-processing container to add the new capabilities and guide pipeline creation from the model output to the query response.

6 EVALUATION

In this section, we evaluate SensiX++ with various vision and audio recognition models. First, we investigate the overall inference throughput achieved by SensiX++ across various multi-model configurations. Second, we conduct several micro-benchmarks to quantify and isolate the benefits of various components in SensiX++. Last, we report on an in-depth analysis of the costs and overheads associated with SensiX++.

Table 2. List of Workloads; the Parenthesised Number Represents the Latency Requirement in Milliseconds

Id	#	Models	Variation
W_1	3	MobileNet v2, VGGFace, Yolo v3	Default
W_2	3	DenseNet 121, TinyYolo v3, Emotion	Model
W_3	3	MobileNet v2 (TPU), MobileNet SSD v2, Keyword	
W_4	1	MobileNet v2	# of models
W_5	2	MobileNet v2, VGGFace	
W_6	4	MobileNet v2, VGGFace, Yolo v3, Emotion	
W_7	3	MobileNet v2 (500), VGGFace (500), Yolo v3 (500)	Latency requirement
W_8	3	MobileNet v2 (300), VGGFace (500), Yolo v3 (1000),	
W_9	3	DenseNet 169 (400), ResNet 50 v2 (400), Inception v3 (400)	Combination
W_{10}	3	MobileNet v2 (500), DenseNet 121 (500), YamNet (500)	

6.1 Experimental Setup

Models: We select a broad range of models mainly tailored for vision and audio recognition tasks, covering diverse types of objectives, frameworks, processors, and model architectures. Table 1 summarises the models.

Edge accelerators: For the experiments, we use NVidia Jetson AGX[3] (a GPU-powered edge board released by NVidia) with Google Coral TPU accelerator.[4] Jetson AGX hosts an 8-core Nvidia Carmel Arm and a 512-core Nvidia VoltaTM GPU with 64 Tensor Cores able to deliver up to 32 TOPs. CPU and GPU share a common bank of 32 GB of LPDDR4 RAM. Google Coral accelerator has a Google Edge TPU coprocessor supporting 4 TOPS (int8). As we are observing in today's smart cameras and speakers, we envision that such powerful edge accelerators will be naturally equipped with sensory edge devices in order to provide on-device intelligence.

Sensors: We use a Wisenet XNV-6080R camera[5] equipped with microphone to provide images and audio samples.

6.2 Overall Throughput

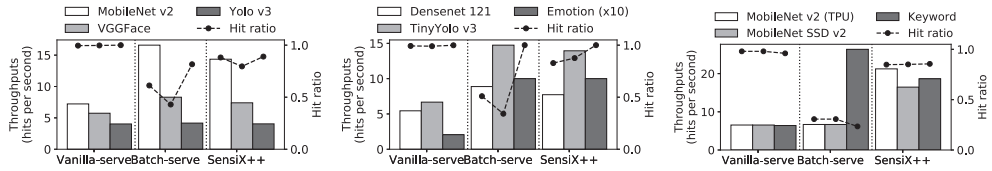
We investigate the overall throughput of SensiX++ across various multi-tenant configurations.

Workloads: We compose eight workloads to benchmark, by selectively using models in Table 1. Table 2 shows the details. We select W_1 (MobilNet v2, Yolo v3, VGG Face) as a default workload and vary diverse aspects for the comprehensive analysis in terms of model types (W_2, W_3), the number of concurrent models (W_4, W_5, W_6), and latency requirements (W_7, W_8). By default, we set 300 ms and 100 Hz to the latency requirements and the maximum inference rate for all models.

Baselines: We compare SensiX++ against two baselines:

— Vanilla Serving (Vanilla-serve): This baseline represents the current practice of ML model execution on existing model-serving systems such as TensorFlow Serving[6] and TorchServe.[7] For each model, *vanilla-serve* constructs a model container that includes the end-to-end inference pipeline in it (from data capture to model execution). As such, this baseline does

[3]https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit
[4]https://coral.ai/products/accelerator/
[5]https://www.hanwhasecurity.com/xnv-6080r.html
[6]https://www.tensorflow.org/tfx/guide/serving
[7]https://pytorch.org/serve/



(a) W_1 (MobileNet v2, VGG Face, Yolo v3 (b) W_2 (DenseNet 121, TinyYolo v3, Emotion) (c) W_3 (MobileNet v2, MobileNet SSD, KeyWord on Coral TPU)

Fig. 7. Overall throughput across different multi-tenant configurations.

not share any data operations with other models. The model inference is triggered when requested, i.e., whenever the image frame is generated, so Vanilla-serve does not perform batch processing, i.e., the batch size = 1.

— Static-batch Serving (Batch-serve): This baseline is an optimised version of Vanilla-serve. It builds over the Vanilla-serve baseline by adding batch processing to maximise the inference throughput. However, it does not account for the resource contention issues in a multi-model setting (as discussed in Section 5.2). When a model is registered, it profiles the end-to-end latency with different batch sizes (as in Figure 4) and finds the maximum batch size that meets the given latency requirement. Then, it continuously computes the inference with the selected batch size regardless of the workload of other concurrent models. Note that, while today's ML frameworks support batch processing, application developers need to manually specify the batch size to use.

Metrics: We measure the effectiveness of SensiX++ by measuring model inference *throughput* that meets the latency requirement of the model as specified in the manifest file. More specifically, we count the number of inferences per second, for which the end-to-end latency is below the latency requirement of the model. We define end-to-end latency as the time from data acquisition to model inference. We further measure the efficiency of SensiX++ by measuring the *hit ratio*, defined as a ratio between (a) the number of data samples that succeed in meeting the latency requirement and (b) the total number of data samples generated. A higher hit ratio indicates higher resource efficiency, i.e., system resources are optimally used without waste.

6.2.1 Overall Performance. Figure 7 shows the model inference throughput of the workloads, W_1, W_2, W_3; the bars and line represent the throughput and hit ratio, respectively. The results show that SensiX++ achieves higher throughput than Vanilla-serve by virtue of its batch processing and operation sharing. More specifically, SensiX++ achieves up to 60% throughput increase in the default workload, $W1$. The throughput of MobileNet v2, Yolo v3, and VGGFace increases from 7.2, 5.7, and 4.0 to 14.4, 7.4, and 4.0, respectively. However, the throughput of Yolo v3 does not increase meaningfully due to its heavy processing load; batch processing of Yolo v3 with the size of 2 already exceeds the latency requirement, so SensiX++ does not increase its batch size.

We also compare the performance between SensiX++ and Batch-serve. Interestingly, both schemes show comparable throughput across the models, but SensiX++ shows much higher hit ratios, which implies the optimal use of resources. In Batch-serve, the batch size of a model is determined independently to other models, thus the actual throughput the model achieves is lower than the expected one from the selected batch size due to resource contention. Thus, in W_1, the hit ratio of VGGFace of Batch-serve decreases down to 0.43, meaning that more than half of the data samples generated in a camera are dropped in the queue, or the output of the model inferences is discarded due to the violation of the latency requirement. Unlike Batch-serve, SensiX++ achieves

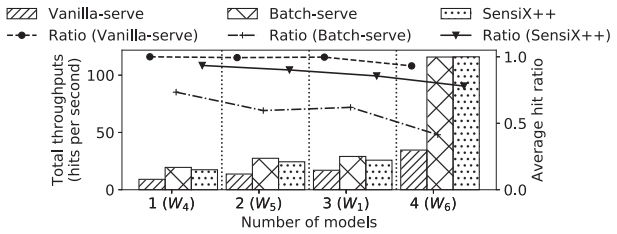


Fig. 8. Effect of the number of models.

the hit ratio of nearly 1 even with unpredictable, fluctuating concurrent workloads. We investigate the performance with different combinations of models in W_2 and observe a similar trend to W_1. Similar to W_1, relatively lightweight models (TinyYolo v3 and Emotion) largely benefit from batch processing, e.g., throughput increase of SensiX++ by up to 2× and 50× compared with Vanilla-serve, respectively; note that we scaled down the throughput of Emotion in Figure 7(b).

We further investigate the SensiX++ performance in the Coral TPU framework with W_3. The Coral TPU framework has different characteristics from other frameworks. First, it does not allow concurrent access to Coral TPUs from different processors. Thus, we develop a unified container that manages all the inferences of TPU models and adopts a round-robin scheduler for fairness. Second, it does not support batch processing and corresponding internal optimisation in the framework due to the limited memory (8 MB). However, when there are multiple models, the execution of a batch of samples at once in the TPU framework also enables high throughput. This is because Coral TPU can afford a very limited number of models in the memory (usually, only one vision model) and needs to write model data every time when a new model is loaded. Thus, the first time the model runs is always slower than the later times, and TPU models also show a similar pattern as shown in Figure 4.

Figure 7(c) shows that SensiX++ outperforms the other baselines in the TPU framework as well. SensiX++ achieves the throughput increase by up to around three times for all models. More specifically, the throughput of MobileNet v2, MobileNet SSD v2, and Keyword increases up from 6.5 (Vanilla-serve) to 21.2, 16.5, and 18.7 (SensiX++), respectively. Interestingly, Batch-serve does not increase the throughput of MobileNet v2 and MobileNet SSD. This is because Batch-serve determines the batch size and sampling rate of each model under the assumption that the very model is running alone. However, due to the lack of parallel execution capability of the TPU framework, the queueing time of data samples becomes longer than expected in order to wait for other models to be completed, and many of them are discarded due to the violation of the latency requirement. On the contrary, Batch-serve increases the throughput of the Keyword model by having a relatively higher sampling rate, i.e., less number of samples are discarded. However, we can observe that a large portion of data samples is still discarded; the hit ratio of Keyword in Batch-serve is 0.23.

6.2.2 Effect of Number of Models. Figure 8 shows the total throughput and average hit ratio while varying the number of models. The results show that, across the different number of models, SensiX++ achieves higher throughput than Vanilla-serve while maintaining higher hit ratios than Batch-serve. More specifically, the total throughput increases from 9.1, 13.7, 16.9, and 34.6 (Vanilla-serve) to 17.4, 24.5, 25.8, and 116.0 (SensiX++). In the heavy workload (W_6), Batch-serve shows comparable throughput, but a much lower hit ratio (0.4) than SensiX++ (0.8). We further delve into a detailed analysis of multi-tenancy. When the number of models is one, specifically W_4, both Batch-serve and SensiX++ show higher throughput than Vanilla-serve, as both exploit

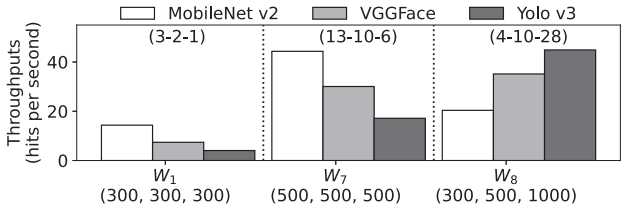
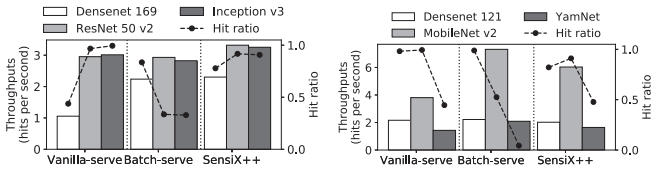


Fig. 9. SensiX++ with different latency requirements.

larger batch sizes. Interestingly, even with a single model, the hit ratio of Batch-serve is relatively lower than SensiX++ because it uses a fixed batch size obtained during the offline phase, and the actual model execution time can fluctuate at runtime. As the number of models escalates, from W_4 to W_5, W_1, and W_6, resource contention among models intensifies, leading to a decrease in the hit ratio across all baselines. Despite this, it is noteworthy that SensiX++ still maintains a reasonable hit ratio of approximately 0.8, even when there are four models. As shown in Algorithm 1, this is due to SensiX++'s dynamic batch size adjustment for each model based on its runtime execution time, reflecting the background workloads. However, the lack of this capability in Batch-serve results in a significant decrease in the hit ratio, even though it demonstrates throughput similar to SensiX++.

6.2.3 Performance with Different Latency Requirements. Figure 9 shows the behaviour of SensiX++ for different latency requirements. The parenthesised numbers in the X-axis are the latency requirement of each model and the numbers in the box represent the batch size that was selected the most during runtime. For example, the most selected batch sizes of MobileNet v2, VGGFace, and Yolo v3 in W_1 are 3, 2, and 1, respectively. The results show that SensiX++ guarantees higher throughput when the loose latency requirement is used. More specifically, when the latency requirement is set to 500 ms in W_7, the throughput of three models increases from 14.4, 7.4, and 4.0 (W_1) to 44.3, 30.0, and 17.2 (W_7). This is enabled by taking the longer batch size from the looser latency requirement. For example, the throughput increase of Yolo v3 achieves up to four times (W_7) only at the latency expense of 200 ms. W_8 represents the case when concurrent models have different latency requirements, i.e., 300, 500, and 1000 ms. The results show that the adaptive scheduler of SensiX++ well distributes the resource use based on different requirements. Compared to W_1, the throughput increase is different depending on the increase of latency requirement of the model. For example, the throughput increase of VGGFace is 22.6 when the latency requirement is set from 300 ms (W_1) to 500 ms (W_8), but the increase of Yolo v3 is 40.8 when its latency increases from 300 ms (W_1) to 1,000 ms (W_8). Interestingly, we observe the throughput of MobileNet v2 increases even with the same latency requirement, e.g., 14.4 (W_1) to 20.4 (W_8). This is because other models use fewer resources from the longer batch size in W_8, and MobileNet v2 can be assigned with more available resources.

6.2.4 Testing with More Workloads, Devices, and Baselines. **More workloads:** We further show the applicability of SensiX++ to different workload settings and edge devices. First, we devise a new workload, W_9, with three models (DenseNet 169, ResNet 50 v2, Inception v3). Since these models are relatively heavier than the models in the W_1, i.e., longer execution time, we set the latency requirement to 400 ms. Figure 10(a) shows the model inference throughput and hit ratios of the workload, W_9. Similar to the results in Section 6.2.1, SensiX++ shows higher throughput than Vanilla-serve. More specifically, the total throughput of Vanilla-serve is 7.0 whereas the SensiX++'s total throughput is 8.9 (27% increase). Interestingly, the results also show that SensiX++



(a) W_9 (DenseNet 169, ResNet 50 v2, Inception v3

(b) W_{10} (MobileNet v2, DenseNet 121, YamNet on Jetson NX)

Fig. 10. Testing with more workloads and devices.

outperforms Batch-serve in terms of both throughput and hit ratio. The total throughput and average ratio of Batch-serve are 8.0 and 0.5, whereas those of SensiX++ are 8.9 and 0.9.

More devices: Next, we investigate the performance of SensiX++ on Nvidia Jetson NX.[8] Jetson NX hosts a 6-core Nvidia Carmel Arm and 384 Nvidia CUDA cores and 48 Tensor cores. 8 GB of LPDDR4x RAM is shared by CPU and GPU. Figure 10(b) shows the performance of SensiX++ on Jetson NX with W_{10} (MobileNet v2, DenseNet 121, YamNet). Since Jetson NX is less powerful than Jetson AGX, we set the latency requirement of these models to 500 ms. Even on a different edge device, Jetson NX, we can observe that SensiX++ provides a similar performance trend to the results on Jetson AGX, which shows the generalisability of SensiX++. More specifically, SensiX++ increases the total throughput from 7.4 (Vanilla-serve) to 9.7. Similarly, SensiX++ shows comparable throughput to Batch-serve, but it increases the average hit ratio from 0.52 (Batch-serve) to 0.74.

More baselines: Here, we extend the system performance comparison to include two additional baselines. As previously mentioned, Batch-serve is an optimised version of Vanilla-serve, designed to find the maximum batch size that meets the latency requirements for a given device. However, Batch-serve is not yet supported by current ML frameworks. In the absence of runtime device information during application development, developers must manually specify the batch size based on their own target environment. Consequently, the batch size remains static, irrespective of the platform the model is deployed on. To quantify the impact of this practice and highlight the system-driven adaptation, we incorporated two more baselines–Batch-serve(AGX) and Batch-serve(Nano)–into the experiment with W_{10} performed on Jetson NX. Batch-serve(AGX) and Batch-serve(Nano) utilise profiles obtained from Jetson AGX and Nano,[9] respectively. These profiles represent scenarios where developers target Jetson AGX and Jetson Nano, but deploy models on Jetson NX. Table 3 shows the total throughput and average hit ratio of three models W_{10} (MobileNet v2, Dense 121, YamNet). When the target platform is Jetson AGX but the deployed platform is Jetson NX, the average hit ratio drops from 0.52 (Batch-serve) to 0.38 (Batch-serve(AGX)), due to increased resource contention among models. This happens because Jetson AGX is more powerful than Jetson NX and consequently larger batch sizes are used for all three models. Interestingly, the total throughput of Batch-serve(AGX) is similar to Batch-serve, as both fully utilise CPU resources. Contrarily, Batch-serve(Nano) reduces the total throughput drastically from 11.63 (Batch-serve) to 5.72 (Batch-serve(Nano)), despite an increased average hit ratio. This is due to the fact that Jetson Nano is less powerful than Jetson NX, leading Batch-serve(Nano) to use smaller batch sizes than Batch-serve. While Batch-serve outperforms both Batch-serve(AGX) and Batch-serve(NX), its performance lags behind that of SensiX++ due to its inability to holistically schedule all concurrent models.

[8] https://www.nvidia.com/en-gb/autonomous-machines/embedded-systems/jetson-xavier-nx/
[9] https://www.nvidia.com/en-gb/autonomous-machines/embedded-systems/jetson-nano/

Table 3. Performance Comparison with More Baselines

Metrics	Batch-serve	Batch-serve(AGX)	Batch-serve(Nano)	SensiX++
Total throughput (hits per second)	11.63	11.45	5.72	9.69
Average hit ratio	0.52	0.38	0.88	0.74

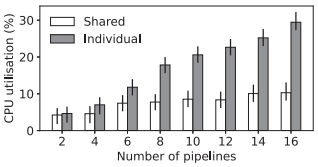


Fig. 11. Average CPU utilisation for shared and individual sensor pipelines.

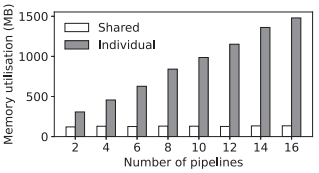


Fig. 12. Memory utilisation for shared and individual sensor pipelines.

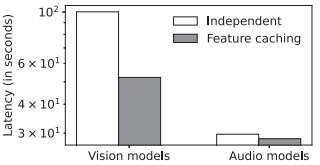


Fig. 13. Latency reduction due to feature caching.

6.3 Micro-Benchmarks

We conduct micro-benchmarks to quantify the benefits of SensiX++ components. First, we further break down the performance of SensiX++, mainly focusing on the benefits of sharing in two components: (a) a shared transformation pipeline in the data coordinator and (b) feature caching in the model server. Second, we investigate the performance gain from the system-aware container creation.

6.3.1 Benefits of Shared Transformation Pipeline. To evaluate the benefit of sharing the transformation pipeline across multiple models, we created 16 pipelines inspired by the requirements of the models listed in Table 1. We deployed these pipelines first using the sharing approach proposed by SensiX++ (Section 5.1) and then as individual pipelines. The latter represents a scenario where each model is deployed individually without awareness of other models running on the system and their requirements, resulting in repetitions in the transformations they perform on the sensor data. Figures 11 and 12 report the CPU utilisation and resident set size utilisation of the data coordinator, respectively. We notice that as the number of pipelines increases, the benefit of sharing operations results in significant resource savings. The individual pipelines use more resources proportional to the number of pipelines deployed, while the sharing approach scales more slowly with the number of pipelines because operations within the pipelines are executed only once. Already with four pipelines deployed, we notice significant benefits in the sharing approach, which saves 35% of CPU and 80% of memory compared to the individual pipelines.

6.3.2 Latency Reduction using Feature Caching. To demonstrate the effect of SensiX++'s Feature Caching component, we evaluate it on two workloads:

— A set of three audio recognition models, namely, EmotionNet for emotion recognition, YamNet for acoustic event detection, and Res-8 for Keyword Detection. While these models have different inference tasks, they share the same feature extraction pipeline, which involves computation of Mel-filterbank features from raw audio.
— A set of three visual recognition models, namely, MobileNetV2, DenseNet121, and YoloV3. For these models, we consider *image translation* as the shared featurisation operation. Image Translation is a popular mechanism to reduce domain shift (i.e., the divergence between training and test data distributions) [40]. It involves passing the test image to a pre-trained translation model (e.g., Pix2Pix [29]) to obtain a translated image, which is closer to the training data distribution.

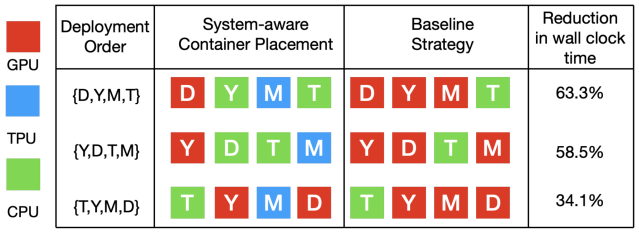


Fig. 14. Our system-aware container creation provides a significant reduction in the wall clock time for computing inferences for multiple models.

Figure 13 illustrates the latency reduction achieved by feature caching as compared to the naïve baseline when each model independently performs the featurisation operations. For the vision models, we obtain a reduction of 48% in the end-to-end to inference latency. This is primarily because the image translation operation is expensive to perform; hence, by doing it once and caching its results for other models results in significant latency improvements. For the audio models, we observe a 5% latency reduction by sharing the Mel-filterbank generation pipeline across models. As this operation is much cheaper than image translation, the latency reduction is smaller. In summary, our results show that by requiring the model developers to provide a trivial piece of metadata about their feature extraction pipelines, SensiX++ can offer the ability of feature caching and provide clear latency improvements.

6.3.3 Performance Gains from System-Aware Container Creation. We now compare our approach to the system-aware creation of model containers against the existing paradigm where model containers are unaware of the system state. Four visual recognition models are used for this experiment, namely, **D**ense121 (CPU or GPU), **M**obileNetV2 (CPU, GPU, or Coral TPU), **T**inyYoloV3 (CPU), and **Y**oloV3 (CPU or GPU). The values in the parentheses denote the processors on which a model can be executed. Recall that the processor-specific model weights are specified in the manifest file.

In Figure 14, we show three scenarios for deploying these models. In each scenario, we take a deployment order, e.g., {**D,Y,M,T**} indicates that **D** is first deployed on the system followed by **Y**, **M**, and **T**. When a model is about to be deployed, we check the current resource utilisation of each processor and assign the model to a processor based on its availability and compatibility with the model. For {**D,Y,M,T**}, our algorithm assigns **D** to the GPU, **Y** to the CPU, **M** to the TPU, and **T** to the CPU. In the absence of this system-aware strategy, a baseline strategy would have assigned **D**, **Y**, and **M** to the GPU considering the benefits of GPU acceleration, which would have led to resource contention on the GPU in our multi-model scenario and increased the wall clock time of computing inferences for all the models. Our proposed strategy outperforms the baseline by providing 34–63% reduction in wall clock time for computing inferences on all the models.

6.4 System Overhead Analysis

We conduct an in-depth cost analysis to better understand the runtime behaviour of SensiX++. To bring MLOps and multi-tenant model serving, besides the operations originally required for the model inference, SensiX++ additionally performs the following operations: (a) the container creation when a model is added, (b) the data transformation, (c) adaptive scheduling at runtime, and (d) applying post-processing functions and interfacing with a data store in query serving phase. Since the cost of adaptive scheduling is negligible (<1 ms), we focus on the other costs here.

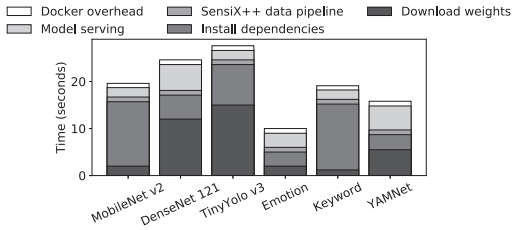


Fig. 15. Time taken for creating model containers.

Table 4. Average Latency of Data Coordinator Operations

	Operation	Latency (ms)
	Abstractions only	0.51 (0.11)
Vision	Sampling rate	0.025 (0.053)
	Resolution	1.29 (0.16)
	Colour space	0.30 (0.082)
Audio	Aggregation window	0.024 (0.028)
	Bit depth	7.44 (0.25)
	Sampling rate	9.42 (0.21)

Overhead of container creation: Figure 15 shows the time breakdown of various steps involved in creating the docker-based execution pipeline for six different ML models. We observe that the end-to-end deployment of the models in SensiX++ takes less than 30 seconds. The bulk of this time is spent in downloading the model weights over the network and installing model-specific library dependencies (e.g., for feature extraction) as specified in the model manifest file. SensiX++'s overhead, which includes integrating the *Data coordinator* and *Adaptive scheduler* with the model's inference pipeline and exposing the model interface as a REST API, is minimal and ranges between 4 and 7 seconds.

Sensor abstractions and transformations latency: The abstractions and transformations provided by the data coordinator are intended to offer a uniform view of heterogeneous hardware and simplify the deployment of ML models that have not been specifically developed for the sensors in use. As such, these operations should contribute minimally to the overhead of the system and introduce minimal latency in the dispatch of the sensor data to the other components in the system. Table 4 reports the latency of the individual operations performed by the data coordinator.[10] We notice that the latency introduced by the abstractions is limited since this is a thin layer over the sensor drivers that transfer data to other components in the system. Similarly, the transformations that reduce the frames per second or aggregate audio samples in different windows do not apply any actual transformation to the data but drop unnecessary frames or aggregate audio samples, hence they introduce very short latency. The other data transformations instead take more time to compute (in particular, for acoustic transformations) since they perform modifications to actual samples before dispatching them.

Query serving latency: Figure 16 shows the latency associated with query serving for four different tasks accumulated by three different factors: executing the post-processing function, writing the function output to data store, and reading from the data store to serve a query. Across this variety of tasks, SensiX++ manages to respond to queries within roughly 10 ms.

[10]We used 640 × 480 images and audio samples of 1 second at 32 kHz sampling rate.

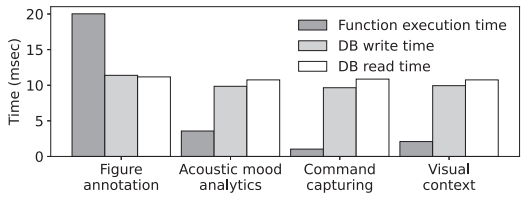


Fig. 16. Latency associated with query serving.

Power cost for model inference serving: We investigate the power cost required for serving model inferences, using software-based power consumption modelling[11] provided by Nvidia. Interestingly, for the same workload, the difference in the total energy consumption between SensiX++ and the baselines is marginal, showing no statistical significance. For example, Jetson AGX consumes around 13 J/s on SensiX++, Vanilla-serve, and Batch-serve with the MAXN power mode; the difference was around 0.15 J/s. This is mainly because idle power of Jetson AGX (i.e., the baseline power without any workload) is already high. In addition, the MAXN power mode disables the **dynamic voltage and frequency scaling (DVFS)**, i.e., using static voltage and frequency of each processor, which makes less variations of energy cost depending on the runtime workload. For example, when we break down the energy consumption, 13 J/s, the baseline energy is 8.8 J/s but the additional energy consumed by ML execution is 4.2 J/s; the additional energy cost on Vanilla-serve and Batch-serve is around 4.05 J/s and 4.3 J/s, respectively. We conjecture that this is mainly because these devices are not designed to run on batteries and thus adopt less power-optimised processors. We leave the investigation of the power consumption of SensiX++ on battery-powered devices as future work.

6.5 Summary of Results

- SensiX++ provides much higher throughput than today's vanilla serving, e.g., achieving up to 60% throughput increase when three models (MobileNet v2, Yolo v3, and VGGFace) are concurrently running. SensiX++ also uses computing resources more optimistically than static batch processing. Both of them show comparable throughput, but SensiX++ shows a 38% increase in hit ratio for the same workload.
- Across the different number of models, SensiX++ achieves higher throughput than vanilla serving while maintaining higher hit ratios than static batch serving, e.g., two to three times increase in inference throughput compared to vanilla serving.
- With different latency requirements of ML models, SensiX++ well distributes the resource use based on different requirements.
- Through multiple micro-benchmarks, we show that SensiX++ achieves the performance gain by sharing redundant data operations and system-aware container creation.
- We investigate the overhead of the SensiX++ operations in diverse aspects and show that SensiX++ has several benefits with low overhead.

7 OUTLOOK

We have presented SensiX++, a multi-tenant runtime for model execution with integrated MLOps on edge devices. Thanks to the modular design, SensiX++ enables great flexibility in the deployment of multiple models efficiently with fine-grained control on edge devices. It minimises data

[11] https://docs.nvidia.com/jetson/archives/l4t-archived/l4t-325/index.html#page/Tegra%20\Linux%20Driver%20Package%20Development%20Guide/power_management_jetson_xavier.html#wwpID0E0AG0HA

operation redundancy, manages data and device heterogeneity, reduces resource contention, and enables automatic MLOps. Our benchmarks on an edge device highlight the simplicity of deploying and coordinating diverse vision and acoustics models and the benefits that arise from the automation offered by its key components (i.e., model server, adaptive scheduler, and data coordinator). This is a significant step forward compared to current edge MLOps, which do not consider multi-tenant scenarios or treat each model as an independent silo without benefiting from system-aware sharing across models and holistic coordination as offered by SensiX++.

In the current implementation, SensiX++ takes a container as a basic unit of the model execution, i.e., running one model on a single container. It guarantees independence and portability of ML execution environments, but can also incur the waste of storage space, especially when different models share common frameworks or runtime dependencies. We expect that this problem can be addressed by the re-design of the docker container architecture, e.g., allowing the modular execution of the docker container and adopting efficient communication across containers.

In this article, we mainly focus on a single sensory device serving multi-tenant models. However, we envision that ML collaboration between nearby or remote sensory devices will be prevalent to support a higher quality of service and seamless operations, but it will be more challenging to be realised, e.g., dynamic discovery of available devices, holistic orchestration of distributed resources, and communication fault-tolerant scheduling. We plan to extend SensiX++ to support MLOps to multiple, distributed edges by addressing the aforementioned challenges.

REFERENCES

[1] 2021. BentoML. (2021). Retrieved August 8, 2023 from https://www.bentoml.ai

[2] 2021. Coral Keyphrase Detector. (2021). Retrieved August 8, 2023 from https://github.com/google-coral/project-keyword-spotter

[3] 2021. ElectrifAI. (2021). Retrieved August 8, 2023 from https://electrifai.net

[4] 2021. Emotion Classification. (2021). Retrieved August 8, 2023 from https://github.com/Data-Science-kosta/Speech-Emotion-Classification-with-PyTorch/

[5] 2021. KubeFlow. (2021). Retrieved August 8, 2023 from https://www.kubeflow.org

[6] 2021. LevelDB. (2021). Retrieved August 8, 2023 from https://github.com/google/leveldb

[7] 2021. Michelangelo. (2021). Retrieved August 8, 2023 from https://eng.uber.com/michelangelo-machine-learning-platform/

[8] 2021. SageMaker. (2021). Retrieved August 8, 2023 from https://aws.amazon.com/sagemaker/

[9] 2021. YAMNet. (2021). Retrieved August 8, 2023 from https://github.com/tensorflow/models/tree/master/research/audioset/yamnet

[10] Utku Günay Acer, Marc van den Broeck, Chulhong Min, Mallesham Dasari, and Fahim Kawsar. 2022. The city as a personal assistant: Turning urban landmarks into conversational agents for serving hyper local information. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 6, 2 (July 2022), Article 40, 31 pages. https://doi.org/10.1145/3534573

[11] Mattia Antonini, Miguel Pincheira, Massimo Vecchio, and Fabio Antonelli. 2022. Tiny-MLOps: A framework for orchestrating ML applications at the far edge of IoT systems. In *Proceedings of the 2022 IEEE International Conference on Evolving and Adaptive Intelligent Systems (EAIS'22)*. 1–8. https://doi.org/10.1109/EAIS51927.2022.9787703

[12] Mattia Antonini, Tran Huy Vu, Chulhong Min, Alessandro Montanari, Akhil Mathur, and Fahim Kawsar. 2019. Resource characterisation of personal-scale sensing models on edge accelerators. In *Proceedings of the 1st International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things (AIChallengeIoT'19)*. ACM, New York, NY, 49–55. https://doi.org/10.1145/3363347.3363363

[13] Tayebeh Bahreini and Daniel Grosu. 2017. Efficient placement of multi-component applications in edge computing systems. In *Proceedings of the 2nd ACM/IEEE Symposium on Edge Computing (SEC'17)*. Association for Computing Machinery, New York, NY, Article 5, 11 pages. https://doi.org/10.1145/3132211.3134454

[14] Abu Bakar, Tousif Rahman, Alessandro Montanari, Jie Lei, Rishad Shafik, and Fahim Kawsar. 2022. Logic-based intelligence for batteryless sensors. In *Proceedings of the 23rd Annual International Workshop on Mobile Computing Systems and Applications (HotMobile'22)*. Association for Computing Machinery, New York, NY, 22–28. https://doi.org/10.1145/3508396.3512870

[15] Abu Bakar, Tousif Rahman, Rishad Shafik, Fahim Kawsar, and Alessandro Montanari. 2022. Adaptive intelligence for batteryless sensors using software-accelerated tsetlin machines. In *Proceedings of the 20th Conference on Embedded*

Networked Sensor Systems (SenSys'22). Association for Computing Machinery, New York, NY. https://doi.org/10.1145/3560905.3568512

[16] Sourav Bhattacharya and Nicholas D. Lane. 2016. Sparsification and separation of deep learning layers for constrained resource inference on wearables. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM (SenSys'16)*. Association for Computing Machinery, New York, NY, 176–189. https://doi.org/10.1145/2994551.2994564

[17] Henrik Blunck, Niels Olof Bouvin, Tobias Franke, Kaj Grønbæk, Mikkel B. Kjaergaard, Paul Lukowicz, and Markus Wüstenberg. 2013. On heterogeneity in mobile sensing applications aiming at representative data collection. In *Proceedings of the 2013 ACM Conference on Pervasive and Ubiquitous Computing Adjunct Publication (UbiComp'13 Adjunct)*. Association for Computing Machinery, New York, NY, 1087–1098. https://doi.org/10.1145/2494091.2499576

[18] Qiong Cao, Li Shen, Weidi Xie, Omkar M. Parkhi, and Andrew Zisserman. 2018. VGGFace2: A dataset for recognising faces across pose and age. In *Proceedings of the 2018 13th IEEE International Conference on Automatic Face Gesture Recognition (FG'18)*. 67–74. https://doi.org/10.1109/FG.2018.00020

[19] Daniel Crankshaw, Peter Bailis, Joseph E. Gonzalez, Haoyuan Li, Zhao Zhang, Michael J. Franklin, Ali Ghodsi, and Michael I. Jordan. 2015. The missing piece in complex analytics: Low latency, scalable model management and serving with velox. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research (CIDR'15)*. www.cidrdb.org. http://cidrdb.org/cidr2015/Papers/CIDR15_Paper19u.pdf

[20] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A low-latency online prediction serving system. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*. USENIX Association, Boston, MA, 613–627. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw

[21] Biyi Fang, Xiao Zeng, and Mi Zhang. 2018. NestDNN: Resource-aware multi-tenant on-device deep learning for continuous mobile vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking (MobiCom'18)*. Association for Computing Machinery, New York, NY, 115–127. https://doi.org/10.1145/3241539.3241559

[22] Petko Georgiev, Nicholas D. Lane, Kiran K. Rachuri, and Cecilia Mascolo. 2016. LEO: Scheduling sensor inference algorithms across heterogeneous mobile processors and network resources. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking (MobiCom'16)*. Association for Computing Machinery, New York, NY, 320–333. https://doi.org/10.1145/2973750.2973777

[23] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. 2014. Towards wearable cognitive assistance. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'14)*. ACM, New York, NY, 68–81. https://doi.org/10.1145/2594368.2594383

[24] Song Han, Huizi Mao, and William J. Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. https://arxiv.org/abs/1510.00149

[25] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. 2016. MCDNN: An approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'16)*. Association for Computing Machinery, New York, NY, 123–136. https://doi.org/10.1145/2906388.2906396

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Identity Mappings in Deep Residual Networks. (2016). https://doi.org/10.48550/ARXIV.1603.05027

[27] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. 2018. Densely connected convolutional networks. https://arxiv.org/abs/1608.06993

[28] Shawn Hymel, Colby Banbury, Daniel Situnayake, Alex Elium, Carl Ward, Mat Kelcey, Mathijs Baaijens, Mateusz Majchrzycki, Jenny Plunkett, David Tischler, Alessandro Grande, Louis Moreau, Dmitry Maslov, Artie Beavis, Jan Jongboom, and Vijay Janapa Reddi. 2022. Edge Impulse: An MLOps Platform for Tiny Machine Learning. Retrieved from https://doi.org/10.48550/ARXIV.2212.03332

[29] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. 2017. Image-to-image translation with conditional adversarial networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'17)*.

[30] Joo Seong Jeong, Jingyu Lee, Donghyun Kim, Changmin Jeon, Changjin Jeong, Youngki Lee, and Byung-Gon Chun. 2022. Band: Coordinated multi-DNN inference on heterogeneous mobile processors. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services (MobiSys'22)*. Association for Computing Machinery, New York, NY, 235–247. https://doi.org/10.1145/3498361.3538948

[31] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. 2017. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *SIGARCH Compututer Architecture. News* 45, 1 (April 2017), 615–629. https://doi.org/10.1145/3093337.3037698

[32] Fahim Kawsar, Chulhong Min, Akhil Mathur, and Alesandro Montanari. 2018. Earables for personal-scale behavior analytics. *IEEE Pervasive Computing* 17, 3 (July 2018), 83–89. https://doi.org/10.1109/MPRV.2018.03367740

[33] Jong Hwan Ko, Taesik Na, Mohammad Faisal Amir, and Saibal Mukhopadhyay. 2018. Edge-host partitioning of deep neural networks with feature space encoding for resource-constrained internet-of-things platforms. In *Proceedings of the 2018 15th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS'18)*. 1–6. https://doi.org/10.1109/AVSS.2018.8639121

[34] Nicholas D. Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, Lei Jiao, Lorena Qendro, and Fahim Kawsar. 2016. DeepX: A software accelerator for low-power deep learning inference on mobile devices. In *Proceedings of the 15th International Conference on Information Processing in Sensor Networks (IPSN'16)*. IEEE Press, Article 23, 12 pages.

[35] Nicholas D. Lane, Emiliano Miluzzo, Hong Lu, Daniel Peebles, Tanzeem Choudhury, and Andrew T. Campbell. 2010. A survey of mobile phone sensing. *IEEE Communications Magazine* 48, 9 (September 2010), 140–150. https://doi.org/10.1109/MCOM.2010.5560598

[36] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2016. Pruning filters for efficient convNets. https://arxiv.org/abs/1608.08710

[37] Dawei Liang and Edison Thomaz. 2019. Audio-based activities of daily living (ADL) recognition with large-scale acoustic embeddings from online videos. *Proceedings of the ACM Interactive, Mobile, Wearable and Ubiquitous Technologies* 3, 1 (March 2019), Article 17, 18 pages. https://doi.org/10.1145/3314404

[38] Sicong Liu, Yingyan Lin, Zimu Zhou, Kaiming Nan, Hui Liu, and Junzhao Du. 2018. On-demand deep model compression for mobile devices: A usage-driven model selection framework. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'18)*. Association for Computing Machinery, New York, NY, 389–400. https://doi.org/10.1145/3210240.3210337

[39] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. 2018. Rethinking the value of network pruning. https://arxiv.org/abs/1810.05270

[40] Akhil Mathur, Anton Isopoussu, Fahim Kawsar, Nadia Berthouze, and Nicholas D. Lane. 2019. Mic2Mic: Using cycle-consistent generative adversarial networks to overcome microphone variability in speech systems. In *Proceedings of the 18th International Conference on Information Processing in Sensor Networks (IPSN'19)*. Association for Computing Machinery, New York, NY, 169–180. https://doi.org/10.1145/3302506.3310398

[41] Akhil Mathur, Nicholas D. Lane, Sourav Bhattacharya, Aidan Boran, Claudio Forlivesi, and Fahim Kawsar. 2017. Deepeye: Resource efficient local execution of multiple deep vision models using wearable commodity hardware. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. 68–81.

[42] Akhil Mathur, Tianlin Zhang, Sourav Bhattacharya, Petar Veličković, Leonid Joffe, Nicholas D. Lane, Fahim Kawsar, and Pietro Lió. 2018. Using deep data augmentation training to address software and hardware heterogeneities in wearable and smartphone sensing devices. In *Proceedings of the 17th ACM/IEEE International Conference on Information Processing in Sensor Networks*. IEEE Press, 200–211.

[43] Chulhong Min, Akhil Mathur, Alessandro Montanari, and Fahim Kawsar. 2019. An early characterisation of wearing variability on motion signals for wearables. In *Proceedings of the 23rd International Symposium on Wearable Computers (ISWC'19)*. ACM, New York, NY, 166–168. https://doi.org/10.1145/3341163.3347716

[44] Chulhong Min, Akhil Mathur, Alessandro Montanari, and Fahim Kawsar. 2022. SensiX: A system for best-effort inference of machine learning models in multi-device environments. *IEEE Transactions on Mobile Computing* 22, 9 (2022), 5525–5538. https://doi.org/10.1109/TMC.2022.3173914

[45] Chulhong Min, Alessandro Montanari, Akhil Mathur, and Fahim Kawsar. 2019. A closer look at quality-aware runtime assessment of sensing models in multi-device environments. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems (SenSys'19)*. ACM, New York, NY, 271–284. https://doi.org/10.1145/3356250.3360043

[46] Akshay Naresh Modi, Chiu Yuen Koo, Chuan Yu Foo, Clemens Mewald, Denis M. Baylor, Eric Breck, Heng-Tze Cheng, Jarek Wilkiewicz, Levent Koc, Lukasz Lew, Martin A. Zinkevich, Martin Wicke, Mustafa Ispir, Neoklis Polyzotis, Noah Fiedel, Salem Elie Haykal, Steven Whang, Sudip Roy, Sukriti Ramesh, Vihan Jain, Xin Zhang, and Zakaria Haque. 2017. TFX: A TensorFlow-based production-scale machine learning platform. In *Proceedings of KDD 2017*.

[47] Alessandro Montanari, Mohammed Alloulah, and Fahim Kawsar. 2019. Degradable inference for energy autonomous vision applications. In *Adjunct Proceedings of the 2019 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2019 ACM International Symposium on Wearable Computers*. 592–597.

[48] Alessandro Montanari, Fredrika Kringberg, Alice Valentini, Cecilia Mascolo, and Amanda Prorok. 2018. Surveying areas in developing regions through context aware drone mobility. In *Proceedings of the 4th ACM Workshop on Micro Aerial Vehicle Networks, Systems, and Applications*. 27–32.

[49] Alessandro Montanari, Afra Mashhadi, Akhil Mathur, and Fahim Kawsar. 2016. Understanding the privacy design space for personal connected objects. In *Proceedings of the 30th International BCS Human Computer Interaction Conference 30*. 1–13.

[50] Alessandro Montanari, Manuja Sharma, Dainius Jenkus, Mohammed Alloulah, Lorena Qendro, and Fahim Kawsar. 2020. ePerceptive: Energy reactive embedded intelligence for batteryless sensors. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems (SenSys'20)*. Association for Computing Machinery, New York, NY, 382–394. https://doi.org/10.1145/3384419.3430782

[51] Arthur Moss, Hyunjong Lee, Lei Xun, Chulhong Min, Fahim Kawsar, and Alessandro Montanari. 2022. Ultra-low power DNN accelerators for IoT: Resource characterization of the MAX78000. In *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems (SenSys'22)*, 934–940.

[52] Deepak Narayanan, Keshav Santhanam, Amar Phanishayee, and Matei Zaharia. 2018. Accelerating deep learning workloads through efficient multi-model execution. In *NeurIPS Workshop on Systems for Machine Learning*. 20.

[53] Francisco Javier Ordóñez and Daniel Roggen. 2016. Deep convolutional and LSTM recurrent neural networks for multimodal wearable activity recognition. *Sensors* 16, 1 (2016). https://doi.org/10.3390/s16010115

[54] Andrei Paleyes, Raoul-Gabriel Urma, and Neil D. Lawrence. 2020. Challenges in deploying machine learning: A survey of case studies. *CoRR* abs/2011.09926 (2020). arXiv:2011.09926. https://arxiv.org/abs/2011.09926

[55] Philipp Raith and Schahram Dustdar. 2021. Edge intelligence as a service. In *Proceedings of the 2021 IEEE International Conference on Services Computing (SCC'21)*. 252–262. https://doi.org/10.1109/SCC53864.2021.00038

[56] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. 2016. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA'16)*. IEEE Press, 267–278. https://doi.org/10.1109/ISCA.2016.32

[57] Joseph Redmon and Ali Farhadi. 2018. YOLOv3: An incremental improvement. https://arxiv.org/abs/1804.02767

[58] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2019. MobileNetV2: Inverted residuals and linear bottlenecks. https://arxiv.org/abs/1801.04381

[59] Mahadev Satyanarayanan. 2017. The emergence of edge computing. *Computer* 50, 1 (2017), 30–39. https://doi.org/10.1109/MC.2017.9

[60] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. 2015. Hidden technical debt in machine learning systems. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'15)*. MIT Press, Cambridge, MA, 2503–2511.

[61] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. 2015. Rethinking the Inception Architecture for Computer Vision. (2015). https://doi.org/10.48550/ARXIV.1512.00567

[62] Surat Teerapittayanon, Bradley McDanel, and H. T. Kung. 2017. Distributed deep neural networks over the cloud, the edge and end devices. In *Proceedings of the 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS'17)*. 328–339. https://doi.org/10.1109/ICDCS.2017.226

[63] Juheon Yi, Chulhong Min, and Fahim Kawsar. 2021. Vision paper: Towards software-defined video analytics with cross-camera collaboration. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems (SenSys'21)*. Association for Computing Machinery, New York, NY, 474–477. https://doi.org/10.1145/3485730.3493453

[64] Michael Zhu and Suyog Gupta. 2017. To prune, or not to prune: Exploring the efficacy of pruning for model compression. https://arxiv.org/abs/1710.01878

Received 1 April 2022; revised 28 July 2023; accepted 29 July 2023