electronics

MDPI

Article

Flexible Deployment of Machine Learning Inference Pipelines in the Cloud–Edge–IoT Continuum

Karolina Bogacka [1,2], Piotr Sowiński [1,2], Anastasiya Danilenka [1,2], Francisco Mahedero Biot [3], Katarzyna Wasielewska-Michniewska [1], Maria Ganzha [1,2,*], Marcin Paprzycki [1] and Carlos E. Palau [3]

1 Systems Research Institute, Polish Academy of Sciences, ul. Newelska 6, 01-447 Warsaw, Poland; karolina.bogacka@ibspan.waw.pl (K.B.); piotr.sowinski@ibspan.waw.pl (P.S.); danastas@ibspan.waw.pl (A.D.); katarzyna.wasielewska@ibspan.waw.pl (K.W.-M.); marcin.paprzycki@ibspan.waw.pl (M.P.)
2 Faculty of Mathematics and Information Science, Warsaw University of Technology, ul. Koszykowa 75, 00-662 Warsaw, Poland
3 Communications Department, Universitat Politècnica de València, Camí de Vera, s/n, 46022 Valencia, Spain; framabio@teleco.upv.es (F.M.B.); cpalau@dcom.upv.es (C.E.P.)
* Correspondence: maria.ganzha@ibspan.waw.pl

Abstract: Currently, deploying machine learning workloads in the Cloud–Edge–IoT continuum is challenging due to the wide variety of available hardware platforms, stringent performance requirements, and the heterogeneity of the workloads themselves. To alleviate this, a novel, flexible approach for machine learning inference is introduced, which is suitable for deployment in diverse environments—including edge devices. The proposed solution has a modular design and is compatible with a wide range of user-defined machine learning pipelines. To improve energy efficiency and scalability, a high-performance communication protocol for inference is propounded, along with a scale-out mechanism based on a load balancer. The inference service plugs into the ASSIST-IoT reference architecture, thus taking advantage of its other components. The solution was evaluated in two scenarios closely emulating real-life use cases, with demanding workloads and requirements constituting several different deployment scenarios. The results from the evaluation show that the proposed software meets the high throughput and low latency of inference requirements of the use cases while effectively adapting to the available hardware. The code and documentation, in addition to the data used in the evaluation, were open-sourced to foster adoption of the solution.

Keywords: machine learning; edge computing; IoT; cloud–edge–IoT; inference; gRPC; inference server

Citation: Bogacka , K.; Sowiński, P.; Danilenka, A.; Biot, F.M.; Wasielewska-Michniewska, K.; Ganzha, M.; Paprzycki, M.; Palau, C.E. Flexible Deployment of Machine Learning Inference Pipelines in the Cloud–Edge–IoT Continuum. *Electronics* **2024**, *13*, 1888. https://doi.org/10.3390/electronics13101888

Academic Editors: Simeone Marino and Palden Lama

Received: 26 February 2024
Revised: 15 April 2024
Accepted: 7 May 2024
Published: 11 May 2024

1. Introduction

The emerging popularity of machine learning (ML) solutions in recent years has led to a sharp increase in the number of industry deployments. Many ML pipelines that could previously be integrated only into environments with vast computational resources, most often cloud-based, have been further expanded to include edge devices or even the Internet of Things (IoT) [1], i.e., the Cloud–Edge–IoT continuum. In the Cloud–Edge–IoT continuum, processing and storage tasks are performed on all levels of the network hierarchy and not just in the cloud [2]. Such deployments often face similar challenges, many of them purely related to the infrastructure: its energy consumption, scalability, and the lack of user-friendly tools [3]. Therefore, many require similar solutions, which for ML often come in the form of general purpose ML inference servers.

Here, an ML inference *server* is understood as an application that, upon receiving a request containing data intended as input for inference, uses an ML model to obtain predictions, which it returns in the form of a response. ML inference servers can function as standalone deployments without additional software infrastructure, thus making them more suitable for edge environments. As a consequence, numerous approaches to ML inference servers have been introduced by the industry.

In contrast, recent academic works tend to focus on developing elaborate infrastructures dedicated to model serving in the form of machine learning inference *systems* [4–6]. These systems are not limited to a single application but instead leverage existing containerization and orchestration tools to support and manage the deployment of complex architectures. They are often integrated with well-known platforms such as Docker [7] and Kubernetes [8]. As a result, it is common for ML inference pipelines inside these systems to be implemented as chains or graphs of networked services, in which each of the ML models, as well as preprocessing and preprocessing steps, is encapsulated as a separate container. In the scope of this work, we assume an ML inference pipeline to be a series of complex data processing steps, which typically have to be deployed alongside the ML model [9] and with a particular focus on providing ML inference services.

There are many advantages of the inference system approach, especially with the growing length of ML pipelines and increasing model complexity in modern deployments [4]. First of all, it allows the pipeline to be easily modifiable. Any changes to the selection or ordering of data processing steps do not necessitate the redeployment of the whole pipeline. Instead, its composition can often be altered with just a few commands provided by the integrated orchestration tools (for example, Kubernetes). Those commands are usually well-documented and known to the maintainers of the infrastructure, which lowers the barrier to adoption [10,11]. Only when a completely new data processing step is added does it warrant additional effort from a software developer. As such, many of the changes to the pipeline can be made very quickly. In addition, most of the data processing code has been separated into independent components by design, which promotes reusability. Already developed components can be easily repurposed into other deployments and pipelines. Because of that, this approach is inherently extensible and flexible, as it can accommodate complex pipelines involving multiple ML models as ensembles, as well as multiple preprocessing steps. Furthermore, the encapsulation of data processing steps as standalone applications simplifies the development of custom components. Instead of ensuring that new code is well-integrated within a pre-existing server (a solution that would be harder to debug and deploy), the user only has to worry about providing a well-performing component. Overall, the concept of flexible ML inference pipelines enables rapid, iterative experimentation with the deployed ML workflow. This focus on experimentation may be especially beneficial for projects involving applied ML research tested in pilot conditions, as it allows many variants of the workflow to be easily created and tested. The users can adapt the pipeline to the real-life environment by adding or removing custom preprocessing and postprocessing steps, testing multiple versions of a given module, and changing the behavior of existing transformations through parameter modification, all without the need to rebuild the application as a whole.

However, the design philosophy of treating each step in the pipeline as a separate container also introduces a certain performance overhead. Even though the employment of Kubernetes orchestration in edge environments is not only possible but also increasingly popular [12], limited resources still necessitate a different architectural approach in edge-friendly solutions. Firstly, although the encapsulation of applications inside of, for instance, Docker containers, has only limited influence on the overall CPU usage, it is a noticeable influence nonetheless [13]. This influence is then multiplied with each preprocessing and postprocessing step, thus limiting the length of the pipeline. Secondly, dividing preprocessing steps that use the same modules between different containers, which run separate processes, means that the containers have to reserve RAM or GPU memory for the same modules multiple times. This is a factor that influences the overall usage of those resources [7]. Finally, and perhaps most importantly, this architecture causes the need for continuous serialization and deserialization of data when passed from one preprocessing step to the next. The consistent inclusion of such a transformation between each step diminishes the effectiveness of the overall solutions. These drawbacks are not as relevant for deployments located on multiple powerful devices as for those involving only a singular machine with limited communicational and computational capabilities. As such, providing

dynamic pipelines by dividing discrete steps between separate containers fits better with the characteristics of cloud than edge environments.

Related to our research work involving the testing of ML models in real-life environments, there was an interest in a solution that would facilitate easy experimentation with ML inference pipelines by providing the flexibility, extensibility, and reusability of ML inference systems. It should be stressed that this solution would have to be deployed in a very resource-constrained edge environment. The main challenge of this paper therefore lies in providing the aforementioned features while facing stringent environmental limitations. In this work, an approach designed to address this problem and developed within the scope of the ASSIST-IoT Horizon 2020 project [14] is presented. Namely, the Modular Inference Server (MIS), a flexible solution for deploying ML inference pipelines in a wide variety of computational settings, is proposed along with the complementary Component Repository. The Component Repository supports managing and persisting components used to build ML inference pipelines.

The proposed system was designed with two real-life use cases in mind, which differ greatly in terms of used ML models and hardware requirements. Thus, a large focus was placed on obtaining an ML inference server that would be general enough to fulfill the needs of both scenarios. Special attention was put on ensuring compatibility with a variety of hardware platforms present in the use cases. Consequently, the persistent storage of pipeline components was recognized as a useful addition, thus improving their overall scalability and reusability. Finally, the resulting framework still had to offer fast communication and low resource consumption or otherwise risk losing compliance with the stringent performance requirements of the use cases. As an ML inference server, the MIS achieves these goals by prioritizing a simpler architecture than that of ML inference systems and by not requiring any data transfer between server instances. As a consequence, the MIS does not offer support for distributing its pipelines across multiple compute nodes.

In summary, the main contributions of this work consist of (1) an extensive comparison of freely available solutions for ML inference serving (including ML inference servers and ML inference systems); (2) a novel solution for integrating flexible and reusable ML inference pipelines into an ML inference server; (3) examples of two different ML inference pipelines designed for two real-life use cases and integrated into the MIS; (4) the description and results analysis of experiments testing the performance of the solution in multiple scenarios motivated by real-life use cases. Our approach for integrating ML inference pipelines focuses on providing a combination of wide hardware support, lightweight communication, and scalability, which we were unable to find in existing works.

The code used to implement the MIS and the Component Repository, as well as to prepare the presented ML use cases and conduct the benchmarks, has been made public under the Apache 2.0 license. It is available on GitHub (as inference-server (https://github.com/Modular-ML-inference/inference-server), component-repository (https://github.com/Modular-ML-inference/component-repository), ml-usecase (https://github.com/Modular-ML-inference/ml-usecase), and benchmark-driver (https://github.com/Modular-ML-inference/benchmark-driver); accessed on 3 April 2024) and Zenodo [15–18].

This paper is organized as follows. Section 2 introduces the proposed approach to the problem of flexible inference pipelines in the Cloud–Edge–IoT continuum—the Modular Inference Server and the supplementary Component Repository. Section 3 provides relevant context for experiments with the MIS in the form of use case descriptions. Section 4 explains the setup of the experiments used to test the effectiveness of the solution. Section 5 contains an analysis of the results obtained from conducted experiments. In Section 6, the current state of the art is presented, with an emphasis given to the identified research gaps and how the MIS aims to address them. Here, the MIS is compared with other relevant approaches. Finally, Section 7 discusses potential ambiguities and areas requiring future work, and Section 8 formulates the final conclusions.

2. Proposed Solution

In order to fulfill the aforementioned requirements, the proposed solution prioritizes flexible pipeline design and low resource consumption. As depicted in Figure 1, it consists of two applications: the Modular Inference Server (MIS) and the Component Repository. The MIS, which can also function in standalone mode, provides the functionalities of an ML inference server. Based on the configuration supplied by the administrator, the MIS constructs the full ML inference pipeline—including a lightweight gRPC service, the ML model wrapped in an inferencer component, and a list of preprocessing and postprocessing functions. All of these components are pluggable—they can be easily replaced and chained. If any of the pluggable components specified in the configuration are not present on the local filesystem, the MIS can dynamically download them from the Component Repository. The Component Repository is responsible for storing the metadata and the serialized files of the pluggable components. It can be used to quickly distribute new functionalities across multiple MIS instances. The MIS, however, still serves as the foundation of the whole solution.

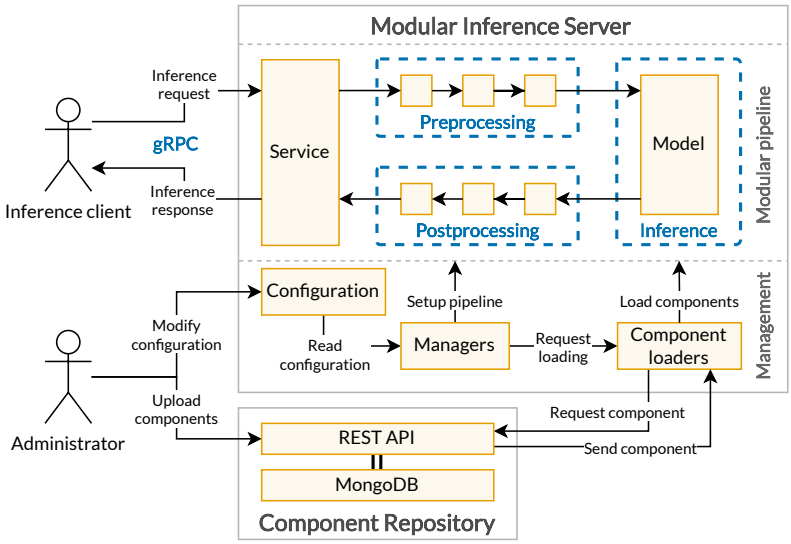

Figure 1. A high-level overview of the proposed solution emphasizing relevant user interactions with the system as a whole.

2.1. Modular Inference Server

The Modular Inference Server was originally developed as an inference component for the ASSIST-IoT Federated Learning (FL) Local Operations enabler, where it is a part of the larger ASSIST-IoT FL system [19]. The design of the MIS extends the idea of pluggable Data Transformations (first introduced in the FL system [20]) to a wider variety of pluggable component types. By encapsulating all processing steps in a single application, the MIS removes the need for interstep communication over the network, thereby positively contributing to the energy efficiency of the solution.

The MIS, after it builds the ML inference pipeline at the start of the application, functions as a gRPC server. The pluggable components can be used to dynamically reconfigure the pipeline. As shown in Figure 2, the MIS supports four categories of pluggable components: Data Transformation, Model, Inferencer, and Service. Data Transformations handle the preprocessing and postprocessing that are operations common in ML pipelines. Model components are essentially serialized ML models. Inferencers handle the initialization of Models and form an abstraction over how the Models perform predictions. Services encapsulate gRPC interfaces for the pipeline, along with their desired request and response formats. In general, the components can and should contain custom code, with the length of the preprocessing and postprocessing pipelines not being limited by the MIS.

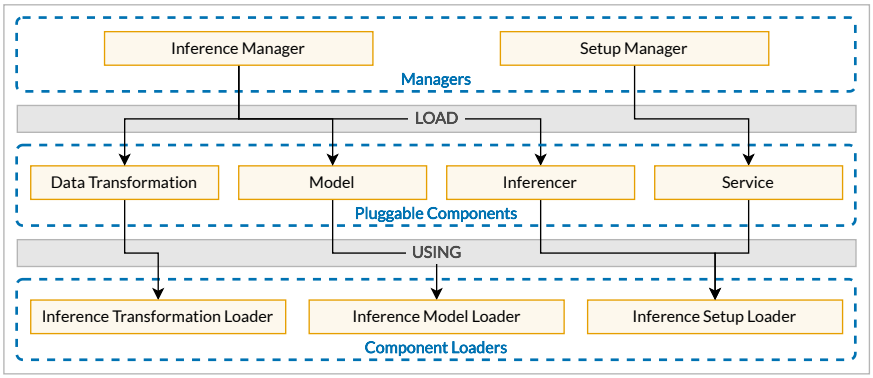

Figure 2. Pluggable components of the MIS and their interactions with the rest of the architecture.

Figure 3 presents the initialization of an ML inference pipeline, which is performed based on the application's configuration and specified in JSON files. The configuration defines the components to be loaded, their parameters, their order in the pipeline, and the data formats expected by the Service and the Model. This metadata allows the MIS to check the correctness of the ML inference pipeline by comparing the input data format passed through the pipeline with the format expected by the Model.

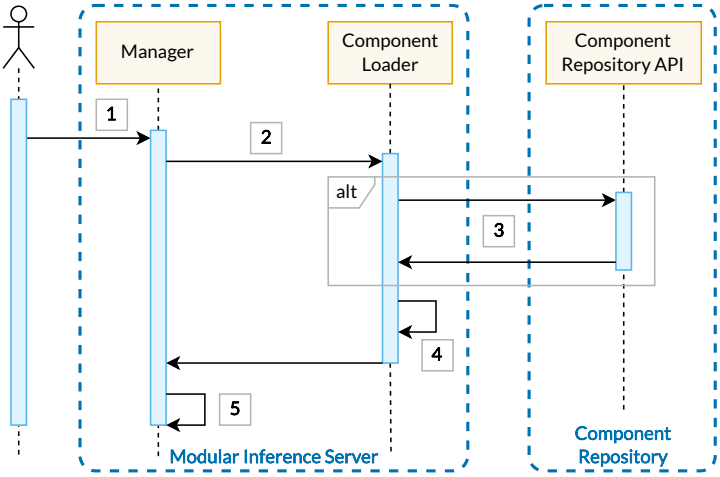

Figure 3. Sequence diagram illustrating the process of loading and initializing the pluggable components. (1) The administrator modifies the configuration and restarts the application. (2) The Manager checks which component needs to be loaded and calls the appropriate Component Loader to do it. (3) The Component Loader checks if the component is in the local file system. If not, it sends a request to the Component Repository to download the component. (4) If the downloaded component is a model, the Component Loader unpacks the file and uses a dedicated inferencer method to load it. Otherwise, the Component Loader decompresses and initializes the component. (5) The Manager connects the component to other elements in the pipeline. If the inference pipeline is ready, it checks whether the input/output data formats are consistent.

The system administrator can choose to either include the components in the MIS's file system or upload them to the Component Repository. The former allows the MIS to operate in standalone mode, thus simplifying the deployment and lowering overall hardware requirements. On the other hand, with the Component Repository, multiple MIS instances can download components dynamically, thereby increasing the scalability of the solution.

A decision was made for the MIS to utilize gRPC for inference communication, including bidirectional gRPC streaming for its default interface. gRPC was selected due to the very good throughput and latency characteristics offered by this modern, open-source remote procedure call framework [21,22]. gRPC uses HTTP/2 as the underlying protocol

to create robust and long-lived connections with advanced mechanisms such as keep-alive messages, flow control, and multiplexing [23,24]. Bidirectional streaming allows the MIS to asynchronously stream requests in and responses out, thereby minimizing the latency. Additionally, as gRPC streaming combines the data compression offered by Protocol Buffers with multiplexing, reducing the overhead necessary to communicate large amounts of data when compared with protocols such as REST, its inclusion is expected to result in a significantly more lightweight communication with little impact on the computational resources [21,25]. Due to the notable influence of telecommunication networks on the energy consumption of the ICT ecosystem, this design choice is expected to result in the improved energy efficiency of the overall solution. It should also be stressed that, in contrast to ML inference systems, the MIS does not perform any internal communication over the network, thereby removing the need for repeated serialization and deserialization and further reducing energy use.

The dedicated Service, which is provided out of the box with the MIS, was designed to be as efficient and reusable as possible. Both requests and responses encode the data as a map with values in an established format (`tensorflow.TensorProto`). This format is commonly used by TensorFlow [26], as well as TensorFlow Serving [27], and can effectively encode multidimensional arrays of various types (integer, string, Boolean, float, etc.) while preserving the information about its original shape. This interface allows for very diverse data types to be effectively communicated in and out of the MIS, including entire batches of data in a single request.

The MIS is publicly available under the Apache 2.0 license on GitHub (https://github.com/Modular-ML-inference/inference-server) (accessed on 3 April 2024) and Zenodo [15]. More information on the creation of custom pluggable components can be found in the repository's README file. It contains extensive instructions on how to construct ML inference pipelines, as well as directions on how to develop, serialize, and integrate the components into the MIS based on Model and Data Transformation examples. The repository also includes components prepared for the use cases, along with the configuration used to deploy them.

2.2. Component Repository

The Component Repository (called the FL Repository during its development in the ASSIST-IoT project) consists of a lean FastAPI [28] application that handles the modification of the underlying MongoDB [29] database instance. It allows the administrator to easily create and update the metadata of pluggable components, upload their serialized files, download the metadata, download the files, delete the metadata and the files, and display a list of all currently available components in a given category. All of those functionalities are implemented in a RESTful API.

The Component Repository handles every class of pluggable components used by the Modular Inference Server: Data Transformations, Models, Inferencers, and Services. The metadata of these components includes their description, software, and hardware requirements, as well as accepted input parameters and their default values. The serialized components themselves are stored using the GridFS functionality of MongoDB, which is offered for files that may exceed the limit of 16 MB. MongoDB also supports robust replication, thereby allowing for large deployments of the Component Repository if needed. The Component Repository is available under the Apache 2.0 license on GitHub (https://github.com/Modular-ML-inference/component-repository) (accessed on 3 April 2024) and Zenodo [16], along with usage instructions and API documentation.

2.3. Deployment and Integration

In order to make the MIS and the Component Repository deployable in containerized environments, they were integrated with the Kubernetes orchestration platform, including the selected tools commonly combined with it: Helm [30] and Prometheus [31]. Helm serves as a package manager for Kubernetes applications—thus allowing the developer

to easily define, install, and upgrade them—whereas Prometheus provides a dedicated monitoring solution. Subsequently, a Helm chart for the Modular Inference Server was created. The chart exposes the gRPC inference service and HTTP endpoint broadcasting metrics in the Prometheus format. Through the efficient use of Kubernetes Persistent Volumes and ConfigMaps, the chart empowers the system administrator with the ability to easily modify the application on the fly. Deploying multiple instances of the MIS is supported in order to allow for running multiple ML inference pipelines in parallel or to replicate a single pipeline. Additionally, the MIS can be combined with a dedicated load balancer (e.g., Envoy) to spread the requests over multiple MIS replicas, thus allowing the system to handle more clients. Docker images for the MIS were built for the x86-64 and ARM64 processor architectures to enable deployment on a wide range of cloud and edge devices. For the Component Repository, a separate Helm chart was created. The chart includes an easily modifiable ConfigMap, which describes the configuration of the application.

The MIS was designed as a machine learning inference server and can be effectively deployed in standalone mode as a single container (even without Kubernetes), which is a feature deemed important due to the inclusion of edge environments in pilot use cases. Because of that, its Kubernetes configuration was kept straightforward, with the only additional component being the supplementary Component Repository. As such, it is worthwile to explore in depth the influence of the use cases on the design and deployment of the MIS.

3. Use Cases

This section describes the use cases and machine learning inference pipelines employed to test the Modular Inference Server. These real-life scenarios were derived from the pilot implementations of the ASSIST-IoT project and serve to validate the performance of the MIS with realistic workloads and requirements. They were selected to be as diverse as possible, thus representing very different workload types.

3.1. Worker Safety: Fall Detection

The first use case, based on the ASSIST-IoT Smart Safety of Workers pilot, involves detecting the falls of construction workers in real time. As described in detail in a recent work [32], the construction workers were equipped with acceleration sensors, which reported their readings to an edge device nearby with a sampling frequency of 2 Hz. The edge device then assembled an acceleration time series and fed it as input to a neural network twice every second. The output of the network indicated the confidence that the worker had fallen during the span of the time series. This information was then used to alert the responsible persons about a possible accident.

Implementing this use case in practice comes with specific requirements, which stem directly from the nature of the task. Mainly, the latency of detecting falls must be minimized to provide aid to the potentially injured worker as quickly as possible. The MIS and its interfaces have a major influence on this latency. Ideally, the time it takes for a request to be sent, processed, and returned should always be less than 100 ms, with a mean latency around 10 ms. Therefore, low latency here is understood as the latency that consistently meets these requirements. The connection to the inference service must also be reliable, so as to detect all falls that occurred. The ARM64-based edge device developed in the ASSIST-IoT project that the inference will run on, the Gateway Edge Node [33], has very limited processing capabilities. This fact influenced the design of the ML pipeline, which aims to be as lightweight as possible.

The fall detection model must be able to run on the resource-constrained GWEN, all while scaling to a high number of workers. In the field tests conducted within the ASSIST-IoT project, scaling to up to 10 workers was tested with the GWEN, as this was the number of available acceleration sensors. However, the system should support more workers per edge device, as many as 40–60. The expected number of inferences per second

can be then simply expressed as $2n$, where n is the number of workers currently served by the edge device, assuming the aforementioned 2 Hz sampling rate.

To implement this use case, the final model from a recent work was used [32], with 203 parameters in total. Figure 4 presents the Modular Inference Server pipeline implemented for this task.

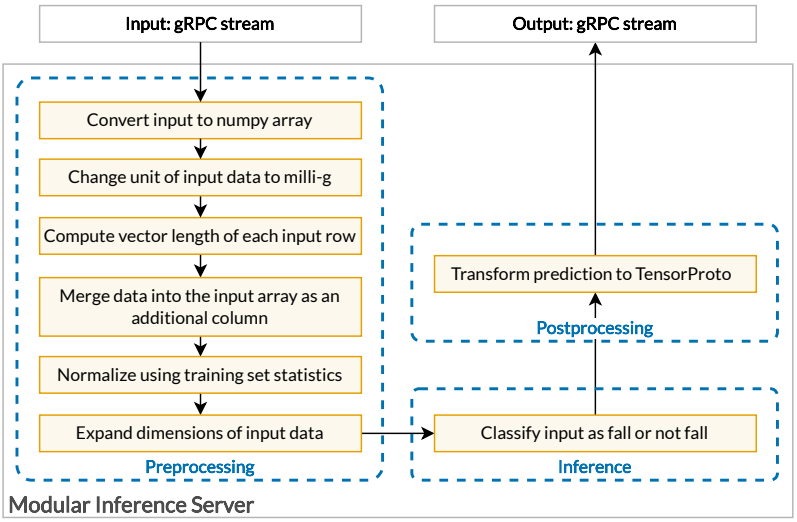

Figure 4. Inference pipeline for the fall detection use case.

The first preprocessing step transforms the input received over gRPC into the format of a numpy [34] array, which is often used for numerical operations in Python. Then, as the unit in which acceleration is measured by the tags differs from the milli-g (where g stands for the universal gravitation constant) employed by the model, it is converted in the second preprocessing step. Subsequently, a new feature is created based on the input values by calculating the length of the vector formed by the three acceleration axes supplied by the client. The result is added as a new column to the input data. Later, the obtained array is normalized with the help of statistics calculated on the training dataset to ensure that the scale of the input data follows the scale of the data the model was trained on. Afterwards, the dimension of the input data is expanded to form a batch of one input sequence for model inference. In the inference stage, the model processes the input data and assigns the probability of this input sequence being a fall in the range [0–1]. Finally, the prediction is encoded as a map of `tensorflow.TensorProto` and transmitted back, along with the original identifier of the message. The size of the output messages is both considerably smaller and more deterministic than in the next use case focusing on scratch detection.

3.2. Car Inspection: Scratch Detection

The car inspection use case is a part of the ASSIST-IoT Cohesive Vehicle Monitoring and Diagnostics Pilot. Here, the goal is to detect scratches on images of cars taken with specialized vehicle scanners [19]. The scanners can be placed, for example, in rent-a-car companies or in maintenance garages. The scanners take numerous photos of the vehicles passing by, thereby allowing for further machine learning-based automated damage inspection. Each scanner is equipped with multiple cameras to provide images from different points of view, thus resulting in each passing vehicle having 50–300 images taken [19] with more than 200 vehicles passing through the scanner daily. This effectively necessitates the machine learning inference system to have a high throughput so as to provide the users with scratch detection results as quickly as possible. The use case-defined preferred inference time per vehicle lies in the range of 2 to 5 min and depends on the business processes linked to the damage recognition. For example, a vehicle inspector at a rent-a-car company may require a faster processing time than a maintenance garage. The use case focused on two damage types: scratches and rim damage. For this work, only the scratch

damage type was chosen for experiments. To facilitate the necessary image processing and accelerate ML inference, each scanner was connected to an edge device equipped with a GPU.

In the use case, the Mask R-CNN model was used [35], which is a popular solution for image segmentation tasks. The Mask R-CNN model consists of the following parts: the backbone network that processes the image and creates feature maps; the region proposal network that identifies candidate objects; RoI (region of interest) pooling for extracting the features from candidate objects; classification and bounding box regression heads for candidate objects classification and localization; and mask segmentation branch responsible for generating pixel masks for each detected object. Overall, the model has 45,880,411 parameters and consumes 180 MB of disk space, thus making it a lot more resource-demanding than the previously presented fall detection use case.

For each input image, the model produces a prediction in a dictionary format, which is specified by the PyTorch [36] Mask R-CNN model builder documentation [37], where the length of dictionary entries represents the number of detected instances (scratches):

- Boxes: The list of bounding boxes, where each box is described with a list of coordinates [x1, y1, x2, y2].
- Labels: The list of predicted classes. In the presented use case, the values in the list are always equal to "1" due to there being only one class of damages in the task.
- Scores: The list of prediction confidences with values in the [0–1] range.
- Masks: The list of 2-dimensional masks with values in the [0–1] range.

What should be noted here is that the size of the prediction depends on the number of detected scratches in the image. This effectively means that when deployed in the MIS, the inference responses will be larger for images containing scratches.

The model used in the experiments was trained on a manually labeled, representative training dataset obtained from real vehicle scanners. Due to privacy and licensing concerns, neither the training dataset nor the trained model can be made public. However, the experiments presented in this work can still be partially reproduced. Similar results can be obtained by using the raw Mask R-CNN model, with appropriate model architecture modifications. The relevant source code and instructions for how to achieve this were published under the Apache 2.0 license on GitHub (https://github.com/Modular-ML-inference/ml-usecase) (accessed on 3 April 2024) and Zenodo [17].

The model operates in the MIS pipeline presented in Figure 5. The first preprocessing step transforms the input data received over gRPC into a numpy array. Then, as the data passed into the model can be a batch of images of any length, that numpy array is decomposed into a list of arrays (one for every image in the batch). Each RGB input image is represented by an array of dimensions (height, width, and channel). In the next step, the axes are transposed to shape the image to the format expected by the model (batch, channel, height, and width). The final step of preprocessing rescales each image into the range of [0–1], as expected by the model, and forms the input batch as a list of Torch tensors. The model performs inference on the input batch and returns a list of dictionaries as the prediction. Each dictionary in the list is in the format described above. The detected scratches are then filtered based on the reported confidence of the model, excluding the ones that the model is not confident in. For the presented use case, the confidence threshold was set to 0.5. The final list of detected scratches is transformed into a map of `tensorflow.TensorProto` to send it back over gRPC. The response includes concatenated information about the labels, boxes, scores, and masks found by the model for the entire batch of images passed into it. Alongside it, a list of integers is included, which encodes how many of those predictions were found for each image in the batch, thereby enabling the client to assign each prediction to a specific input image.

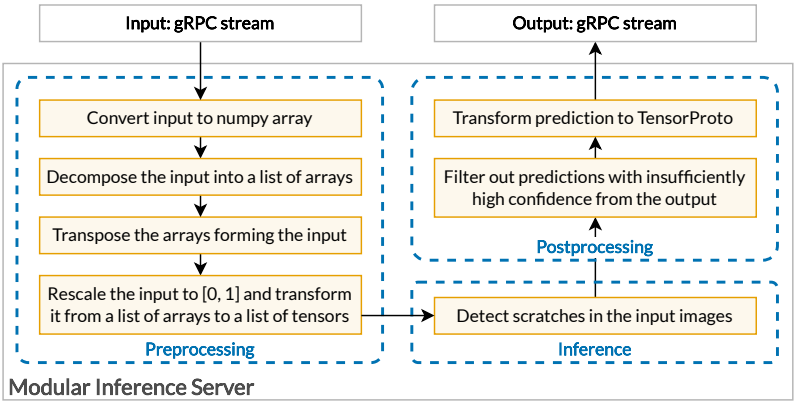

Figure 5. Inference pipeline for the scratch detection use case.

4. Experimental Setup

The setup for the experiments with the Modular Inference Server was designed to maximize the consistency of benchmark results while emulating the real deployment as much as possible. In all scenarios, the task of sending inference requests, analyzing inference results, and collecting various metrics was delegated to a separate machine: an x86-64 workstation. This separation allowed us to minimize the number of services that had to be run on the inference server, thus yielding cleaner benchmark results.

During the experiments, two different machines were used for hosting the Modular Inference Server. The first is the GWEN, the ARM64-based edge device developed as a part of the ASSIST-IoT project. The GWEN was designed to be power-efficient and thus has very limited processing capabilities. The second is an x86-64-based GPU server with ample computing resources. The detailed specifications of all machines used in the experiments can be found in Table 1. The machines were connected via Gigabit Ethernet, with a TP-Link TL-SG108 L2 switch, as illustrated in Figure 6.

Table 1. Specifications of machines used in the experiments.

Device	Arch.	CPU	Logical Cores	GPU	RAM	Linux Kernel
Client	x86-64	Intel Xeon Gold 5218	32	–	128 GB	5.4.0
GWEN	ARM64	i.MX 8M Plus Quad	4	–	4 GB	5.16.71
GPU server	x86-64	AMD Ryzen 5 3600	12	Nvidia RTX 3090 Ti (24 GB)	64 GB	6.2.0

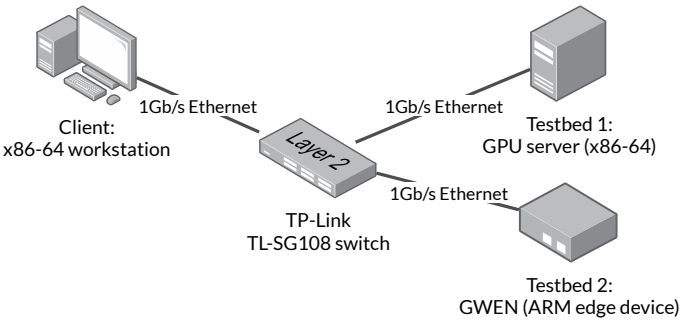

Figure 6. Network diagram of the test setup.

The inference requests were made by a dedicated test driver application written in Scala and running on the client machine. The test driver uses the Apache Pekko Streams library to reactively and reliably manage streaming gRPC requests to the Modular Inference Server. It takes accurate (sub-microsecond) measurements of request and response times using the system's monotonic clock. This allows for very precise calculation of the round-trip request–response latency in the experiments. The application was containerized for

easier use and published under the Apache 2.0 license on GitHub (https://github.com/Modular-ML-inference/benchmark-driver) (accessed on 3 April 2024) and Zenodo [18], along with usage instructions.

The client machine also hosted a Prometheus 2.48.1 server to collect metrics from two sources: the Modular Inference Server instances and the Prometheus Node exporter. The MIS exposed metrics related to request processing time and the status of the Python virtual machine. On both the GWEN and the GPU server, Prometheus Node exporter 1.7.0 was installed to expose metrics about the operating system and the hardware. Additionally, Nvidia DCGM exporter 3.3.0 was installed on the GPU server to expose metrics about the GPU.

In the experiments, a total of three different deployment scenarios were considered, as summarized in Figure 7. For the GWEN edge device, due to its limited computing resources, only one instance of the MIS was deployed in Docker without a load balancer (scenario A). For the GPU server, two deployment variants were used with Kubernetes (kubeadm 1.28.2). In scenario B, one instance of the Modular Inference Server was deployed without load balancing. In scenario C, a varying number of MIS instances were deployed in Kubernetes (1, 2, and 4), with a load balancer directing gRPC requests to them. The load balancer used in the tests was Envoy 1.18.3 in a round-robin configuration.

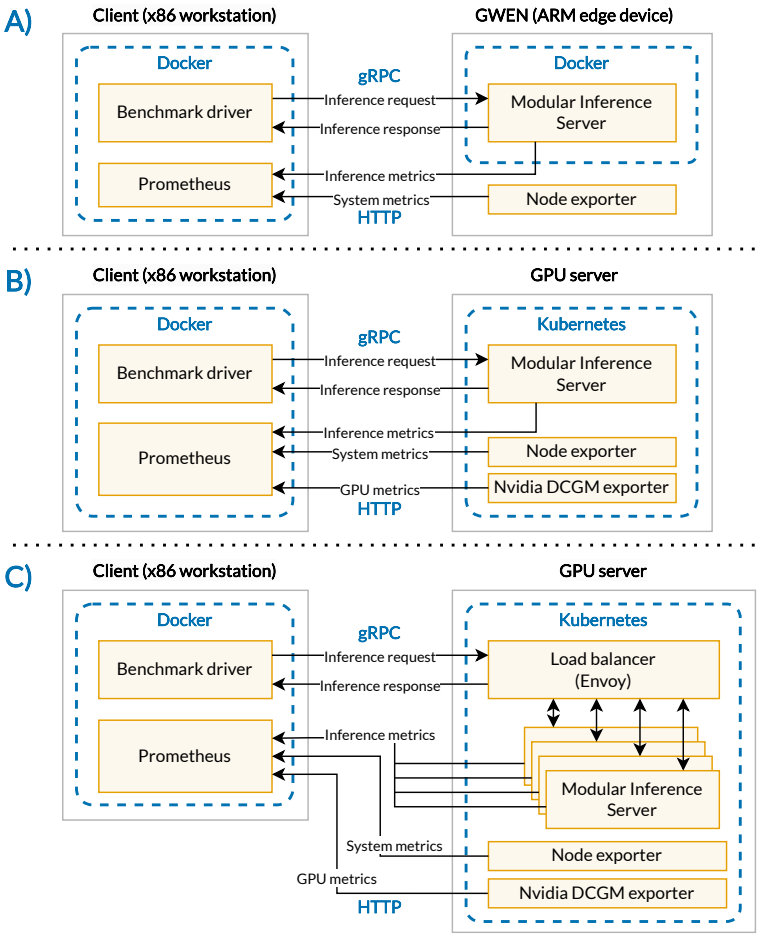

Figure 7. Component diagram of the benchmark setup. (**A**) Setup with the GWEN (only for the fall detection scenario). (**B**) Setup with the GPU server, without load balancing. (**C**) Setup with the GPU server, with load balancing and varying number of inference server pods.

The approach to the experiment design in this study was informed by existing works describing ML inference servers and ML inference systems [6,38–41], both in terms of metric selection, as well as the number and architecture of devices included in the setup.

The experiments focused on scenarios with a single compute node while testing different hardware and deployment strategies. Due to the MIS being an ML inference server, it does not rely on complex, multinode deployments to realize its pipelines. Instead, all stages of the pipeline are encapsulated within one compute node. The devices used in the experiments were chosen on the basis of use case requirements, thus mimicking the hardware used in real-life scenarios. At the same time, the used devices represent two of the most popular CPU architectures and possess different ML acceleration capabilities (GPU), thus demonstrating the portability of the MIS.

4.1. Fall Detection

For the fall detection use case, two host devices were tested: the GWEN and the x86-64 server. In both cases, only the CPU was used for inference, even though the server had a GPU installed. This is due to the very small size of the model used (203 parameters) and the need to maintain very low and consistent latency. With standard GPU accelerators, parallelization is only possible with batching or by time-sharing the GPU between multiple applications, both of which would incur additional latency. Therefore, the GPU was not used in these experiments.

The workload was simulated with real-world data collected from workers performing a range of activities on an active construction site during the trials of the ASSIST-IoT project. The dataset consists of 11 h of recorded acceleration patterns from a single accelerometer and is available publicly on GitHub (https://github.com/Modular-ML-inference/ml-usecase) (accessed on 3 April 2024) and Zenodo [17]. During experiments, the benchmark driver used it to simulate the load of a configurable number of devices. Several instances of the benchmark driver could be launched simultaneously to simulate multiple clients—such a situation would occur if several GWENs only collected acceleration data, while the inference was performed on a central, more powerful machine. Each simulated device generated data for 15 min with the use case-dictated 2 Hz frequency.

The following experiment variants were conducted. With the GWEN (setup A on Figure 7), only one client was tested, which corresponded to the real-life scenario. The one client simulated the load of 10, 20, 40, 80, or 160 devices. With the GPU server, both setup B and C were tested (without or with load balancing; see Figure 7). In setup C, the number of MIS pods was 1, 2, or 4. In all experiments with the server, the number of clients was 1, 4, or 16, and the number of devices per client was 10, 20, 40, 80, or 160.

4.2. Scratch Detection

Due to the aforementioned privacy and licensing restrictions, the real images from vehicle scanners could not be published. However, preserving the features of the real dataset is important for the experiments, as the number of detected scratches on each image has an impact on the total amount of data transferred during the inference. This is because for each detected scratch, additional data is returned (e.g., image masks), thus yielding larger response sizes. Hence, to precisely model the use case with regard to the volume of data returned from the pipeline, the distribution of the number of detected scratches must be kept.

In order to achieve this while keeping the experiments described in this work reproducible, the CarDD [42] dataset was used as a substitute instead of the real images from vehicle scanners. Specifically, first, the trained model performed inference on the real dataset, and for each inferenced image, the number of scratches reported by the model was recorded. This was used to estimate the probability distribution of the number of detected scratches per image (Figure 8). Subsequently, the same model was used for inference on the CarDD dataset, and the number of detected scratches was again recorded for each image, thus forming the second probability distribution. During the next step, this distribution was adjusted to match the real dataset's distribution by subsampling the images that reported a given number of detected scratches in the necessary proportions. This resulted in a subset of the CarDD dataset that has the same detected scratches

probability distribution as the evaluation dataset. This ensured that during the inference experiments performed in this paper, the distribution of response sizes from the MIS closely mimicked the one that would be obtained with the real dataset. The code needed to generate this subset, along with the list of used images from CarDD is available on GitHub (https://github.com/Modular-ML-inference/ml-usecase) (accessed on 3 April 2024) and Zenodo [17].

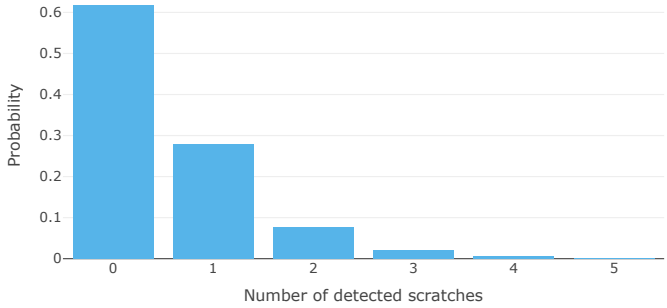

Figure 8. Probability distribution of the number of detected scratches per image in the evaluation dataset.

The workload was simulated with the benchmark driver application, where each client corresponded to one vehicle scanner. Every 3 min, the client generated a scan consisting of 50 to 300 images, with continuous uniform distribution of image count. Each image was 1200 pixels in width and 900 pixels in height with three color channels. The images were grouped into batches of one or more images and sent over gRPC to the MIS. The Modular Inference Server was deployed on the GPU server (benchmark setups B and C; see Figure 7), and the GPU was utilized as the inference device. In setup C, GPU time sharing was employed to allow multiple MIS instances to access the accelerator.

The following experiment variants were conducted. With both setup B and C (see Figure 7), the number of clients was 1, 2, or 4, while the batch size (number of images per gRPC request) was 1, 4, or 16. In setup C, the number of MIS pods was 1 or 2. In all experiments, the clients generated vehicle scans in the same manner as described above.

4.3. Experiment Summary

Table 2 summarizes the deployments used in the performed experiments. The two use cases were tested in several very different configurations, with the fall detection use case using exclusively CPU-based inference, while scratch detection resorted to GPU-based inference. In addition, Table 3 summarizes the pipelines deployed to the MIS in both use cases.

Table 2. Summary of Modular Inference Server deployments in the performed experiments.

Use Case	Testbed	ML Inference Device	Container Orchestration	Load Balancer	Instances
Fall detection	GWEN	ARM64 CPU	Docker	No	1
Fall detection	GPU server	x86-64 CPU	Kubernetes	No	1
Fall detection	GPU server	x86-64 CPU	Kubernetes	Yes	1, 2, 4
Scratch detection	GPU server	GPU	Kubernetes	No	1
Scratch detection	GPU server	GPU	Kubernetes	Yes	1, 2

Table 3. Summary of tested Modular Inference Server pipelines. The input tensors here refer to the tensors sent in the request over gRPC. In the table, b stands for the batch size.

Use Case	Model Type	Parameters	Input Tensor Type	Input Tensor Size
Fall detection	LSTM	203	int16	$(9, 3)$
Scratch detection	Mask R-CNN	45,880,411	uint8	$(b, 900, 1200, 3)$

The two use cases have very different deployment, pipeline, and workload characteristics. For fall detection, the requests are very small, very frequent, and arrive at a constant pace. For scratch detection, the requests are much larger and arrive in irregular bursts. The use case-specific performance requirements also differ—in fall detection, latency must be optimized, while for scratch detection, throughput is the most important aspect. Overall, the designed experiments present a diverse challenge for the MIS.

5. Results and Analysis

The following section describes the results of the experiments performed with the Modular Inference Server in the context of the two presented use cases.

5.1. Fall Detection Results

For the fall detection use case, the most important performance aspect is latency (the time from the client sending an inference request to it getting a response). Throughput is less of a concern, as long as the MIS is able to operate in near-real time, that is, without creating long queues of requests. Therefore, the following analysis focuses on end-to-end latency. Due to the large number of performed experiments, the detailed results were placed in Appendix A, while this section focuses only on the most important results.

For the resource-constrained GWEN, only one deployment variant was investigated: one MIS container in Docker, and one client with a varying number of client devices, which served as the data sources. Figure 9 illustrates the request–response latency distribution, thus measuring the time from the client sending an individual request in a gRPC stream to it getting the corresponding response. The plot does not include the experiment with 160 client devices, as the MIS did not manage to process that many requests in real time, which yielded very high latencies (see Appendix A). As can be seen in the figure, for 10, 20, and 40 devices, the most common latency was around 8 ms, with some requests taking less. For 40 and 80 devices, requests taking longer appeared more often. In all presented cases, the maximum latency did not exceed 62 ms, while the median ranged from 7.52 ms (40 devices) to 11.63 ms (80 devices).

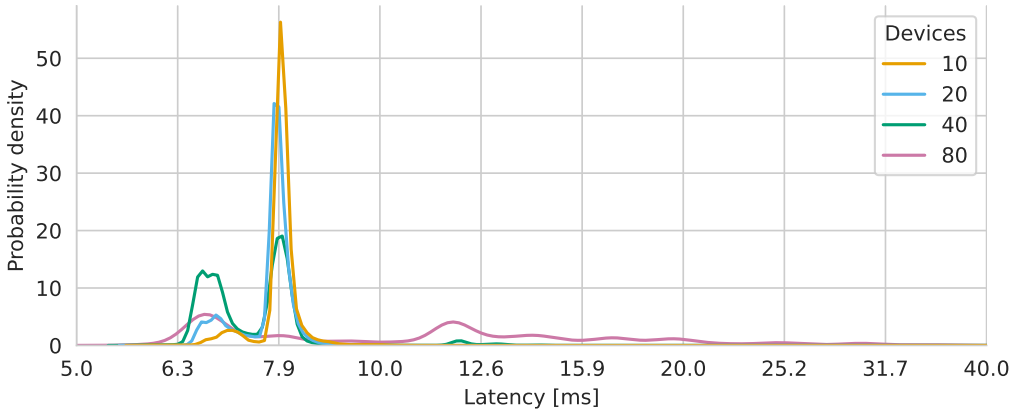

Figure 9. Inference request–reponse latency distribution for the deployment on the GWEN. The distribution is visualized using a kernel density estimate. The X axis is logarithmic.

Interestingly, for higher numbers of devices, the proportion of sub-8 ms requests is larger than with only 10 devices. The cause of this phenomenon has not been investigated in depth; however, it may be a consequence of saturating the streams on both the client and the server. When a stream processor has no elements to process, it is typically paused. For example, the process is removed from the CPU by the kernel scheduler, or, in Apache Pekko, the stream stage is removed from the actor thread to allow other workloads to take its place. In such a case, when a new stream element arrives later, the stream processor must be started again, which unavoidably introduces some latency. This can be circumvented if the stream processor always has more elements to process, which is a situation which occurs with streams of higher throughput.

Due to the nature of the gRPC protocol, the request and response streams are not synchronous. This effectively means that the client may send several messages in the request stream before it starts getting the corresponding responses from the server. In the context of this study, an inference request that was sent but not yet responded to is called an *in-flight request*—it can be thought of as queued for processing. Figure 10 presents the distribution of in-flight requests for the experiments on the GWEN. The in-flight count was measured and recorded every time a request was sent and a response was received. Hence, for 10 and 20 devices, the in-flight count was only 0 or 1, which corresponds to requests being sent and received serially in a synchronous manner. For 40 devices, situations with two in-flight requests occurred, while for 80 devices, the in-flight count reached up to 10 requests (not shown on the plot due to the very small bar size).

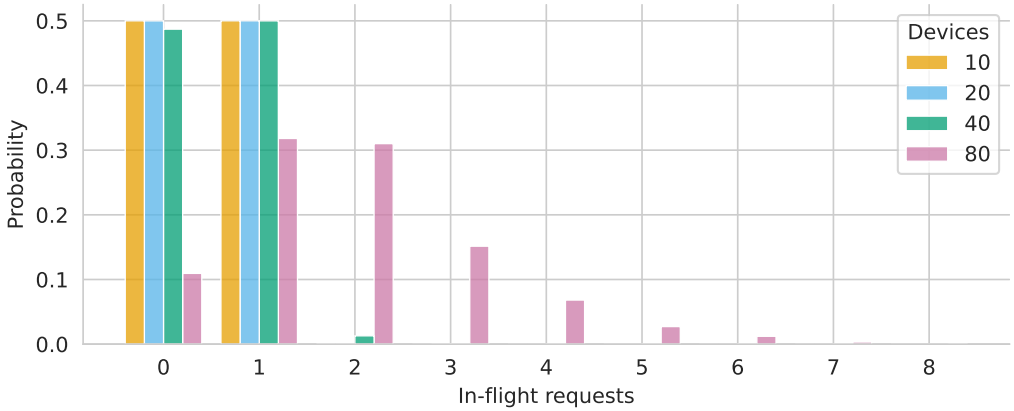

Figure 10. Probability distribution of in-flight fall inference requests for the deployment on the GWEN.

In experiments with the server, different deployment variants and numbers of clients were tried, thus yielding two more dimensions in the data to explore. Figure 11 presents the relation between the number of clients, devices per client, the deployment variant, and the mean inference latency. In cases where the mean latency was very high (more than 10^4 ms), the inference was not real time due to the formation of large queues of in-flight requests. It can be observed that one-pod deployments were not able to support real-time scenarios with 16 clients and 40 devices per client, or 4 clients and 160 devices per client (a total of 640 devices). Increasing the number of MIS pods to two allowed the server to handle these two scenarios. Consequently, four pods could handle 16 clients with 160 devices each, thus raising the total number of supported devices to 2560. When comparing the one-pod deployments (with or without load balancing), it can be seen that the load balancer appeared to be introducing additional latency. This is expected—the load balancer is essentially a relay which requires additional CPU time. However, for many-pod deployments, the load balancer must be included to help the system scale to a higher number of devices.

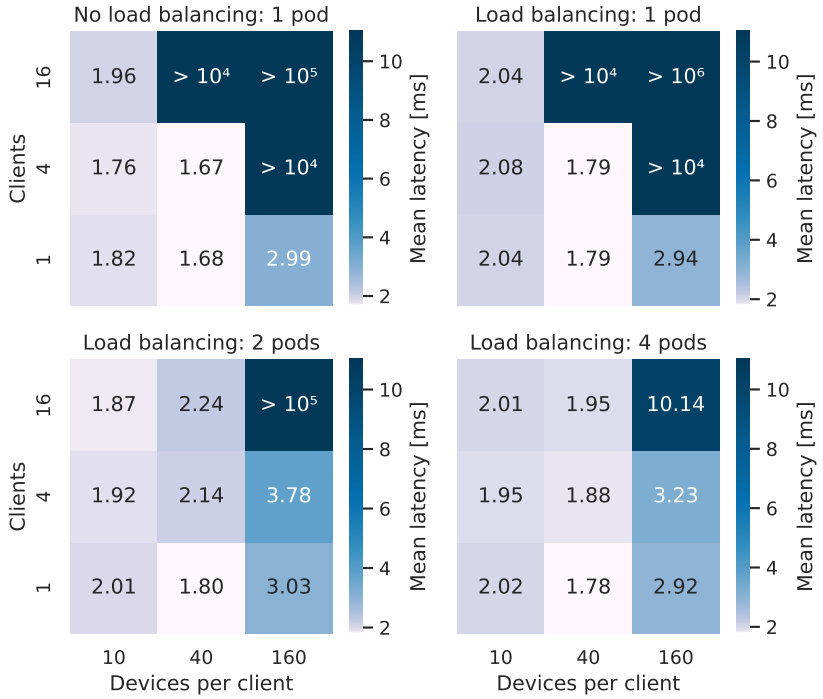

Figure 11. Mean inference request–response latency for all deployment variants on the server.

The same stream saturation phenomenon to that found in the GWEN deployment is noticeable in Figure 11, when comparing the experiments with 10 and 40 devices per client. Namely, the variants with 40 devices per client sometimes have lower mean latencies, which may stem from the same root cause as with the GWEN.

The latency distributions for 40 devices per client are visualized in Figure 12. In the first two subplots, the peaks of the distributions for variants with load balancing are visibly shifted to the right in relation to the variant without load balancing. This implies that the load balancer consistently increased the latency. The distributions for higher numbers of clients are also flatter, with a higher overall variance. This is caused by the clients having to wait for the server to finish processing a request made by a different client.

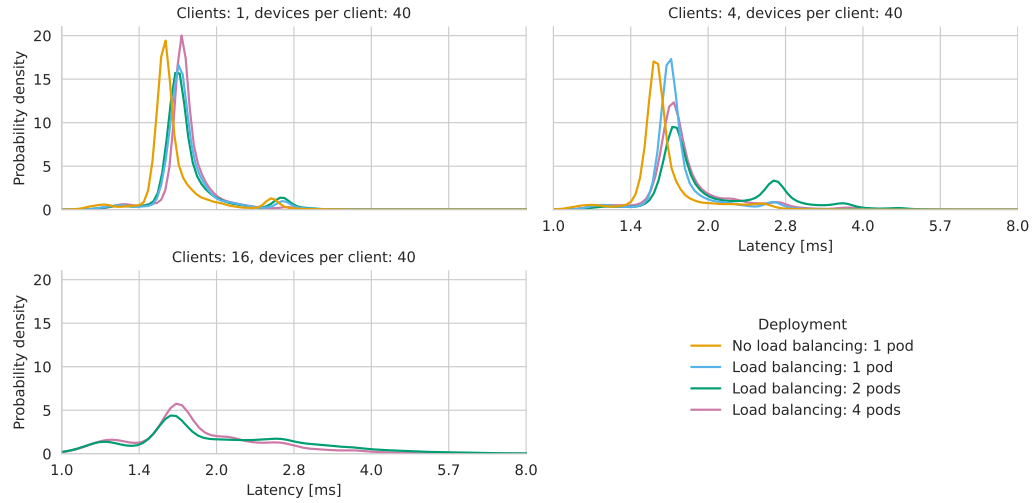

Figure 12. Fall inference request–response latency distribution for all deployment variants on the server, with 40 devices per client. The distribution is visualized using a kernel density estimate. The x axis is logarithmic. The missing data series are for experiments which did not execute in real time.

Finally, performance metrics collected during the experiments were explored. Figure 13 visualizes how the total CPU usage and network traffic scaled with a growing workload. For the GWEN (on the left subplot) it can be seen that CPU usage increased predictably with the increasing number of devices. It should be noted here that the MIS is a Python application. Python uses the Global Interpreter Lock (GIL), which limits the number of active Python interpreter threads to one [43]. This means that the MIS is almost entirely single-threaded—almost, because code outside the interpreter (e.g., optimized numerical routines, network code) can execute asynchronously. Therefore, for 160 devices, the GWEN CPU usage was observed to be on average 114.8%, which corresponds to one fully loaded MIS instance. For 80 devices, the average CPU usage was at 95.9%.

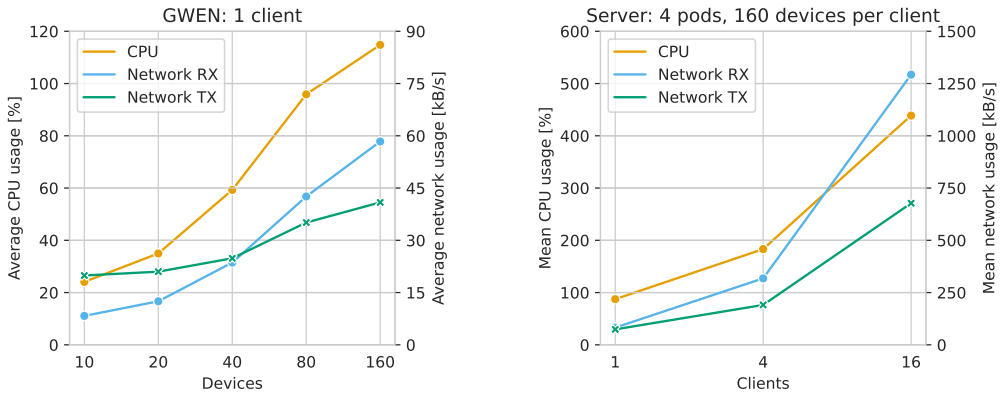

Figure 13. Total CPU and network usage on the GWEN and the server for fall inference. Both X axes are logarithmic. Network usage was measured on the Ethernet interface of the device that hosted the MIS. Note that 100% CPU usage corresponds to one fully loaded CPU core; hence, values over 100% are possible on multicore systems.

There was a disparity between network transmission (TX) and receive (RX) usages for 10 and 20 devices on the GWEN. In the fall detection use case, the requests to the MIS were larger than its responses, and therefore the RX should have been higher. This can be, however, explained by the presence of the Prometheus metrics exporters on the GWEN. The client workstation's Prometheus instance regularly reads the metrics from the exporters, thus adding significant TX traffic. This traffic is constant and independent of the number of devices.

For the server, with four pods and 160 devices per client, the CPU usage scaled predictably to an average of 438.7% with 16 clients. The average total network usage (RX + TX) reached 1.92 MB/s in the most challenging scenario, thus using approximately only 1.5% of the total bandwidth of the Gigabit connection.

5.2. Scratch Detection Results

In the scratch detection use case, the most important performance aspect is the time needed to process a complete vehicle scan. Contrary to the fall detection use case, individual request latency is not important, and therefore the focus of this analysis was different.

Figure 14 illustrates the total time needed to process a complete vehicle scan, from the first request being sent to the last response being received. The most significant difference that can be observed is the impact of batch sizes. The processing time dropped sharply by increasing the batch size to 4 and then to 16. This is because the GPU can perform inference on multiple images in parallel—at the cost of higher memory usage. Therefore, with larger batch sizes, the GPU is expected to be better utilized.

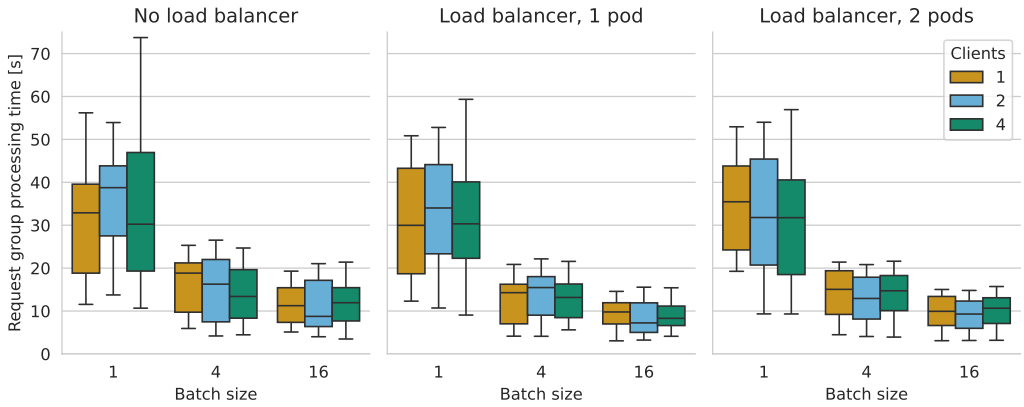

Figure 14. Distribution of the total time to process a scratch inference request group (a complete vehicle scan).

The plots in Figure 14 show a very high variance for all experiments, which was largely due to the varying vehicle scan size (from 50 to 300 images). This variance makes it harder to assess the impact of other variables (number of clients, deployment). Therefore, Table 4 presents the same results, but they have been normalized by the number of images in each vehicle scan, thereby effectively removing this variance. Firstly, it is noticeable that for batch sizes 4 and 16, and with one MIS pod, the deployment with load balancing was consistently faster than no load balancing. The variance was also decreased with the load balancer, especially for the more challenging cases with more clients. This was likely caused by the additional buffering done by the load balancer, which can increase throughput and make the traffic flow more consistent. Secondly, it can be observed that the difference between times with one and two pods was minimal.

Table 4. Total time to process a scratch inference request group (a complete vehicle scan) normalized by the number of images in the scan. The indicated values are the mean time ± standard deviation in milliseconds. Lowest mean values in a given row are bolded.

Clients	Batch Size	No Load Balancer	Load Balancer, 1 Pod	Load Balancer, 2 Pods
1	1	187.83 ± 2.27	**179.64** ± 5.90	181.35 ± 0.95
1	4	89.61 ± 2.92	75.01 ± 2.49	**74.44** ± 1.44
1	16	68.15 ± 1.31	**53.66** ± 2.01	53.76 ± 3.23
2	1	179.62 ± 3.01	**175.93** ± 0.86	180.93 ± 1.82
2	4	82.42 ± 4.90	74.29 ± 1.68	**73.75** ± 1.13
2	16	70.35 ± 10.37	56.34 ± 4.16	**55.40** ± 3.33
4	1	211.53 ± 40.27	**180.57** ± 8.43	183.53 ± 16.20
4	4	85.76 ± 4.28	74.48 ± 1.96	**73.59** ± 1.37
4	16	68.53 ± 7.33	**55.00** ± 2.08	55.20 ± 2.69

Moving on to performance metrics, Figure 15 visualizes the GPU utilization and GPU memory usage in different deployments with four clients. Looking at the GPU utilization distribution (top-left subplot), it is evident that the GPU usage peaks was higher with larger batch sizes, as expected. For a batch size of 16, the peak utilization reached 100%, thereby making use of the full potential of the GPU. Although the peak utilization increased with batch size, the opposite was true for mean utilization (top-right subplot). Effectively, total processing time on the GPU decreases with larger batch sizes. Together, these two observations show that larger batch sizes are doubly beneficial—by both using

the hardware to the maximum and utilizing less GPU time overall—thereby leaving more time for other processes. The only outlier in these results was the higher than expected mean GPU usage for two pods and a batch size of four, which was most likely caused by a random occurrence.

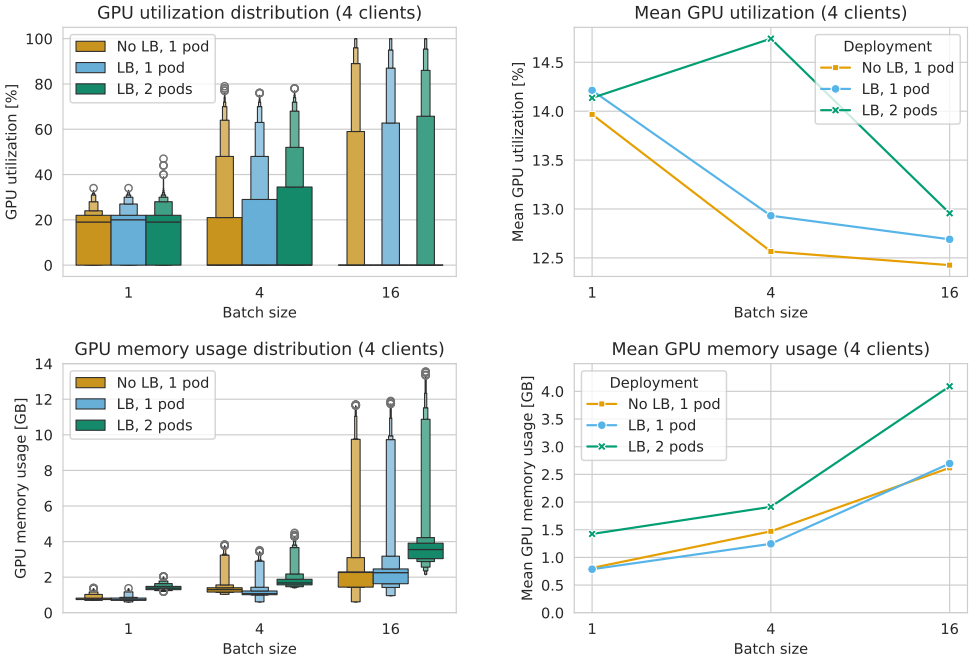

Figure 15. Distribution and mean of GPU utilization and GPU memory (frame buffer) usage during scratch inference experiments with 4 clients. The X axes on the right plots are logarithmic.

The GPU memory usage increased predictably with the batch size, as higher batch sizes require linearly more memory. With four clients, two pods, and a batch size of 16, the GPU memory usage reached almost 14 GB, which is well below the capacity of this GPU (24 GB). Unlike the mean GPU utilization, the mean GPU memory usage was positively correlated with the batch size.

Figure 16 illustrates network usage in different deployment variants with four clients. Firstly, on the left subplot, it can be observed that the peak network receive usage was noticeably higher with a load balancer in place, thereby reaching up to ~120 MB/s (the limit of the Gigabit Ethernet connection). This is likely due to the additional buffering done by the load balancer, which in turn allows the network bandwidth to be used in full. As for network transmit, the usage increased predictably with batch size, thus peaking at about 35 MB/s for a batch size of 16 and two pods with load balancing.

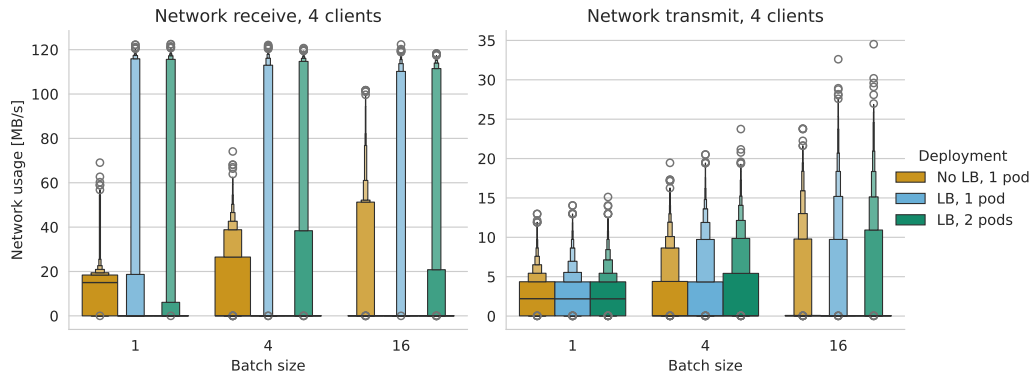

Figure 16. Distribution of receive/transmit network traffic on the Ethernet interface of the GPU server during scratch inference experiments with 4 clients.

6. Related Works

In the following section, a review of the state-of-the-art ML inference servers and ML inference systems is conducted. A special emphasis is put on comparing their capabilities to the Modular Inference Server and the Component Repository.

6.1. Machine Learning Inference Servers

There are many industry examples of machine learning inference servers [27,44–46] with support for versatile features, including different communication protocols, ML libraries, and hardware types used for inference. Some of those solutions also provide the ability to serve multiple ML models in parallel. Others include accommodations for more complex ML pipelines, thereby involving models in ensembles or chained sequentially. When it comes to managing resource-constrained environments, a wide variety of approaches has been utilized, with SmartLite [39] being a particularly interesting one. It leverages a database management system to store the model parameters and structural information of neural networks, thus achieving a smaller memory footprint aside from lower memory consumption. However, as the framework does not focus on providing complex ML preprocessing and postprocessing pipelines, it does not fully suit the requirements of the study. EdgeEye [41], on the other hand, includes optimized inference engines to support deployment on edge. Nevertheless, as it explicitly focuses on providing real-time intelligent video analytics, it is of limited use for our purposes.

Recent work in this research area focuses on providing more sophisticated functionalities to ML inference servers, for instance by optimizing GPU utilization in multimodel workflows [47]. Unfortunately, some of the industry ML serving approaches are designed by large companies for specific edge environments and as such do not prioritize interoperability. Instead, they prefer to limit themselves to certain dedicated runtimes and focus on maintaining maximal efficiency [48,49]. Others, like TensorFlow Serving [27] and Triton Inference Server [44], do allow some interoperability by supporting custom inference serving components, but they do not extend this functionality to full flexible preprocessing pipelines. TensorFlow Serving [27] is an especially interesting example, as it enables the user to flexibly upload and update components (which can include preprocessing and postprocessing steps) by placing them in a selected directory. However, for a new component to be added, it requires the presence of dedicated classes managing its lifecycle in the application. As such, only predefined components can be easily swapped, with no support towards changing their order or type without recreating the application. Only the AMD Inference Server [45] and TorchServer [46] allow for the flexible reconfiguration of preprocessing pipelines in some limited form.

Many (among them the AMD Inference Server, as well as the aforementioned TensorFlow Serving, Triton Inference Server, and TorchServe) include gRPC for inference by default, thereby employing well-maintained and documented serialization formats. However, TorchServe allows for the flexible pipelines to be served only through HTTP API, thereby excluding gRPC. Finally, Roboflow Inference [50], aside from acting as a regular inference server, can provide inference based on an RSTP or UDP stream when supplied with the appropriate IP address.

A comparison of publicly available ML inference servers can be found in Table A2 in Appendix B. In general, the landscape of ML inference servers is dominated by frameworks developed by the industry. Most of them are published under open licenses, such as Apache 2.0 or MIT. The majority provides adaptations for multiple processor architectures, as well as the use of various accelerators for ML inference.

6.2. Machine Learning Inference Systems

Machine learning inference systems tend to focus more on ensuring well-functioning inference in the cloud and therefore commonly incorporate various technologies designed for those environments [10,11]. For example, MArk [5] minimizes the cost of ML serving infrastructures constructed using AWS tools by effectively utilizing predictive autoscaling

and dynamic batching. BATCH [51] expands on that idea by using a dedicated optimizer to provide tail latency guarantees and adaptive batching support. S3ML [52] optimizes the functioning of a machine learning inference system while ensuring the additional security of the solution by leveraging the capabilities of a Trusted Execution Environment (TEE). Here, a TEE stands for a secure, integrity-protected processing environment, which includes storage and memory capabilities alike [53]. Interestingly, among the technologies utilized by S3ML are Kubernetes and gRPC, although gRPC streaming has not been attempted. RILaaS [54] also heavily uses gRPC to provide low-latency and reliable serving of ML inference in cloud, edge, and fog environments, albeit focusing mostly on the area of robotics. DLHub [55] takes a narrow focus as well, thereby aiming to design an ML inference serving solution dedicated to scientific research. It includes a model database with descriptive metadata, which is devised to make ML models developed as a result of publicly available research. In addition, the authors performed a comparison of various approaches to ML serving, in which the gRPC interface of TensorFlow Serving was one of the best in terms of performance. Clipper [6] prioritizes making ensemble predictions more robust, thus addressing the need for relevant straggler mitigation. It also makes it possible to dynamically select the inference model to be used by the service.

A tendency towards the automation of ML inference systems, to the point of letting the system automatically select a variant of the model, is exhibited in INFaaS [56]. Here, a different variant is selected per query, with possible criteria including performance, cost, and accuracy requirements. Kairos [57], on the other hand, leverages a query distribution to optimially utilize heterogenous cloud resources. Finally, hybrid models [58] smartly balances the accuracy guaranteed by inference performed on large models, which can be deployed in the cloud, with the low latency achieved through running smaller models in edge environments. By automatically filtering particularly demanding queries and sending them to the cloud, the system successfully integrates two sides of the Cloud–Edge–IoT continuum.

Some ML inference systems leverage distributed computing to enable a developer to deploy them on the edge. Often, DNN partitioning is used to effectively accelerate the prediction process by distributing the workloads to nearby mobile devices, such as in Edgent [40]. EdgeFlow [59] successfully extends this technology to more complex model architectures, thereby providing wider functionality and ensuring that the solution remains up to date with contemporary trends. Both focus on the acceleration of edge inference rather than built-in support for complex preprocessing and postprocessing functionalities.

InferLine [4] describes a very interesting serving solution, which aims to deploy complex ML pipelines, thereby allowing the administrator to freely chain models and preprocessing steps as long as they can be expressed as Directed Acyclic Graphs (DAGs). A similar approach is included in popular production-grade systems such as KServe [11] and Seldon Core [10]. Flexible inference with three types of reusable components (including data transformations) stored in a dedicated library is offered by LASER [60] to facilitate rapid experimentation and to quickly accommodate changes in the system. Unfortunately, the design of the solution is suited more towards large clusters: a limited number of models and ML libraries is supported, and the solution is not openly available. Velox [61] similarly promotes reusability by storing each ML model with a dedicated, custom preprocessing function. However, this method of providing custom preprocessing is not designed for frequent and quick pipeline modifications. Rafiki [62] is a combined system for ML training and inference, where the inference is considered as a supplementary feature. It expects all preprocessing functions to be already present in the training code, thus limiting its applicability.

The most commonly used and publicly available inference system solutions were compared in Table 5. They are considerably lesser in number than publicly available ML inference servers, as they can possibly integrate the aforementioned servers into their Kubernetes infrastructure in order to gain their features. Due to this fact, although there are custom servers built for a specific GPU architecture or ML platform, those servers may

be quite easily combined with more general inference systems [10,11]. All three of the compared systems support similar functionalities, such as autoscaling, tracing, and drift detection. Two of them also include support for flexible preprocessing pipelines.

Table 5. A comparison of publicly available machine learning inference systems.

	KServe [11]	**Seldon Core [10]**	**BentoML [63]**
API protocols	gRPC, HTTP	gRPC, HTTP	gRPC, HTTP
Supported architectures	PPC641E, ARM64, x86-64, S390X	x86-64	ARM64, x86-64
Supported platforms	Kubernetes	Kubernetes	Kubernetes, Bento Cloud, OpenLLM, OneDiffusion
Drift detection	Yes, with bias/outlier detection	Yes	Yes
Autoscaling	Yes	Yes	Yes
Tracing	No	Yes	Yes
Load balancing	No	Yes	No
Ensemble support	Yes	Yes	Yes
Explainability support	Yes	Yes	Yes
Flexible preprocessing	Yes	Yes	Yes
Extensible component repository	No	No	No
License	Apache 2.0	Business Source License 1.1	Apache 2.0

6.3. Comparison with State of the Art

Based on the presented state-of-the-art overview, a substantial research gap can be identified. There is a clear lack of serving solutions that would provide flexibility, reusability, and interoperability in the process of creating inference pipelines while remaining suitable for edge deployment in terms of resource utilization and communication speed. This gap is addressed by the Modular Inference Server.

In summary, although the MIS does not contain features supporting the deployment of multiple models simultaneously, it does offer functionalities that are rare for standalone ML inference servers, such as custom, modular, and flexible preprocessing and postprocessing. These functionalities tend to be more common in ML inference systems (as showcased in Appendix B), which commonly involve larger infrastructural overhead and are therefore less suitable for edge environments. Other frameworks with custom, flexible preprocessing both incur significant limitations. The AMD Inference Server [45] is optimized mainly for AMD products. This means that it does not fulfill the use case requirements due to the infrastructure including devices with an ARM64 CPU and a Nvidia GPU. Additionally, the AMD Inference Server does not offer a gRPC streaming interface, which is used in the MIS to greatly reduce the latency and to improve the reliability of inference services. TorchServe [46] similarly supports flexible pipelines only when served through RESTful HTTP interfaces, with no possibility of integration with the gRPC capabilities of the framework. Furthermore, although TorchServe allows the user to define complex ML inference workflows with custom components, it does not enable easy parameter modification or component repurpose and rearrangement. Instead, the workflow as a whole has to be unregistered, modified, compressed to the specified format, and reuploaded to apply even the smallest of changes, with no built-in versioning or updating capabilities. Similarly,

even though the application is able to dynamically download a workflow based on the input URI, no tools for managing the remote archive are supplied. In contrast, the MIS incorporates a unique (for ML inference systems and servers alike) approach to reusability in the form of an easily extensible component repository, which allows the administrators to quickly add or change selected components across many deployments.

A full feature comparison table of publicly available machine learning inference servers, including the MIS, can be found in Table A2 in Appendix B. The Table describes aspects such as available API protocols and ML runtimes, CPU architecture and inference hardware support, the license of the ML inference server, as well as the inclusion of various functionalities facilitating complex ML inference pipelines. ML inference systems were not included in the comparison due to a major difference in design assumptions. The MIS must be suitable not only for cloud but also edge deployment, thus making it fully functional as a standalone application to be prioritized. The MIS cannot be strictly classified as an ML inference system, as the Kubernetes-based orchestration tools provided with it are only supplementary. Therefore, publicly available ML inference servers were used as a benchmark instead.

7. Discussion and Future Work

What follows is a discussion of the obtained results, along with an outline of future work directions. Investigated here are the functionalities of the MIS and the Component Repository, performance and deployment aspects, and the specific use cases that were included in the experiments.

7.1. Software Functionalities

The Modular Inference Server and the Component Repository have demonstrably covered the needs of the two tested use cases. The software was successfully applied in pilot trials involving real hardware, as well as in relevant simulated workloads. However, for future production deployments, further work on selected improvements and functionalities would be needed.

Firstly, the software currently assumes that the deployment is done in a secure private network, without the possibility of any foul play—these presuppositions were reflected in the approach to interface design used by the system. Introducing authentication, authorization, and encryption to increase the security of these components would allow for a broader range of applications. Secondly, the mechanism of modular inference pipelines could be extended to support the integration of multiple models chained together or forming an ensemble. Similarly, the pipeline module format verification in the MIS could be formalized and expanded into a semantically coherent system. Integrated into a command line tool and coupled with the Component Repository, it would aid users in designing custom pipelines based on pre-existing components, thereby increasing overall reusability. Additionally, a broader analysis of potentially useful features adapting the MIS to scenarios beyond the ASSIST-IoT use cases in the Cloud–Edge–IoT continuum could be conducted, thus focusing on important aspects such as varying data location or geographical distribution [64].

7.2. Performance and Deployment

The Modular Inference Server was tested in a range of demanding and diverse benchmarks, thus demonstrating robust performance. It could cater both to latency-sensitive workloads (fall detection) and to throughput-restricted ones (scratch detection). The software scaled both up/down (from the GWEN to the x86-64 server) and out (with multiple MIS instances per server). It also managed to run on two of the most popular CPU architectures (x86-64 and ARM64) and make efficient use of the GPU when available. As such, it has fulfilled the requirements of adaptability within the described use cases. Due to the use of Docker, the MIS can be assumed to be usable on any reasonably modern x86-64 or ARM64 platform supporting containers, which covers a significant portion of the computing continuum. The range of supported CPU architectures can be extended in the future

to include, for example, RISC-V. Adapting the MIS to new architectures is feasible, given the few dependencies of the MIS (the most important are Docker and Python, which are already ported to RISC-V). However, some of the needed machine learning libraries (e.g., TensorFlow) are at the time of writing still poorly supported on RISC-V, thus constituting the largest barrier. A broader range of ML acceleration hardware should also be tested, including various Neural Processing Units (NPUs) and GPUs from other vendors. This would be especially useful for energy-efficient edge devices such as the GWEN. Additionally, further experiments, including scenarios with the load balancer managing the communication between devices of different computational abilities, should be included in future work.

As for Cloud–Edge–IoT continuum support, the MIS can not only run in the standalone mode suitable for resource-constrained edge devices but also in larger Kubernetes deployments, thus making it suitable for the cloud or cloud-native deployments on the edge [12]. Extending the MIS's support to IoT devices depends largely on the device in question—some would argue, for example, that the GWEN and comparable hardware platforms (e.g., RaspberryPi) qualify as IoT devices. As of now, the MIS supports any x86-64 or ARM64 platform that can run Docker. Using the MIS on smaller devices is currently not possible but could be investigated, for example, with the use of WebAssembly [65]. This technology, although very promising, is still relatively young, and it is not immediately clear how feasible would such an implementation be.

The results from the conducted experiments can be used to draw general conclusions about other workloads that can be deployed with the MIS. While investigating the effect a load balancer had on deployments with one MIS instance, it has been observed that although it did increase latency in the fall detection use case, it decreased the time to process a vehicle scan in scratch detection. This is a well-known trade-off of latency versus throughput—with additional buffering and optimized networking code, the load balancer can increase throughput, although at the cost of latency. Thus, when deploying workloads with the MIS, using the load balancer for one-pod deployments is recommended only if throughput is more important for the use case.

Analyzing the collected performance metrics allowed us to determine the bottlenecks in both use cases. Depending on the used pipeline and the workload characteristics, the availability of different resources may be the limiting factor. For example, the fall detection use case is visibly CPU-bound, and more devices could be handled by simply increasing the number of available CPU cores and MIS instances. On the other hand, in scratch detection, full parallelization can only be achieved with batching, which is in turn restricted by the available GPU memory. Peak inference throughput also appears to be limited by network speed; therefore, employing faster networking (e.g., 10 Gigabit Ethernet) or compressing the requests may substantially decrease inference time. In summary, for a given workload, it is essential to evaluate its resource use, determine the bottlenecks, and select appropriate hardware.

7.3. Use Cases

In the fall detection use case, the GWEN could responsively handle up to 80 client devices with a single MIS instance, thus meeting the relevant requirements. The achieved latencies were well within the expected ranges, while the in-flight request count analysis shows that the MIS was able to process the requests in real time. In the server deployment the solution scaled predictably—most likely it would be able to process thousands of devices on many-core CPUs. In the future, using a specialized ML accelerator could be investigated to further increase the number of supported devices per GWEN, or to make it possible to use a larger ML model. The accelerator must be chosen carefully so as to support very-low-latency inference, which would be very hard to achieve with standard GPUs employing batching and time sharing concurrency.

For the scratch detection use case, the MIS supported up to four clients without any issues in all deployment scenarios. Network bandwidth was clearly a bottleneck,

which was mostly due to the images being sent as raw tensors without compression. Each image encoded in such a manner takes up 3 MB, which would be significantly decreased if, for example, JPEG encoding was used instead. This may be achieved by either treating the JPEG image as a flat tensor of unsigned 8-bit integers (bytes) or by implementing a custom gRPC `Service` in the MIS. Higher processing throughput per server machine could likely be obtained by using Multi-Instance GPU (MIG) technology, which allows multiple applications to use the same GPU concurrently [66]. This, however, requires using highly specialized hardware.

8. Conclusions

In this work, the system for flexible and reusable ML inference in Cloud–Edge–IoT environments is presented in the form of the novel Modular Inference Server supplemented by the Component Repository. The system was compared with other publicly available ML inference serving applications in terms of feature support and suitability for the edge environment. The proposed approach possesses a unique combination of functionalities not found in state-of-the-art solutions. It features modular and flexible inference pipelines, integration with a database of reusable components, and bidirectional streaming in gRPC. The MIS is a significant advancement over competing solutions by offering full modularity while being applicable to a wide variety of deployment scenarios across the Cloud–Edge–IoT continuum. The proposed system was rigorously tested in two scenarios based on varied, real-life use cases. In the conducted experiments, the solution's performance, scalability, and stability was evaluated. The results confirm that the MIS can support the two use cases while maintaining performance in line with the relevant requirements. The solution is fully open-source and publicly available on GitHub (https://github.com/Modular-ML-inference) (accessed on 3 April 2024).

Author Contributions: Conceptualization, K.B., P.S., F.M.B., M.G. and K.W.-M.; methodology, K.B., P.S. and A.D.; software, K.B., P.S., A.D. and K.W.-M.; validation, P.S. and F.M.B.; formal analysis, K.B., P.S. and A.D.; investigation, K.B. and P.S.; resources, M.G. and M.P.; data curation, K.B., P.S. and A.D.; writing—original draft preparation, K.B., P.S. and A.D.; writing—review and editing, F.M.B., M.G., M.P., K.W.-M. and C.E.P.; visualization, K.B., P.S. and A.D.; supervision, M.G., M.P. and C.E.P.; project administration, M.G. and C.E.P.; funding acquisition, M.G., M.P. and C.E.P. All authors have read and agreed to the published version of the manuscript.

Funding: This work was funded by the European Commission, in part under the Horizon 2020 project ASSIST-IoT, grant number 957258. The work of Marcin Paprzycki and Katarzyna Wasielewska-Michniewska was funded under the Horizon Europe project aerOS, grant number 101069732.

Institutional Review Board Statement: Ethical review and approval were waived for this study due to the fact that the experiments carried out (collecting acceleration data) were not classified as medical research, and they did not involve more than a minimal risk to the subjects.

Informed Consent Statement: Informed consent was obtained from all subjects involved in gathering the acceleration dataset.

Data Availability Statement: The data presented in this study are openly available in Zenodo at https://doi.org/10.5281/zenodo.10508868 (accessed on 3 April 2024), reference number 10508868. The software introduced in this study is available in Zenodo at https://doi.org/10.5281/zenodo.10508858 (accessed on 3 April 2024), reference number 10508858; https://doi.org/10.5281/zenodo.10508862 (accessed on 3 April 2024), reference number 10508862; https://doi.org/10.5281/zenodo.10508864 (accessed on 3 April 2024), reference number 10508864.

Conflicts of Interest: The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

AKS	AI Kernel Scheduler
API	Application Programming Interface
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
GIL	Global Interpreter Lock
GPU	Graphics Processing Unit
GWEN	Gateway Edge Node
IoT	Internet of Things
LB	Load Balancer
LSTM	Long Short-Term Memory
MIG	Multi-Instance GPU
MIS	Modular Inference Server
ML	Machine Learning
RAM	Random Access Memory
RGB	Red Green Blue
RX	Network Receive
TEE	Trusted Execution Environment
TX	Network Transmit

Appendix A. Detailed Fall Inference Results

Table A1 presents detailed results for request–response latency in the fall detection use case for all deployment variants. In the table, large latency values were summarized as "$>10^x$", where $x \geq 3$. In these cases, the MIS operated with a very long queue of requests, and therefore fall detection was not performed in real time.

Table A1. Request–response latency in the fall detection benchmark: 5th percentile, median, 95th percentile, and maximum. All values are in milliseconds.

Deployment	Clients	Devices/Client	5 pctl	Median	95 pctl	Max
GWEN	1	10	7.09	7.98	8.37	61.83
GWEN	1	20	6.73	7.88	8.25	48.60
GWEN	1	40	6.58	7.52	8.26	49.70
GWEN	1	80	6.57	11.63	24.13	56.39
GWEN	1	160	$>10^4$	$>10^5$	$>10^5$	$>10^5$
Server no LB, 1 pod	1	10	1.55	1.72	2.30	36.52
Server no LB, 1 pod	1	40	1.41	1.60	2.49	41.93
Server no LB, 1 pod	1	160	1.51	2.57	5.53	38.21
Server no LB, 1 pod	4	10	1.50	1.68	2.18	48.49
Server no LB, 1 pod	4	40	1.37	1.59	2.34	43.08
Server no LB, 1 pod	4	160	$>10^3$	$>10^4$	$>10^4$	$>10^4$
Server no LB, 1 pod	16	10	1.32	1.74	3.09	90.34
Server no LB, 1 pod	16	40	$>10^3$	$>10^4$	$>10^5$	$>10^5$
Server no LB, 1 pod	16	160	$>10^5$	$>10^5$	$>10^5$	$>10^5$
Server LB, 1 pod	1	10	1.72	1.94	2.57	38.66
Server LB, 1 pod	1	40	1.53	1.71	2.50	43.64
Server LB, 1 pod	1	160	1.62	2.69	5.04	51.54
Server LB, 1 pod	4	10	1.67	1.88	3.03	40.48
Server LB, 1 pod	4	40	1.49	1.70	2.54	49.20
Server LB, 1 pod	4	160	$>10^3$	$>10^4$	$>10^4$	$>10^4$
Server LB, 1 pod	16	10	1.43	1.86	3.03	67.29

Table A1. *Cont.*

Deployment	Clients	Devices/Client	5 pctl	Median	95 pctl	Max
Server LB, 1 pod	16	40	$>10^3$	$>10^4$	$>10^4$	$>10^4$
Server LB, 1 pod	16	160	$>10^5$	$>10^6$	$>10^6$	$>10^6$
Server LB, 2 pods	1	10	1.73	1.92	2.49	47.70
Server LB, 2 pods	1	40	1.52	1.70	2.65	42.87
Server LB, 2 pods	1	160	1.61	2.68	5.55	43.11
Server LB, 2 pods	4	10	1.64	1.84	2.42	44.64
Server LB, 2 pods	4	40	1.55	1.81	3.53	47.83
Server LB, 2 pods	4	160	1.44	2.93	8.99	49.32
Server LB, 2 pods	16	10	1.31	1.80	2.59	65.65
Server LB, 2 pods	16	40	1.17	1.86	4.25	95.54
Server LB, 2 pods	16	160	$>10^4$	$>10^5$	$>10^5$	$>10^5$
Server LB, 4 pods	1	10	1.70	1.92	2.54	36.53
Server LB, 4 pods	1	40	1.51	1.73	2.15	35.92
Server LB, 4 pods	1	160	1.61	2.70	5.18	43.60
Server LB, 4 pods	4	10	1.67	1.87	2.42	41.78
Server LB, 4 pods	4	40	1.45	1.74	2.74	47.15
Server LB, 4 pods	4	160	1.40	2.75	6.70	53.72
Server LB, 4 pods	16	10	1.44	1.88	2.89	100.77
Server LB, 4 pods	16	40	1.17	1.74	3.29	130.65
Server LB, 4 pods	16	160	1.10	3.62	39.29	462.47

Appendix B. Comparison Table of Machine Learning Inference Servers

Table A2 summarizes feature support offered by various publicly available machine learning inference servers. Here, custom inference stands for the ability to create custom components responsible for obtaining predictions from the model; custom preprocessing—the ability to create custom preprocessing components; flexible preprocessing—the ability to freely connect and modify the order of those components without substantially changing the underlying application; and extensible component repository—the availability of a database storing the aforementioned pluggable components, which contains more than ML models bundled with inference serving interfaces and can be easily extended by the administrator. When looking at supported architectures, an effort was made to check the documentation for any mention of available Docker images or instructions on how to build them dedicated to that architecture. The table also includes the proposed solution (the Modular Inference Server with the Component Repository), comparing it with state-of-the-art approaches.

Table A2. Comparison of existing machine learning inference servers, including the proposed solution.

	AMD Inference Server [45]	**Triton Inference Server [44]**	**BentoML [63]**	**TorchServe [46]**	**Multi Model Server [49]**	**Roboflow Inference [50]**	**Tensorflow Serving [27]**	**ForestFlow [67]**	**DeepDetect [68]**	**Presented Solution**
API protocols	gRPC, HTTP, WebSocket	gRPC, gRPC stream, HTTP	gRPC, HTTP	gRPC, gRPC stream, HTTP	HTTP	HTTP	gRPC, gRPC stream, HTTP	GraphPipe, HTTP	HTTP	gRPC, gRPC stream
Supported architectures	x86-64	ARM64, x86-64	ARM64, x86-64	x86-64	x86-64	ARM64, x86-64	x86-64	x86-64	ARM, ARM64, x86-64	ARM64, x86-64
Inference hardware support	x86-64 CPU, AMD GPU, AMD FPGA	ARM64 CPU, x86-64 CPU, AWS Inferentia, Nvidia GPU	ARM64 CPU, x86-64 CPU, Nvidia GPU	x86-64 CPU, Nvidia GPU	x86-64 CPU, Nvidia GPU	ARM64 CPU, x86-64 CPU, Nvidia GPU, Nvidia Jetson	x86-64 CPU, Nvidia GPU	x86-64 CPU	ARM CPU, ARM64 CPU, x86-64 CPU, Nvidia GPU	ARM64 CPU, x86-64 CPU, Nvidia GPU
ML runtime support	ONNX, PyTorch, TensorFlow	ONNX, PyTorch, TensorFlow, others	ONNX, PyTorch, TensorFlow, others	ONNX, PyTorch, TensorRT, others	ONNX, MXNet	ONNX	TensorFlow	H2O	Caffe, TensorFlow, others	PyTorch, TensorFlow
Multi-model inference	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No
Ensemble support	Yes	Yes	Yes	Yes	No	No	No	No	No	No
Custom inference	Yes	Yes	Yes	Yes	Yes	No	Yes	No	No	Yes
Custom preprocessing	Yes	Yes	Yes	Yes	Yes	No	Yes	No	No	Yes
Flexible preprocessing	Yes, using AKS	No	No	Yes, using Workflows	No	No	No	No	No	Yes
Extensible component repository	No	No	No	No	No	No	No	No	No	Yes
License	Apache 2.0	BSD 3-Clause	Apache 2.0	Apache 2.0	Apache 2.0	Apache 2.0	Apache 2.0	Apache 2.0	GNU LPGL	Apache 2.0

References

1. Pan, J.; McElhannon, J. Future Edge Cloud and Edge Computing for Internet of Things Applications. *IEEE Internet Things J.* **2018**, *5*, 439–449. [CrossRef]
2. Ferry, N.; Dautov, R.; Song, H. Towards a model-based serverless platform for the cloud-edge-iot continuum. In Proceedings of the 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid), Taormina, Italy, 16–19 May 2022; pp. 851–858.
3. Baier, L.; Jöhren, F.; Seebacher, S. Challenges in the Deployment and Operation of Machine Learning in Practice. In Proceedings of the ECIS 2019—27th European Conference on Information Systems, Stockholm & Uppsala, Sweden, 8–14 June 2019; Volume 1.
4. Crankshaw, D.; Sela, G.E.; Mo, X.; Zumar, C.; Stoica, I.; Gonzalez, J.; Tumanov, A. InferLine: Latency-Aware Provisioning and Scaling for Prediction Serving Pipelines. In Proceedings of the 11th ACM Symposium on Cloud Computing, New York, NY, USA, 26–30 July 2020; pp. 477–491. [CrossRef]
5. Zhang, C.; Yu, M.; Wang, W.; Yan, F. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC 19), Renton, WA, USA, 10–12 July 2019; pp. 1049–1062.
6. Crankshaw, D.; Wang, X.; Zhou, G.; Franklin, M.J.; Gonzalez, J.E.; Stoica, I. Clipper: A Low-Latency Online Prediction Serving System. In Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), Boston, MA, USA, 27–29 March 2017; pp. 613–627.
7. Turnbull, J. *The Docker Book: Containerization Is the New Virtualization*; James Turnbull: Brooklyn, NY, USA, 2014.
8. Kubernetes Developers. Kubernetes Documentation. Available online: https://kubernetes.io/docs/home/ (accessed on 3 February 2024).
9. Derakhshan, B.; Mahdiraji, A.R.; Rabl, T.; Markl, V. Continuous Deployment of Machine Learning Pipelines. In Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), Lisbon, Portugal, 26–29 March 2019; pp. 397–408.
10. Seldon Technologies Ltd. Seldon Core Documentation. Available online: https://docs.seldon.io/projects/seldon-core/en/latest/ (accessed on 26 February 2024).
11. KServe Authors. KServe Documentation. Available online: https://kserve.github.io/website/latest/ (accessed on 26 February 2024).
12. Vaño, R.; Lacalle, I.; Sowiński, P.; S-Julián, R.; Palau, C.E. Cloud-native workload orchestration at the edge: A deployment review and future directions. *Sensors* **2023**, *23*, 2215. [CrossRef]
13. Casalicchio, E.; Perciballi, V. Measuring Docker Performance: What a Mess!!! In Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion, New York, NY, USA, 22–26 April 2017; pp. 11–16. [CrossRef]
14. ASSIST-IoT Consortium. ASSIST-IoT—H2020 ICT-56-2020—EU RIA Funded Research Project. Available online: https://assist-iot.eu/ (accessed on 3 April 2024).
15. Bogacka, K. Modular-ML-Inference/Inference-Server. 2024. Available online: https://doi.org/10.5281/zenodo.10508864 (accessed on 3 April 2024). [CrossRef]
16. Bogacka, K. Modular-ML-Inference/Component-Repository. 2024. Available online: https://doi.org/10.5281/zenodo.10508862 (accessed on 3 April 2024). [CrossRef]
17. Danilenka, A.; Bogacka, K. Modular-ML-Inference/Ml-Usecase. 2024. Available online: https://doi.org/10.5281/zenodo.10508868 (accessed on 3 April 2024). [CrossRef]
18. Sowiński, P. Modular-ML-Inference/Benchmark-Driver. 2024. Available online: https://doi.org/10.5281/zenodo.10508858 (accessed on 3 April 2024). [CrossRef]
19. Garro, E.; Lacalle, I.; Bogacka, K.; Danilenka, A.; Wasielewska-Michniewska, K.; Tassakos, C.; Theodouli, A.; Kassiani, A.; Palau, C.E. Decentralized Strategy for Artificial Intelligence in Distributed IoT Ecosystems: Federation in ASSIST-IoT. In *Shaping the Future of IoT with Edge Intelligence: How Edge Computing Enables the Next Generation of IoT Applications*; River Publishers: Abingdon, UK, 2024; p. 231.
20. Bogacka, K.; Danilenka, A.; Wasielewska-Michniewska, K.; Paprzycki, M.; Ganzha, M.; Garro, E.; Tassakos, L. Introducing Federated Learning into Internet of Things Ecosystems–Maintaining Cooperation Between Competing Parties. In Proceedings of the Big Data Analytics in Astronomy, Science, and Engineering: 10th International Conference on Big Data Analytics, BDA 2022, Aizu, Japan, 5–7 December 2022; Springer Nature: Cham, Switzerland, 2023; pp. 53–69.
21. gRPC Authors. gRPC Documentation. Available online: https://grpc.io/ (accessed on 3 April 2024).
22. Bolanowski, M.; Żak, K.; Paszkiewicz, A.; Ganzha, M.; Paprzycki, M.; Sowiński, P.; Lacalle, I.; Palau, C.E. Eficiency of REST and gRPC realizing communication tasks in microservice-based ecosystems. *arXiv* **2022**, arXiv:2208.00682.
23. Giretti, A. Understanding the gRPC Specification. In *Beginning gRPC with ASP. NET Core 6: Build Applications Using ASP. NET Core Razor Pages, Angular, and Best Practices in. NET 6*; Apress: Berkeley, CA, USA, 2022; pp. 85–102.
24. de Klerk, J. gRPC on HTTP/2 Engineering a Robust, High-Performance Protocol. 2022. Available online: https://grpc.io/blog/grpc-on-http2/ (accessed on 3 April 2024).
25. Johansson, M.; Isabella, O. Comparative Study of REST and gRPC for Microservices in Established Software Architectures. 2023. Available online: https://www.diva-portal.org/smash/get/diva2:1772587/FULLTEXT01.pdf (accessed on 3 April 2024).
26. Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G.S.; Davis, A.; Dean, J.; Devin, M.; et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. *arXiv* **2015**, arXiv:1603.04467.

27. TensorFlow Serving Contributors. TensorFlow Serving Documentation. Available online: https://www.tensorflow.org/tfx/guide/serving (accessed on 26 February 2024).
28. Ramírez, S. FastAPI. Available online: https://github.com/tiangolo/fastapi (accessed on 26 February 2024).
29. Bradshaw, S.; Brazil, E.; Chodorow, K. *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*; O'Reilly Media: Sebastopol, CA, USA, 2019.
30. Helm Authors 2024. Helm Documentation. Available online: https://helm.sh/docs/ (accessed on 26 February 2024).
31. Rabenstein, B.; Volz, J. *Prometheus: A Next-Generation Monitoring System (Talk)*; USENIX Association: Dublin, Germany, 2015.
32. Danilenka, A.; Sowiński, P.; Rachwał, K.; Bogacka, K.; Dąbrowska, A.; Kobus, M.; Baszczyński, K.; Okrasa, M.; Olczak, W.; Dymarski, P.; et al. Real-time AI-driven fall detection method for occupational health and safety. *Electronics* **2023**, *12*, 4257. [CrossRef]
33. Szmeja, P.; Fornés-Leal, A.; Lacalle, I.; Palau, C.E.; Ganzha, M.; Pawłowski, W.; Paprzycki, M.; Schabbink, J. ASSIST-IoT: A modular implementation of a reference architecture for the next generation Internet of Things. *Electronics* **2023**, *12*, 854. [CrossRef]
34. Harris, C.R.; Millman, K.J.; van der Walt, S.J.; Gommers, R.; Virtanen, P.; Cournapeau, D.; Wieser, E.; Taylor, J.; Berg, S.; Smith, N.J.; et al. Array programming with NumPy. *Nature* **2020**, *585*, 357–362. [CrossRef]
35. He, K.; Gkioxari, G.; Dollár, P.; Girshick, R. Mask R-CNN. *arXiv* **2018**, arXiv:1703.06870.
36. Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. Pytorch: An imperative style, high-performance deep learning library. *Adv. Neural Inf. Process. Syst.* **2019**, *32*, 8024–8035.
37. PyTorch Contributors. Mask R-CNN Documentation. Available online: https://pytorch.org/vision/main/models/generated/torchvision.models.detection.maskrcnn_resnet50_fpn.html#torchvision.models.detection.maskrcnn_resnet50_fpn (accessed on 26 February 2024).
38. Pääkkönen, P.; Pakkala, D.; Kiljander, J.; Sarala, R. Architecture for enabling edge inference via model transfer from cloud domain in a kubernetes environment. *Future Internet* **2020**, *13*, 5. [CrossRef]
39. Lin, Q.; Wu, S.; Zhao, J.; Dai, J.; Shi, M.; Chen, G.; Li, F. SmartLite: A DBMS-Based Serving System for DNN Inference in Resource-Constrained Environments. *Proc. VLDB Endow.* **2023**, *17*, 278–291. [CrossRef]
40. Li, E.; Zeng, L.; Zhou, Z.; Chen, X. Edge AI: On-demand accelerating deep neural network inference via edge computing. *IEEE Trans. Wirel. Commun.* **2019**, *19*, 447–457. [CrossRef]
41. Liu, P.; Qi, B.; Banerjee, S. Edgeeye: An edge service framework for real-time intelligent video analytics. In Proceedings of the 1st International Workshop on Edge Systems, Analytics and Networking, Munich, Germany, 10 June 2018; pp. 1–6.
42. Wang, X.; Li, W.; Wu, Z. CarDD: A New Dataset for Vision-Based Car Damage Detection. *IEEE Trans. Intell. Transp. Syst.* **2023**, *24*, 7202–7214. [CrossRef]
43. Beazley, D. Understanding the Python GIL (Talk). In Proceedings of the PyCON Python Conference, Atlanta, Georgia, 19–22 February 2010.
44. NVIDIA Corporation & Affiliates. Triton Inference Server Documentation. Available online: https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/index.html (accessed on 26 February 2024).
45. Advanced Micro Devices, Inc. AMD Inference Server Documentation. Available online: https://xilinx.github.io/inference-server/main/index.html (accessed on 26 February 2024).
46. PyTorch Contributors. TorchServe Documentation. Available online: https://pytorch.org/serve/ (accessed on 26 February 2024).
47. Choi, S.; Lee, S.; Kim, Y.; Park, J.; Kwon, Y.; Huh, J. Multi-model Machine Learning Inference Serving with GPU Spatial Partitioning. *arXiv* **2021**, arXiv:2109.01611.
48. Wu, C.J.; Brooks, D.; Chen, K.; Chen, D.; Choudhury, S.; Dukhan, M.; Hazelwood, K.; Isaac, E.; Jia, Y.; Jia, B.; et al. Machine Learning at Facebook: Understanding Inference at the Edge. In Proceedings of the 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), Washington, DC, USA, 16–20 February 2019; pp. 331–344. [CrossRef]
49. Multi Model Server Contributors. Multi Model Server Documentation. Available online: https://github.com/awslabs/multi-model-server/blob/master/docs/README.md (accessed on 26 February 2024).
50. Roboflow Inference Contributors. Roboflow Inference Documentation. Available online: https://inference.roboflow.com/ (accessed on 26 February 2024).
51. Ali, A.; Pinciroli, R.; Yan, F.; Smirni, E. BATCH: Machine Learning Inference Serving on Serverless Platforms with Adaptive Batching. In Proceedings of the SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, Atlanta, GA, USA, 9–19 November 2020; pp. 1–15. [CrossRef]
52. Ma, J.; Yu, C.; Zhou, A.; Wu, B.; Wu, X.; Chen, X.; Chen, X.; Wang, L.; Cao, D. S3ML: A Secure Serving System for Machine Learning Inference. *arXiv* **2020**, arXiv:2010.06212.
53. Sabt, M.; Achemlal, M.; Bouabdallah, A. Trusted Execution Environment: What It is, and What It is Not. In Proceedings of the 2015 IEEE Trustcom/BigDataSE/ISPA, Helsinki, Finland, 20–22 August 2015; Volume 1, pp. 57–64. [CrossRef]
54. Tanwani, A.K.; Anand, R.; Gonzalez, J.E.; Goldberg, K. RILaaS: Robot Inference and Learning as a Service. *IEEE Robot. Autom. Lett.* **2020**, *5*, 4423–4430. [CrossRef]
55. Li, Z.; Chard, R.; Ward, L.; Chard, K.; Skluzacek, T.J.; Babuji, Y.; Woodard, A.; Tuecke, S.; Blaiszik, B.; Franklin, M.J.; et al. DLHub: Simplifying publication, discovery, and use of machine learning models in science. *J. Parallel Distrib. Comput.* **2021**, *147*, 64–76. [CrossRef]

56. Romero, F.; Li, Q.; Yadwadkar, N.J.; Kozyrakis, C. INFaaS: A model-less and managed inference serving system. *arXiv* **2019**, arXiv:1905.13348.
57. Li, B.; Samsi, S.; Gadepally, V.; Tiwari, D. Kairos: Building cost-efficient machine learning inference systems with heterogeneous cloud resources. In Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing, Orlando, FL, USA, 20–23 June 2023; pp. 3–16.
58. Kag, A.; Fedorov, I.; Gangrade, A.; Whatmough, P.; Saligrama, V. Efficient edge inference by selective query. In Proceedings of the Eleventh International Conference on Learning Representations, Virtual, 25–29 April 2022.
59. Hu, C.; Li, B. Distributed inference with deep learning models across heterogeneous edge devices. In Proceedings of the IEEE INFOCOM 2022-IEEE Conference on Computer Communications, Virtual, 2–5 May 2022; pp. 330–339.
60. Agarwal, D.; Long, B.; Traupman, J.; Xin, D.; Zhang, L. LASER: A Scalable Response Prediction Platform for Online Advertising. In Proceedings of the 7th ACM International Conference on Web Search and Data Mining, New York, NY, USA, 24–28 February 2014; pp. 173–182. [CrossRef]
61. Crankshaw, D.; Bailis, P.; Gonzalez, J.E.; Li, H.; Zhang, Z.; Franklin, M.J.; Ghodsi, A.; Jordan, M.I. The Missing Piece in Complex Analytics: Low Latency, Scalable Model Management and Serving with Velox. *arXiv* **2014**, arXiv:1409.3809.
62. Wang, W.; Wang, S.; Gao, J.; Zhang, M.; Chen, G.; Ng, T.K.; Ooi, B.C. Rafiki: Machine Learning as an Analytics Service System. *arXiv* **2018**, arXiv:1804.06087.
63. bentoml.com. BentoML Documentation. Available online: https://docs.bentoml.com/en/latest/ (accessed on 26 February 2024).
64. Marozzo, F.; Orsino, A.; Talia, D.; Trunfio, P. Edge computing solutions for distributed machine learning. In Proceedings of the CBDCom 2022 (International Conference on Cloud and Big Data Computing), Falerna, Italy, 12–15 September 2022; pp. 1–8. Available online: http://cyber-science.org/2022/assets/files/program_agenda.pdf (accessed on 3 April 2024)
65. Kakati, S.; Brorsson, M. WebAssembly Beyond the Web: A Review for the Edge-Cloud Continuum. In Proceedings of the 2023 3rd International Conference on Intelligent Technologies (CONIT), Hubli, India, 23–25 June 2023; pp. 1–8.
66. Li, B.; Gadepally, V.; Samsi, S.; Tiwari, D. Characterizing multi-instance gpu for machine learning workloads. In Proceedings of the 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Lyon, France, 30 May–3 June 2022; pp. 724–731.
67. ForestFlow Authors. ForestFlow Documentation. Available online: https://forestflow.github.io/ForestFlow/ (accessed on 26 February 2024).
68. Jolibrain. DeepDetect Server Documentation. Available online: https://www.deepdetect.com/server/docs/server_docs/ (accessed on 26 February 2024).