On-Demand Distributed Artificial Intelligence Is All You Need

Rafael Pereira[a], Daniel Carreira[a], Fábio Gaspar[a], Carla Mendes[a] and Carlos Costa[a]

[a]Computer Science and Communications Research Centre, School of Technology and Management, Polytechnic of Leiria, 2411-901 Leiria, Portugal

ARTICLE INFO

Keywords:
Distributed Artificial Intelligence
Edge Devices
Kubernetes
Orchestration

ABSTRACT

Artificial intelligence (AI) has experienced remarkable growth, transforming various industries and tasks. However, training complex artificial neural networks with large datasets remains a challenge, especially in resource-constrained environments with low hardware capabilities. In this study, we introduce a platform that harnesses the power of distributed artificial intelligence to enable the training of such networks on edge devices with limited hardware resources. By incorporating Kubernetes (K8S), a container orchestration system, we ensure efficient resource management and workload distribution, optimizing the utilization of available hardware. Our platform empowers end users to train and deploy Artificial Intelligence (AI) models directly on edge devices, eliminating the need for extensive computational infrastructure and opening up new possibilities for deploying intelligent applications in resource-constrained settings, bringing innovation and efficiency.

1. Introduction

AI signifies a pivotal shift in the technological era, revolutionizing the way we process, analyze, and use data. However, AI can often be a computationally expensive process, primarily due to the sheer volume and complexity of the data it handles, compounded by the intricacy of the models employed [4].

Due to the rising prices and duration needed for such intensive computational processing, the issue of having a single computer dedicated to training complicated AI models combined with ever-expanding datasets is becoming financially and temporally unpractical [16]. In response, it is possible to cut down on both times and cost by distributing and optimizing model parameters over several devices. The training process may be more economically viable and efficient thanks to this method, which enables the dispersion of financial and computational burdens. Additionally, parallel processing is made feasible by utilizing the resources of several devices, increasing productivity. In essence, a distributed machine learning strategy can assist in effectively managing the expanding quantity of data and complexity of models in a more economical and effective manner [10].

Distributed Artificial Intelligence (DAI) represents a solution to manage the computational expenses of AI, effectively distributing the heavy processing load across multiple devices/nodes. These nodes can include smartphones, Internet of Things (IoT) devices, computers, or other edge devices [12]. Instead of centralizing all these computationally intensive processes, the Machine Learning (ML) processes are performed on multiple devices, using partitioned versions of the original dataset. This reduces individual device performance requirements and enables the creation of smaller, manageable 'micro-models' that together contribute to a larger, more comprehensive model. Each device processes its assigned data, creating a micro-model that is subsequently sent to a central device or server. This server aggregates these micro-models, merging them to form an improved global model. This process repeats in an iterative cycle, with models constantly transmitted between the central server and the devices, facilitating the global model's training on the collective knowledge of all participating devices [21].

Techniques such as DAI offer remarkable benefits in terms of efficiency and scalability. By harnessing the power of distributed computing, DAI can manage vast datasets that could otherwise be time-consuming and resource-demanding. Moreover, DAI enables the training of models on devices with restricted computational abilities or unstable network connections, making it an apt choice for edge computing scenarios [21]. This transformative approach opens up new horizons for scalable, efficient, and distributed AI, providing a robust framework for future IoT systems.

In DAI, the AI tasks such as training and inference are parallelized or distributed among multiple agents/processes which are either located inside a single machine or across multiple machines where the main two parallelization paradigms are data and model parallelism. However, to effectively implement and manage these DAI tasks, containerization technologies play a crucial role. By utilizing containerization technologies, AI tasks can be encapsulated into lightweight, isolated containers, ensuring consistent and reproducible execution across different environments. This containerization approach simplifies the deployment and scalability of AI applications, allowing them to be seamlessly moved between various computing resources. Additionally, configuration optimization using for instance K8S further enhances the management of distributed AI workloads. Through containerization with Docker and configuration optimization with K8S, DAI systems can achieve efficient parallelization and

*This document is the results of the research project funded by the National Science Foundation.

**The second title footnote which is a longer text matter to fill through the whole text width and overflow into another line in the footnotes area of the first page.

ORCID(s): 0000-0001-8313-7253 (R. Pereira); 0000-0001-8785-6001 (D. Carreira); 0000-0001-5589-9904 (F. Gaspar); 0000-0001-7138-7124 (C. Mendes); 0009-0002-1224-5761 (C. Costa)

https://www.linkedin.com/profile/view?id=danielcarreira (D. Carreira), https://www.linkedin.com/profile/view?id=fabiogaspar11 (F. Gaspar), https://www.linkedin.com/profile/view?id=carla-mendes-5b3586233 (C. Mendes)

distribution of AI tasks, leading to improved scalability, resource utilization, and overall performance [5].

Containerization, specifically using technologies like Docker [1], simplifies the deployment and management of applications. It involves encapsulating an application and its dependencies into a lightweight, isolated container that acts as a standalone unit that can run consistently across different environments, ensuring that the application behaves the same way regardless of where it is deployed. Containerization provides advantages such as improved portability, scalability, and reproducibility of applications.

K8S, an open-source system for managing containerized applications across multiple hosts, provides mechanisms for deployment, maintenance, and scaling. It consists of various components related to resource management, such as services, containers, pods, and nodes [18]. Services act as internal load balancers, enabling the deployment of services that route requests to different backend containers. Containers, facilitated by technologies like Docker, package application components into self-contained units that share network space, process space, and file system. Pods, the basic scheduling units, consist of one or more containers colocated on a host machine, allowing resource sharing and communication via unique IP addresses [18, 17].

K8S follows a master-worker node model, with the master node acting as the primary contact point and orchestrator of the cluster. Worker nodes host pods and possess specified hardware capabilities. The scheduler assigns pods to nodes based on resource availability, ensuring optimal utilization without exceeding limits. The K8S framework handles load balancing of service requests, although layer-4 load balancing has limitations for modern applications. To overcome this, additional load balancers like *Linkerd* and *Istio* can be employed, providing content-aware layer-7 load balancing [18].

The popularity of K8S as the leading container orchestrator among service providers underscores its effectiveness in streamlining the entire lifecycle management of containerized applications. From deployment to scaling and maintenance, K8S excels in orchestrating the various aspects of container management, and its comprehensive set of features and robust ecosystem have solidified its position as the go-to solution for container orchestration in the industry [18, 17].

The primary objective of this project revolves around capitalizing on the potential of distributed computing to enhance the speed and efficiency of machine learning operations. The plan is to create a system that uses a network of interconnected devices, optimally distributing computational tasks to accelerate the learning process. The goal is to simplify the intricacies of distributed training, enabling users to focus more on creating and optimizing their model architectures. By adopting a distributed approach, the project aims to train models locally, based on sample patterns, and aggregate them, resulting in a global model.

The main contributions of this research are as follows:

- **Efficient Resource Allocation**: The solution promotes optimal use of resources by matching computational resources with the specific needs of individual models. If a model only needs a 'T4' GPU, we will not allocate an 'A100' GPU.
- **Dynamic Adjustment**: The system dynamically measures and adjusts the allocated resources during the training process to meet the actual requirements, fostering adaptability and efficiency.

The remainder of this paper is organized as follows: in Section 2 we will review existing literature regarding DAI implemented with K8S for AI model's training and aggregation and Federated Learning (FL). Section 3 details the architecture of our proposed system for organized data processing, micro-model training, and model aggregation. The description of our implemented solution describing the existing modules is detailed in Section **??**. Section 5 outlines the system's testing regarding usability and availability, load testing, and DAI evaluation when compared to having a single device for training and comparing multi versus single nodes in model training. Lastly, Section 6 will entail the conclusions and future work of this paper.

2. Related Work

In this section, we present a comprehensive review of the existing literature related to DAI some implemented using K8S. However due to the scarcity of articles, we also present papers describing implementations of FL, a DAI technique, with K8S. By examining the work of previous researchers in these domains, we aim to identify the key advancements, challenges, and potential research directions in each area.

The work in [6] presents a novel Kubernetes Container Scheduling Strategy (KSCC) based on AI to enhance decision-making processes and optimize container scheduling and load balancing. By utilizing AI, this approach improves the efficiency of scheduling and reduces costs by considering multiple criteria in node selection. Unlike existing techniques that rely on individual terms, the KSCC system provides a broader picture of the cloud's condition and user requirements, allowing for informed scheduling decisions. It also enables the utilization of fractional and multiple nodes of Graphics Processing Units (GPUs) for distributed training. Additionally, this research highlights the role of AI in improving data mining, analysis, and model quality.

The work in [14], focuses on DAI in the context of improving criminal investigations. To address this challenge, the study proposes a FL pipeline that promotes the sharing of artificial intelligence models among government institutions. This approach takes advantage of high-security networks and computational resources available within governmental institutions and compares them to the centralized versions of the algorithms.

The work in [20] addresses a fully automatic ML platform, which uniformly manages server resources and allows users to describe their resource requirements through config-

[1] Docker official page: https://docs.docker.com/

uration files. By automating AI task allocation and scheduling based on cluster load, the platform improves resource utilization and load distribution. It also simplifies model configuration and release processes, enabling researchers to focus on model adjustments. K8S provides additional solutions by implementing operating system virtualization and automating task assignment and restarting.

The work in [15] focuses on the integration of cloud computing into the ML life cycle in drug discovery, how cloud computing, specifically through containerization, scientific workflows, and the concept of MLOps, can address these challenges and facilitate reproducible and robust ML modeling in drug discovery organizations. It emphasizes the benefits of cloud infrastructures, such as elasticity and flexibility, in enabling scalable access to computational resources. Additionally, the study explores the utilization of cloud environments for working with private, sensitive, and regulated data, as well as the potential for collaborative drug discovery efforts within and between organizations. By leveraging cloud computing, this study aims to enhance the ML life cycle in drug discovery and improve overall productivity and efficiency in model training and deployment.

The study [9] defines the challenges of managing resource allocation and execution lifecycle for Distributed Deep Learning (DDL) training jobs in shared and cloud resource environments. It focuses on enhancing the K8S platform by developing a scheduling and scaling controller. The objectives of the author's approach are to enable task dependency-aware gang scheduling, locality-aware task placement, and load-aware job scaling. The study evaluates the approach using a real testbed and simulator with TensorFlow jobs, demonstrating improved resource utilization. The need for distributed training in deep learning is emphasized due to the resource-intensive nature of large-scale models, and the study aims to optimize performance by managing resource allocation and the lifecycle of distributed training jobs within the K8S ecosystem.

The paper [19] delves into the subject of DDL (DAI branch) training using Kubeflow, which is a platform built on top of K8S. The study focuses on existing schedulers used for DDL training on K8S, such as DRAGON and SpeCon. Therefore, this paper introduces an innovative approach that harnesses container migration and gang scheduling to enhance the efficiency of DDL training on K8S-based systems. The proposed scheduler demonstrates its efficacy in improving training time while upholding high levels of accuracy.

The authors in [2] explore DDL using Kubeflow on top of Kubernetes and the authors propose a combined scheduler that integrates the approaches of DRAGON and OASIS, introducing weighted autoscaling and gang scheduling, similarly to the article presented previously. Modifications are made to DRAGON's autoscaling function to enhance training efficiency. Experimental evaluation on a set of TensorFlow jobs demonstrates a training speed improvement of over 26% compared to the default Kubernetes scheduler.

Because there is a limited number of articles that introduce the implementation of DAI using K8S, the subsequent papers concentrate on a related subject concerning DAI, specifically, FL - a technique for DAI that is similar to the approach suggested in this paper. FL, also known as Collaborated Learning (CL), is a method that aims to create an algorithm using multiple devices connected to a central server. In this approach, each device keeps its data locally without sharing it with others, ensuring privacy protection for sensitive information like personal bank account details (mentioned in Section 9.2). This decentralized concept focuses on learning from different datasets to create a shared global model for all devices. Instead of sharing data, federated learning exchanges model parameters between the global model and individual local models, allowing for efficient learning without compromising data privacy [22].

The authors in [1] focus on improving the performance and quality of FL models in the context of IoT devices. The authors propose a novel approach called FedAUR, which leverages the concept of FL in conjunction with K8S to enhance the selection of clients, distribution, and allocation of resources for FL, where the purpose is to discover and train in each round the maximum number of samples with the highest quality to target the desired performance. The authors propose a K8S-based prototype to evaluate the performance of the proposed approach where Graphics Processing Unit (GPU) chunks are assigned to the pods inside a node, overall this approach obtained an improved learning accuracy when compared to other baselines [1].

This paper aims to improve the convergence of FL models by enhancing device availability and integration in the learning process. Existing studies have suggested client selection techniques to enhance convergence and accuracy, but they haven't addressed the flexibility of deploying and selecting clients in different locations and times. To tackle these limitations, the paper introduces an approach called On-Demand-FL. It utilizes containerization technology like Docker to create efficient environments, leveraging IoT and mobile devices as volunteers. K8S is employed for orchestration, and a Genetic Algorithm (GA) optimizes client deployment to solve a multi-objective optimization problem. Experimental results using the Mobile Data Challenge (MDC) dataset and the Localfed framework demonstrate the effectiveness of the proposed approach. It enables dynamic deployment of clients as needed, leading to fewer discarded rounds and increased data availability for learning [3].

This article presents a practical framework for FL that utilizes container-related technologies such as Docker, K8S, and Prometheus. These technologies are used to facilitate model deployment, aggregation, and client device monitoring. By leveraging these tools, the framework enables stable model distribution, load balancing, and resource-aware FL. The framework consists of a centralized server and distributed devices, with key modules including the "Device Manager," "AI Model Repository", and "Data Visualizer". The Device Manager monitors device state and resources, providing a Representational state transfer (RESTful) Application Programming Interface (API) for device registration and resource-aware FL. The AI Model Repository handles

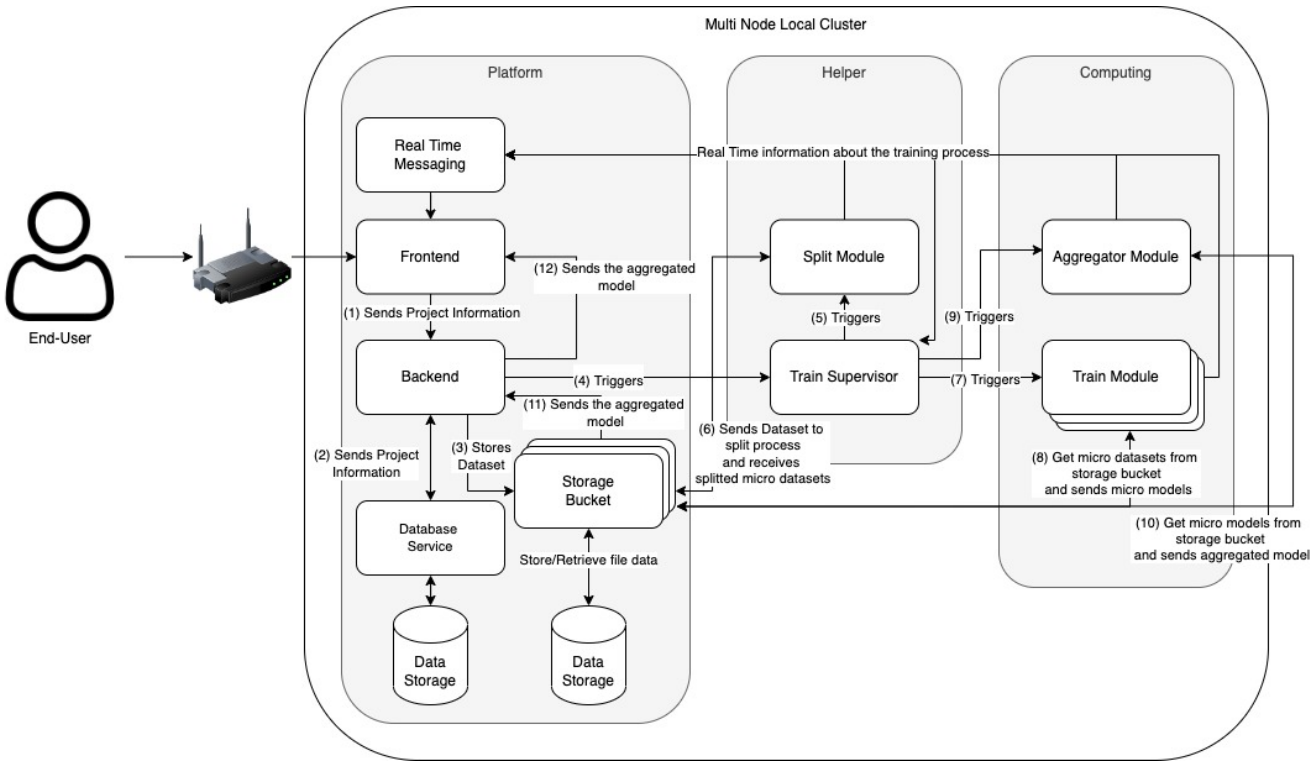

Figure 1: Proposed system architecture.

the distribution and aggregation of Artificial Neural Network (ANN) models to target devices. The central server and devices are organized into a K8S cluster, allowing Docker containers to be deployed and controlled through scheduling or API commands. Overall, this microarchitecture service enhances the effectiveness and efficiency of FL in practical applications by enabling parallel processing and load balancing during model training [13].

The authors in [7] propose and develop KubeFL, a K8S enabled FL Platform that leverages cloud-native technologies such as containers and K8S. KubeFL facilitates the deployment of FL across multiple clients using Docker containers and K8S. This approach, implemented using PyTorch, is deployed on a Docker container within a Pod on each client (Node in K8S), while a corresponding implementation is also conducted on the server (Master in K8S). The authors evaluate the performance of KubeFL through extensive measurements on commercial NVIDIA Jetson TX2 edge devices, considering various practical configurations [7].

3. Architecture

The architecture of our proposed system is a roadmap to creating an effective platform for conducting ML operations. The architecture provides the system's backbone, laying the groundwork for organized data processing, micro-model training, and model aggregation, maintaining the end-user updated with feedback results from the whole process.

As shown in Figure 1 the proposed system architecture shows that the process begins when an end-user interacts with the front-end interface (1) to initiate a new AI project. The frontend transmits this information to the backend (2), where it is subsequently stored in the database. Concurrently, the necessary dataset for the project is secured in a designated storage bucket (3), ready to be accessed by the various system services and modules.

The backend activates the 'Train Supervisor' service (4), a critical component responsible for directing the entire training process. All the modules in the system maintain communication with a real-time messaging service, providing continuous feedback about the infrastructure and its operations.

One of the primary tasks undertaken by the Train Supervisor is the activation of the 'Splitting Module' (5). This module partitions the dataset into smaller, manageable batches which are then stored in the storage bucket (6). The Train Supervisor also initiates the creation of an equal number of 'Training Modules' (7), each corresponding to a split batch.

Each Training Module retrieves its respective dataset batch from the storage bucket (8), commencing the micro-training process. The micro-model generated from this process is then returned to the storage bucket. Upon the completion of all training processes, the Train Supervisor triggers the 'Aggregator Module' (9). This module retrieves all the micro-models from the storage bucket, combining their weights into a unified, final model (10).

This final model is then stored back in the storage bucket, ready to be accessed by the end-user via the frontend upon request. The model's journey to the user starts from the storage bucket (11), then to the backend, and finally to the frontend (12).

The architecture's components are grouped into three main categories: the 'Platform', consisting of the front-end, back-end, database service, storage bucket, and real-time messaging service; the 'Helpers', comprising the Train Supervisor and the Splitting Module; and the 'Computing Modules' that include the Training and Aggregator Modules. The distribution of nodes in the cluster is determined based on these groupings.

3.1. Platform

The platform consists of four pods: frontend, backend, storage bucket, and database. It provides a user interface, project persistence, and real-time messaging. As the workload increases, more pods will be added to ensure high availability.

Frontend. Each frontend pod will serve a static page that will interact with the backend service, and the real-time messaging, allowing users to create AI projects along with their submitted datasets used for the training process. Dashboards of the project will be constructed with information regarding the process of the project from start to end.

Backend. Each backend pod is responsible for providing an API service, allowing it to handle multiple clients concurrently. All the project details, including training tasks, will be persistent in the database directly. Additionally, the files will be handled by a bucket service.

Database. The database stores the training results obtained during the AI training process. These results can include metrics, performance evaluations, and any other pertinent information regarding the trained AI models. By persistently storing the training results, the database enables subsequent analysis, model comparison, and overall evaluation of the training process. The system utilizes a NoSQL database as the storage solution for crucial data, encompassing user information, training task metadata, projects, and training results.

Asynchronous communication protocol. The asynchronous communication protocol plays a crucial role in enabling real-time communication, ensuring the availability of up-to-date data for visualization on the dashboard regarding the training process. Through this communication protocol, the split, training supervisor, aggregator, and train components will establish connections with dedicated project rooms, specifically assigned to each project. This communication makes it able to emit project-related data, which the frontend dashboard will consume resulting in an actively monitor of the project's configurations and training progress in real-time by the users.

Storage bucket. The storage bucket has a crucial role in preserving important files throughout the entire lifecycle of each project, including storing datasets and trained models. The storage bucket acts as a reliable repository for storing and distributing datasets to the split processes, as well as receiving the split datasets from that processes. Once each node finishes the training process, the storage bucket stores the corresponding micro-models and transmits them to the aggregator component. Subsequently, the aggregated model will be returned to the bucket once the aggregator model is finished. This final aggregated model will further be available to access from the frontend page through an endpoint in the backend, enabling users to retrieve the desired information effectively.

3.2. Distributed Learning

Distributed Learning is facilitated through four essential modules: the Train Supervisor, the Split Module, the Train Module, and the Aggregated Module.

Train Supervisor The train supervisor module assumes the pivotal role of overseeing the entire process, right from the dataset's creation and importation by the user. This module takes charge of creating all subsequent modules, including the dataset splitting module, training modules, and aggregation module. It is crucial for the training supervisor to possess the knowledge of when and how each module should be initialized.

For instance, it ensures that the training process cannot commence before the dataset has been properly split for each training session. Similarly, the aggregation of machine learning models cannot take place until all training processes have been successfully completed. By enforcing these prerequisites, the train supervisor module ensures a systematic, automatic, and efficient workflow throughout the entire process.

Split The split modules are a crucial component responsible for handling dataset splitting within the project. One of the primary objectives of this project is to distribute the workload among all available nodes effectively, leading to enhanced efficiency and faster results, without compromising accuracy. The split module plays a vital role in achieving this objective by dividing the imported zip dataset file provided by the user into multiple parts.

By performing dataset splitting, we ensure that no single node trains a machine learning model with insufficient or imbalanced data compared to others. This division of the dataset into suitable parts enables each node to work with a representative and diverse subset of the data, promoting better model training and performance based on a ratio-based approach to determine the number of splits required for the dataset.

Train The training modules are responsible for training machine learning models using the pre-split dataset provided by the split module. These modules encompass a wide variety of convolutional neural networks (CNNs) commonly employed for image classification, alongside various data transformation techniques such as data augmentation and normalization to achieve improved results.

Aggregator The training modules generate multiple micro models through distributed learning, making it imperative to enhance the collective knowledge for improved overall performance. In pursuit of this objective, the aggregator module plays a vital role by combining and aggregating the micro models produced by the training modules.

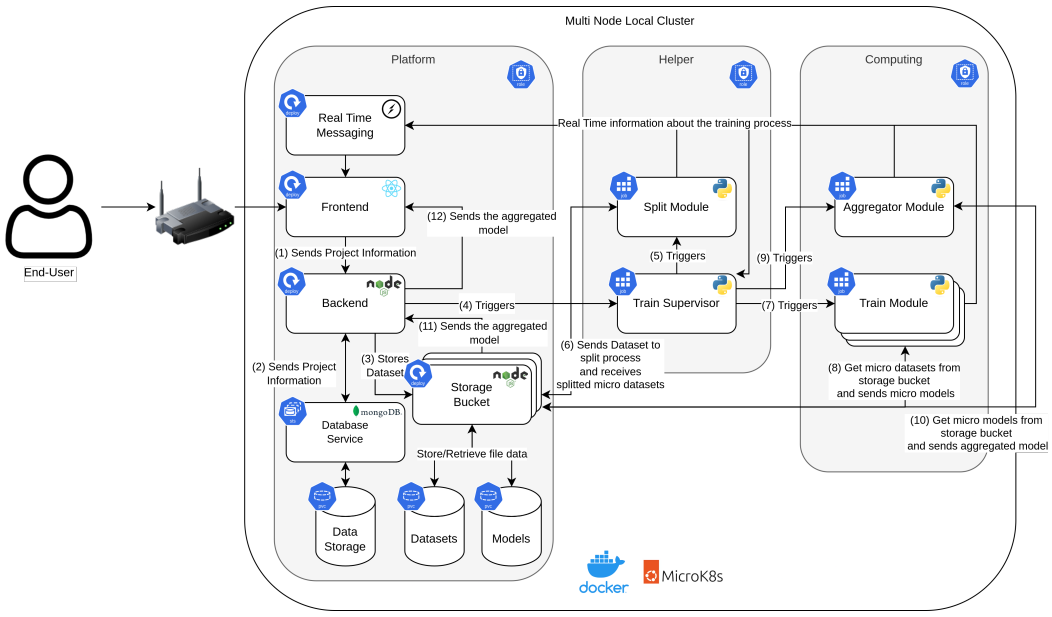

Figure 2: Proposed system's implementation

4. Implementation

The section 3 outlines our proposed architecture, characterized by a cluster-based approach, which is facilitated by K8S and MicroK8s. Figure 2 provides a comprehensive visualization of this implementation, highlighting numerous modules explained in this section. These modules work together to attain a more advanced solution, outperforming traditional implementations, as substantiated in section 5.

A K8S cluster contains several modules that in the current project report serve a distributed AI solution. The 'Config Maps' and 'Persisting Volume Claim (PVC)', acting as the system's memory. They preserve configuration information and guarantee data persistence, respectively. While "Horizontal Pod Autoscaler (HPA)" dynamically adjusts the system's capacity based on workload requirements, "Deployments" grants constant pod uptime. A pod is the basic compute unit in K8S context, which is a container.

'Ingresses' provide a communication link between the cluster and the outside world in the meantime. The 'Jobs' manage specific tasks as the cluster's taskmasters, and the 'Services' facilitate communication between the cluster members. Roles and Labels provide organization to the cluster, and Stateful Sets control the deployment and scaling of pods to ensure a dependable system operation.

The 'Platform' components correspond to our frontend, backend, database service, storage container, and real-time messaging service. The Train Supervisor and the Splitting Module are represented by the "Helpers," while the Training and Aggregator Modules are symbolized by the "Computing Modules." This alignment denotes a smooth integration between the architecture of our system and its actual implementation, bringing the system to life as we had imagined it.

We shall examine the 'Platform' and the function of 'Distributed Learning' in our implementation in further detail in the following subsections.

4.1. Microk8s Kubernetes Cluster

For our cluster, we decided to employ MicroK8s as our Kubernetes cluster solution. Our choice was guided by our requirement for a local, multi-node solution that could be efficiently run on individual devices. Several options were evaluated, such as Minikube, but they fell short of meeting our specifications. MicroK8s, however, with its lightweight and robust design, provided the perfect fit for our needs.

Once the cluster was up and running with MicroK8s, we proceeded to enable several add-ons to further enhance our system's capabilities. The first of these was DNS, which enables service discovery within the cluster, reducing the complexity of service communication. Next, we enabled ingress, which manages external access to the services in the cluster, through HTTP, and WS. This addition allowed to control how and where the services can be accessed from outside the cluster.

Storage was another crucial add-on we activated. This provided persistent volume support, critical for data persistence in stateful applications and other crucial modules, ensuring that data is not lost when pods are replaced or restarted. Lastly, we enabled the dashboard add-on, providing a graphical user interface for the cluster, allowing to manage and monitor the cluster more intuitively.

Together, these add-ons have not only made our cluster more efficient and reliable but have also greatly simplified its management and monitoring. They have thus played a key role in operationalizing the architecture we outlined in the previous section.

4.1.1. Cluster Organization

The K8S implemented architecture uses roles and labels to identify different components of the system, for instance, the "Platform" role, and "Helper" and "Computing" labels. The "Platform" role is responsible for deploying the frontend, backend, real-time messaging, database service, and storage bucket. While this approach simplifies the implementation process, it presents a potential issue: if a node capable of running these modules is unavailable, the deployment process will halt, leading to future problems.

To address this concern, we have adopted a different approach for the other implemented modules, namely node affinity. Node affinity offers a more flexible solution by allowing a preference for specific nodes but not depending solely on them. In cases where a node with the required label for a particular module is not available, the system is designed to create and deploy the module on another available node, ensuring the smooth operation of the overall system.

4.1.2. Ingresses

In our K8S cluster, ingresses play an instrumental role by managing external access to the services within the cluster. We have strategically set up three unique ingress routes, each tailored to a specific purpose:

The first ingress is designed exclusively for the frontend. Configured with a resource path of '/', it handles the traffic from the root URL, directing it to our frontend service. This ingress is void of any supplementary restrictions. It promotes a direct user-interface interaction, enabling users to seamlessly navigate and utilize our application via their web browsers.

The second ingress, assigned to our backend, is characterized by a resource path of '/api'. It is therefore designated as the primary conduit for all API-related traffic. It features a rewrite rule for '/', which essentially reroutes all incoming traffic intended for the root URL to our backend service. To prevent the backend from being overwhelmed with excessively large requests, we have implemented a size limit, capping the maximum content size at 200Mb. This backend-specific ingress plays an essential role, given that our frontend service, operating within the K8S cluster, relies heavily on this exposure for its proper functionality.

The final ingress, designated for real-time messaging, facilitates instantaneous bi-directional communication between our frontend and all the other modules operating within the cluster without a dedicated ingress. This ingress operates via a '/socket.io' resource path, complying with the specifications of the socket.io real-time communication service. Unlike the other ingresses, this one incorporates Cross-Origin Resource Sharing (CORS) and specific communication protocol rules. Such a configuration ensures a quick, hassle-free communication exchange by our real-time engine, providing users with instantaneous updates.

4.1.3. Services

Services in a K8S cluster provide a reliable and stable means of communication between various pods, acting as the key interface between several components of our system. Only the modules that require external exposure or need to establish intra-cluster communication have dedicated services.

The services in the cluster are related to the frontend, backend, socket-service, database, and bucket service deployments. These services ensure that these modules can be efficiently accessed and communicated with, either by end-users or other pods within the system.

The services are configured as ClusterIP type services, a choice influenced by the fact that the cluster is hosted locally. ClusterIP services are the default K8S service, and they provide a service inside the cluster that other pods can communicate with. This choice allows for secure, internal network communication within our K8S cluster. A future and closely ended iteration of this project would use NodePort configuration to restrict accesses from unauthorized sources.

Furthermore, these services are discoverable between pods owing to the DNS resolution enabled in the cluster configuration. This functionality translates service names to their corresponding ClusterIPs, enabling straightforward communication between services without needing to track individual IP addresses. With DNS resolution, pods can communicate using service names, enhancing the system's overall robustness and flexibility.

4.1.4. Deployments, StatefulSets, and Jobs

Deployments and StatefulSets are essential in a K8S environment for guaranteeing the continuous operation and availability of pods. Our solution uses StatefulSets or Deployments to represent the frontend, backend, real-time messaging system, bucket, and database components. They are immediately applied to the cluster at startup, guaranteeing that these crucial system parts are consistently available for use.

Our K8S cluster's jobs play a crucial part in the system's operation, dynamically initiating several processes. Jobs are intended to continue until they are successfully completed, unlike deployments and stateful sets, which aim to keep pods operating continually. If a task fails, the Job will keep retrying it until it succeeds or fails a predetermined number of times. The split, train, aggregator, and train supervisor are some of our job-defined components.

As soon as a project is created on the platform, these jobs are started in a chain-like fashion. To connect to the cluster internally, K8S clients are used, which enable programmatic interaction with the K8S API. This configuration enables us to programmatically apply a variety of functions to the K8S cluster, boosting the efficiency and fluidity of our operations.

4.1.5. Persistence Volumes

The implementation approach began with the utilization of three PVCs: one for datasets, one for models, and one for MongoDB. During the development phase in Multi-Node Local, it was observed that when multiple Pods attempted to mount the PVC as a ReadWrite volume, only the first Pod succeeded. The remaining pods had to wait until the PVC was unmounted, which occurred upon the Pod's termination. Upon closer examination of the microk8s PVC specification,

it was discovered that it allowed multiple ReadWrite mounts only if they were all on the same node, a condition that is not possible to ensure.

Another approach taken into consideration was creating multiple PVCs to mitigate the blocking issue. Each dataset, micro dataset, and model would have been assigned its own PVC. However, this division could not be implemented due to the concurrency locks in place, even with the utilization of a ReadWrite lock and a separate Read lock.

Finally, the approach used in the project involved centralizing the accesses through a single entity, the bucket service. This entity provides an API for storing and retrieving resources in two primary PVCs: Datasets, and Models. The Datasets PVC contains files related to the dataset, including the original .zip file, test dataset .zip file, and the micro datasets generated after splitting. On the other hand, the Models PVC is dedicated to storing the results of both partial and aggregated training.

4.1.6. Runtime Clients and Automated Docker Image Process

In order to interact with the K8S API, the system leverages the power of the Python[2] and JavaScript[3] K8S client libraries. These libraries allow the services to directly communicate K8S cluster control plane efficiently, making use of its robust API to manage the lifecycle of the jobs and services.

The JavaScript K8S client is crucial in deploying the "train supervisor" job. This job is crucial for triggering pods related to the project training process. The train supervisor monitors the state of the training jobs, reacts to changes, and updates the end-user through the real-time messaging, backend, and frontend system as necessary.

The Python K8S client is used by the "train supervisor" job. It triggers a series of subsequent jobs - namely the dataset splitter, micro training jobs, and the aggregator job. This chain of events helps to ensure that our system follows a methodical workflow, with each job performing a specific task and contributing to the overall functionality of the system.

Docker Hub is our choice for storing and accessing the built container images, thereby ensuring efficient and rapid deployment of our services. To keep our Docker images up-to-date, it was implemented an automated deployment process using GitHub Actions. Every time a new commit is made to our repository, GitHub Actions triggers a workflow that builds a new Docker image and pushes it to Docker Hub. This automated process saves time, reduces the potential for human error, and ensures that the service deployment is always running on the latest and most stable version of the codebase.

4.2. Platform

Frontend: The platform features a web interface built with the React framework enabling users to interact with the system. Users can conveniently access and manage their projects, view relevant information, and perform necessary actions through this interface.

Backend: The backend of the platform is developed using Node.js in conjunction with the Express library. It is responsible for handling the communication between the frontend and the database service. Additionally, the backend triggers the creation of a train supervisor pod which is responsible for the creation of pods that ensure the execution of the user requests.

Database service: The platform employs a MongoDB database to store information related to each user's projects and their respective measures.

Real-time messaging: This service utilizes a Node.js WebSocket server with Socket.IO to transmit real-time updates from all the different pods within the system. It plays a critical role in ensuring that the frontend is constantly updated with the latest information, ranging from logs to model accuracy.

All of the mentioned modules constantly submit logs across the real-time messaging system to actively communicate their current states so that users can simply check on ongoing operations via the online interface.

Storage bucket: A dedicated service is utilized to store the datasets imported by users for their projects, as well as the models they create. Two separate Persistent Volume Claims (PVCs) are employed for datasets and models, respectively, ensuring proper organization and storage of these valuable assets.

Horizontal Pod AutoScaling: A crucial aspect of our work is enabling users to train their machine-learning models efficiently without the need for expensive software. However, with the expectation of a large number of users accessing the platform and the resources, the performance of user-accessible platforms could potentially be impacted. To address this challenge, horizontal pod autoscaling (HPA) was implemented.

HPA is a key feature that allows to accommodate the increased user demand. By utilizing this capability, multiple users can simultaneously access the platform through the same user interface (view), while accessing different pods that may be running on separate machines. This effectively distributes the processing workload across multiple devices, ensuring optimal performance even when a high number of users are accessing the service concurrently. By dynamically scaling the number of pods based on demand, the workload can continue being efficient and maintain a responsive and reliable user experience.

4.3. Distributed Learning

Training Supervisor: module implemented in Python with the primary responsibility to initiate and ensure the right pods through the Kubernetes Python library from the resource creation until achieving the final aggregator module.

Split Module handles the division of the imported dataset provided by the user. The dataset is expected to be in the form of a zip file, comprising two folders: one for the train-

[2] https://github.com/kubernetes-client/python/tree/master/kubernetes
[3] https://github.com/kubernetes-client/javascript/tree/master/kubernetes

ing dataset and another for the testing dataset. Within each of these folders, there should be subsequent folders, each associated with a specific class to be classified. The Split module calculates the total number of images within the training dataset and determines the number of splits to create based on the available training pods, thus ensuring that each training pod receives a reliable portion of the data for processing.

Training Module: Python module designed for machine learning training using PyTorch, which is a popular deep learning framework that provides an extensive range of tools and functionalities. Within the training module, there are multiple neural network architectures that are used as the selectable neural networks available in the frontend platform. These networks right now include EfficientNet v2s, ResNet-18, VGG-16, SqueezeNet, and a simple CNN with two convolutional layers.

Aggregator Model: The Aggregator Model is a Python module that also uses the PyTorch framework. Its main objective is to create an aggregated module by combining the micro models generated from the Training modules using a mean-based approach similar to the aggregation process in Federated Learning [11]. The primary aggregation equation employed in this context is:

$$f(w) = \sum_{k=1}^{K} \frac{n_k}{n} F_k(w) \tag{1}$$

where w is the model parameters vector and $f(w)$ represents the overall objective function that is aimed to minimize. Here, K is the total number of clients while n_k is the number of training examples on client k and $n = \sum_{k=1}^{K} n_k$ is the total number of training examples over all clients.

$F_k(w)$ is the local objective function for client k, which is computed as an average of individual functions $f_i(w)$, each of which signifies the loss of the prediction on example i made with model parameters w:

$$F_k(w) = \frac{1}{n_k} \sum_{i \in P_k} f_i(w) \tag{2}$$

In this setup, the weights for the global aggregation of local models are calculated based on the size of the local dataset of each client. During each iteration, micro models are sent to the server for aggregation, and this aggregated model is then shared back with the client.

Following the aggregation process, the final model is tested on the testing dataset and reports the metrics back to the frontend platform. This detailed feedback allows users to analyze the performance of the globally trained model. Moreover, the aggregated model is saved in the bucket, thus ensuring its accessibility for download and further utilization.

5. Experimental Results

This section aims to outline how the system will respond in various scenarios to ensure its usability and availability, therefore this section will detail the performed tests regarding HPA where load test is performed and assessed with the JMeter tool. Moreover, tests comparing the implemented DAI solution with 1 and 3 nodes with the traditional approach where the model training occurs in a single device.

Table 1
Hardware, Operating System, and Libraries Specifications for Test Bed Machines.

Machine 1	
Hardware	
Chip	Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz
Total Number of Cores	12
RAM Memory	16 GB
Software	
Operating System	Fedora 38
Docker	24.0.2, build cb74dfc
MicroK8s	1.27.2 revision 5372
Machine 2 (Virtual Machine)	
Hardware	
Chip	Apple M1 Pro @ 3.2 GHz
Total Number of Cores	5
RAM Memory	4 GB
Software	
Operating System	Ubuntu 22.04
Docker	20.10.21, build 20.10.21-0ubuntu1 22.04.3
MicroK8s	1.27.2 revision 5378
Machine 3 (Virtual Machine)	
Hardware	
Chip	Apple M1 Pro @ 3.2 GHz
Total Number of Cores	5
RAM Memory	8 GB
Software	
Operating System	Ubuntu 22.04
Docker	20.10.24, build 297e12
MicroK8s	1.27.2 revision 5378
Machine 4	
Hardware	
Chip	i5-8250U @ 1.6 GHz
Total Number of Cores	4
RAM Memory	8 GB
Software	
Operating System	Fedora 38
Docker	24.0.2, build cb74dfc
MicroK8s	1.27.2 revision 5372

The tests were conducted using multiple machines, each with its own hardware, operating system, and software specifications. The detailed specifications of the test bed machines are provided in Table 1.

5.1. Horizontal Pod AutoScalling

The purpose of conducting HPA tests is to understand how this method can improve the system's availability within a designated timeframe when faced with increased demand.

We selected the backend service as the primary focus for HPA as it is simpler to perform load tests on. In order to ensure fairness in the tests, we kept the resource allocation of the replica sets fixed at 1 CPU core and 500 MiB of

Table 2
Load Testing without HPA - Connection, Latency, and Rate grouped by Request Status

Request Status	Connect (ms)	Latency (ms)	Rate (%)
Success	13.5	10314.3	2.1
Error	14.7	16152.3	97.8

Table 3
Load Testing with HPA - Connection, Latency, and Rate grouped by Request Status

Request Status	Connect (ms)	Latency (ms)	Rate (%)
Success	391.2	53997.4	40.0
Error	179.0	37378.6	60.0

RAM. For the HPA, we utilized CPU usage as the trigger point, aiming for a target of 75% usage. This means that if the overall load of all the target values reaches 75%, another replica would be created to alleviate the load. These tests run for a total of 10 minutes, giving enough time for the pod to scale.

During the initial testing phase with a single pod on a single node, the results, illustrated in Table 2, aligned with our expectations. When subjected to a pool of 5 000 users generating continuous requests to the service, approximately 97.83% of these requests were rejected, indicating a lack of reliability in the system. Furthermore, the average latency for successful requests was approximately 10 seconds, while error responses experienced an average latency of 16 seconds. If we look at the Figure 3, we can observe the constant delay spikes, hitting in some instances over 125 seconds of latency.

Figure 3: Load Testing without HPA - Latency over Time Graph

In the HPA scenario, we employed 3 of the 4 nodes to receive replicasets when needed, and the results are presented in the Table 3. The number of pods created was variant during the test since the pods are autoscaled based on the live load. However, the test ended with 4 pods orchestrated across the 3 available pods. When looking at the resulting Table 3, the average time to connect increased by a huge margin, around 10 times slower. Further, the latency also increased significantly to 54 seconds in case of success, and 37 seconds in case of error. However, this scalability even so it increased the overall response time, it improved the availability from 98% to 60%, a 38% improvement.

In the scenario involving the HPA, we utilized three out of the four available nodes to accommodate replicasets as needed. The results of this test are presented in Table 3. Due to the autoscaling nature of pods based on live load, the number of created pods varied throughout the test. However, at the conclusion of the test, there were four pods distributed across the three available nodes. Analyzing the elarningdata in Table 3, it is evident that the average connection time experienced a significant increase, becoming approximately 10 times slower. Additionally, the latency also saw a notable increase, reaching 54 seconds for successful responses and 37 seconds for error responses. Despite the increase in overall response time, this scalability improvement resulted in an enhanced availability from 98% to 60%, representing a 38% improvement. Additionally, we can conclude with the help of Figure 4 that the spikes are more prominent after the cluster resize, hitting a max of 180 seconds.

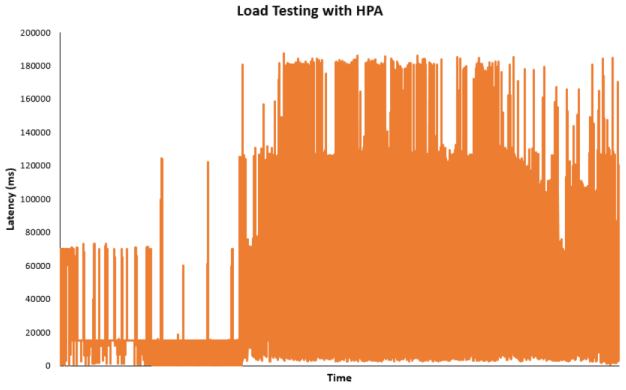

Figure 4: Load Testing with HPA - Latency over Time Graph

5.2. Distributed Learning

In this test, our objective is to demonstrate the benefits of distributing the training process across multiple nodes in terms of time. To achieve this, we have designed two main scenarios. The first scenario focuses on our system utilizing distributed learning, comparing the performance between a single-node setup and a multi-node setup. The second scenario involves Traditional Learning (TL) on a single machine outside the cluster, with two variants: a high-end laptop and a dedicated ML server.

Scenario 1: The cluster setup stayed the same throughout the experimentation, just only targeting Machine 2 (Table 1) in the single-node variant.

Scenario 2: Outside the cluster, we had the same Machine 2 running as the first variant and a ML server with an NVIDIA A100 for the second variant.

In addition to the testing setup, we opted to use the widely

Table 4
Distribution Learning Test - Results (DL - Distributed Learning, TL - Traditional Learning)

Platform	Type	Nodes	Pods	Time (s)
Kubernetes Cluster	DL	1	6	75.4
Kubernetes Cluster	DL	3	6	35.1
Machine 2	TL	-	-	20.4
ML Server	TL	-	-	10.4

recognized MNIST dataset [8] due to its smaller image size of 32 pixels by 32 pixels. Considering the relatively low complexity and size of the dataset, we selected a smaller network architecture with fewer convolutional layers to accommodate it effectively. The train ran for 25 epochs in all the scenarios to grant an equal environment.

Executing the test case presented, we obtained the following Table 4, where the fields include: platform, type of learning, number of nodes, number of pods, and average execution time per epoch. When focusing on scenario 1, the distribution of the pods to different nodes was revealed to be effective in reducing the execution time, by 40 seconds, totaling on average 35 seconds to finish an epoch in each pod. Scenario 2 proved to be the fastest by a significant margin, reducing 15 seconds on top of the best performer of scenario 1. Finally, the most advanced machine for ML task performed the best as was expected, by a margin of 10 seconds.

After executing the presented test case, we obtained results as shown in Table 4. The table includes information such as the platform used, type of learning, number of nodes, number of pods, and average execution time per epoch.

In scenario 1, distributing the pods to different nodes proved to be effective in reducing the execution time, by 40 seconds if compared with the single node. This resulted in an average epoch execution time of 35 seconds per pod. Scenario 2 was even faster, reducing an additional 15 seconds compared to the best performer in scenario 1. As expected, the most advanced machine for ML tasks outperformed the others by a margin of 10 seconds.

6. Conclusions & Future Work

In conclusion, our system harnesses the power of DAI, containerization K8S to address the challenges of training complex ANN on edge devices with limited hardware resources. By incorporating K8S as a container orchestration system, we achieve efficient resource management and workload distribution, optimizing the utilization of available hardware. This empowers end users to train and deploy AI models directly on edge devices, eliminating the need for extensive computational infrastructure.

The use of training pods, along with DAI, enables parallel processing and efficient use of computational power, accelerating the overall training process. The aggregator pod combines the knowledge from each training pod to create a comprehensive AI model that benefits from the collective expertise gained throughout the distributed training. Overall, our platform opens up new possibilities for deploying intelligent applications in resource-constrained settings, bringing innovation and efficiency. By enabling AI training on edge devices with limited hardware resources, we reduce the reliance on extensive computational infrastructure. The seamless communication, fault tolerance, and accelerated training processes achieved through our platform contribute to the advancement of AI in various industries and tasks.

For future work, our primary focus is to enhance privacy and data security regarding the datasets used to train and evaluate the models. By ensuring that data remains on local devices, FL effectively mitigates the risk of sensitive information exposure while driving innovation and collaboration in a secure environment. The integration of our solution with Terraform is another future improvement topic to ensure a cloud implementation of our system, enabling seamless infrastructure provisioning and management. Furthermore, FL provides us with an opportunity to transition our solution to the cloud and enable the gathering of additional nodes from different geographical locations, enhancing the overall performance of the system. The management of the persistent volumes in the storage bucket is yet another future work improvement considering the existing limitation of being restricted to the space of just one node. Various attempts have been made to address this, such as experimenting with an Network File Sharing (NFS) server, but unfortunately, it proved unsuccessful within the given time constraints. As an alternative, we are considering utilizing an external PVC in the cloud to seamlessly integrate it into our solution.

References

[1] S. Abdulrahman, H. Ould-Slimane, R. Chowdhury, A. Mourad, C. Talhi, and M. Guizani. Adaptive upgrade of client resources for improving the quality of federated learning model. *IEEE Internet of Things Journal*, 10(5):4677–4687, 2023.

[2] Muhammad Fadhriga Bestari, Achmad Imam Kistijantoro, and Anggrahita Bayu Sasmita. Dynamic Resource Scheduler for Distributed Deep Learning Training in Kubernetes. In *2020 7th International Conference on Advance Informatics: Concepts, Theory and Applications (ICAICTA)*, pages 1–6, September 2020.

[3] M. Chahoud, H. Sami, A. Mourad, S. Otoum, H. Otrok, J. Bentahar, and M. Guizani. On-demand-fl: A dynamic and efficient multicriteria federated learning client deployment scheme. *IEEE Internet of Things Journal*, pages 1–1, 2023.

[4] Yanqing Duan, John S. Edwards, and Yogesh K Dwivedi. Artificial intelligence for decision making in the era of big data – evolution, challenges and research agenda. *International Journal of Information Management*, 48:63–71, 2019.

[5] N. Janbi, I. Katib, and R. Mehmood. Distributed artificial intelligence: Taxonomy, review, framework, and reference architecture. *Intelligent Systems with Applications*, 18, 2023.

[6] Diaz Jorge-Martinez, Shariq Aziz Butt, Edeh Michael Onyema, Chinmay Chakraborty, Qaisar Shaheen, Emiro De-La-Hoz-Franco, and Paola Ariza-Colpas. Artificial intelligence-based kubernetes container for scheduling nodes of energy composition. *International Journal of System Assurance Engineering and Management*, pages 1–9.

[7] J. Kim, D. Kim, and J. Lee. Design and implementation of kubernetes enabled federated learning platform. volume 2021-October, pages 410–412, 2021. ISSN: 2162-1233.

[8] Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

[9] Chan-Yi Lin, Ting-An Yeh, and Jerry Chou. Dragon: A dynamic scheduling and scaling controller for managing distributed deep learning jobs in kubernetes cluster. In *CLOSER*, pages 569–577, 2019.

[10] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-Efficient Learning of Deep Networks from Decentralized Data. In Aarti Singh and Jerry Zhu, editors, *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, volume 54 of *Proceedings of Machine Learning Research*, pages 1273–1282. PMLR, 20–22 Apr 2017.

[11] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data. *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017*, 2 2016.

[12] S.K. Mishra, K. Sindhu, M.S. Teja, V. Akhil, R.H. Krishna, P. Praveen, and T.K. Mishra. Applications of Federated Learning in Computing Technologies. In *Convergence of Cloud with AI for Big Data Analytics: Foundations and Innovation*, pages 107–120. 2024.

[13] Jongbin Park and Seung Woo Kum. Design and development of server-client cooperation framework for federated learning. In *2022 Thirteenth International Conference on Ubiquitous and Future Networks (ICUFN)*, pages 271–273, July 2022. ISSN: 2165-8536.

[14] Gabriel Souza, Mickael Figueredo, Daniel Sabino, and Nélio Cacho. A pipeline to collaborative ai models creation between brazilian governmental institutions. In *2023 IEEE 15th International Symposium on Autonomous Decentralized System (ISADS)*, pages 1–7, 2023.

[15] Ola Spjuth, Jens Frid, and Andreas Hellander. The machine learning life cycle and the cloud: implications for drug discovery. *Expert opinion on drug discovery*, 16(9):1071–1079, 2021.

[16] Ahmet Ali Süzen, Burhan Duman, and Betül Şen. Benchmark analysis of jetson tx2, jetson nano and raspberry pi using deep-cnn. In *2020 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA)*, pages 1–5, 2020.

[17] Minh-Ngoc Tran, Dinh-Dai Vu, and Younghan Kim. A Survey of Autoscaling in Kubernetes. In *2022 Thirteenth International Conference on Ubiquitous and Future Networks (ICUFN)*, pages 263–265, July 2022. ISSN: 2165-8536.

[18] Gianluca Turin, Andrea Borgarelli, Simone Donetti, Ferruccio Damiani, Einar Broch Johnsen, and S. Lizeth Tapia Tarifa. Predicting resource consumption of Kubernetes container systems using resource models. *Journal of Systems and Software*, 203:111750, September 2023.

[19] Yonatan Viody and Achmad Imam Kistijantoro. Container Migration for Distributed Deep Learning Training Scheduling in Kubernetes. In *2022 9th International Conference on Advanced Informatics: Concepts, Theory and Applications (ICAICTA)*, pages 1–6, September 2022.

[20] Chaoyu Wu, E. Haihong, and Meina Song. An automatic artificial intelligence training platform based on kubernetes. In *Proceedings of the 2020 2nd International Conference on Big Data Engineering and Technology*, BDET 2020, page 58–62, New York, NY, USA, 2020. Association for Computing Machinery.

[21] Chen Zhang, Yu Xie, Hang Bai, Bin Yu, Weihong Li, and Yuan Gao. A survey on federated learning. *Knowledge-Based Systems*, 216:106775, March 2021.

[22] Lan Zou. Chapter 5 - Meta-learning for computer vision. In Lan Zou, editor, *Meta-Learning*, pages 91–208. Academic Press, January 2023.