Journal of Intelligent and Connected Vehicles
2023, **6**(3): 125–135
https://doi.org/10.26599/JICV.2023.9210009

Research Article

Ensuring federated learning reliability for infrastructure-enhanced autonomous driving

Benjamin Acar✉, Marius Sterling

Technical University of Berlin, Berlin, 10623, Germany

Received: March 7, 2023; Accepted: May 10, 2023

ABSTRACT: The application of machine learning techniques, particularly in the context of autonomous driving solutions, has grown exponentially in recent years. As such, the collection of high-quality datasets has become a prerequisite for training new models. However, concerns about privacy and data usage have led to a growing demand for decentralized methods that can be learned without the need for pre-collected data. Federated learning (FL) offers a potential solution to this problem by enabling individual clients to contribute to the learning process by sending model updates rather than training data. While Federated Learning has proven successful in many cases, new challenges have emerged, especially in terms of network availability during training. Since a global instance is responsible for collecting updates from local clients, there is a risk of network downtime if the global server fails. In this study, we propose a novel and crucial concept that addresses this issue by adding redundancy to our network. Rather than deploying a single global model, we deploy a multitude of global models and utilize consensus algorithms to synchronize and keep these replicas updated. By utilizing these replicas, even if the global instance fails, the network remains available. As a result, our solution enables the development of reliable Federated Learning systems, particularly in system architectures suitable for infrastructure-enhanced autonomous driving. Consequently, our findings enable the more effective realization of use cases in the context of cooperative, connected, and automated mobility.

KEYWORDS: federated learning (FL), Kubernetes, autonomous driving

1 Introduction

The field of autonomous mobility systems is experiencing a significant level of interest and investment by governments, corporations, and research institutes. This interest is driven by the potentials of such systems bear to revolutionize both passenger transport and logistics. Recent advancements in computing, sensor, and actuator technology have enabled unprecedented progress in the field of autonomous driving. This has led to a proliferation of research projects, many of which have yielded novel methods, solutions for the development of these systems. One of the key challenges facing the field of autonomous driving is the handling of large volumes of data for the training of machine learning models. The proliferation of devices has led to an increase in the amount of data available for training, which presents exciting opportunities for improving the performance of autonomous mobility systems (Alpaydin, 2021). However, the sheer volume of data also poses significant challenges in terms of handling the data efficiently, effectively, and securely. Additionally, the sensitivity of this data and the growing awareness of privacy concerns further complicate matters by raising questions about model training without infringing the privacy of data owners (Kairouz et al., 2021).

Federated Learning (FL) is a rapidly growing area of research that offers a potential solution to the challenges faced in the handling and processing of big data in the field of autonomous driving. FL is a distributed machine learning technique that allows the simultaneous training of a neural network models across multiple devices, which subsequently compute model parameter updates, forward these to a central server for aggregation and updating of the global model (Yang et al., 2019). Previous research papers have demonstrated the effectiveness of FL in the context of autonomous driving, showing its potential to improve the performance of autonomous systems while addressing privacy concerns (Li et al., 2022; Nakanoya et al., 2021; Nguyen et al., 2022; Zhang et al., 2021). However, FL also has its limitations, particularly in terms of its reliance on a central server for aggregation which poses a risk of a single point of failure, limiting the applicability of server-based FL in certain scenarios (Pokhrel et al., 2020a). Despite this, FL has the potential to revolutionize the field of autonomous driving by enabling the training of models on decentralized data without compromising the privacy of data owners. The decentralized nature of FL allows for the training of models on multiple devices, which can help to improve the performance of autonomous systems while addressing privacy concerns. Our aim is to tackle the single point of failure constraint by incorporating redundancy into the central server, thereby enabling the creation of more dependable and robust FL systems. Our proposed solution will promote the utilization of FL within the scope of our research project, BeIntelli. This project aspires to construct a comprehensive software stack for autonomous driving, along with autonomous test vehicles, a digitized testbed in the heart of Berlin, a mobility data platform, and other objectives. By

✉ Corresponding author.
E-mail: benjamin.acar@tu-berlin.de

integrating smart infrastructure with intelligent vehicles and utilizing the cloud, our project approach will enable the achievement of over-the-horizon perception and prediction of vehicles. This, in turn, will foster the development of cooperative, connected, and automated mobility use cases (Augusto et al., 2021).

The paper is structured as follows: in Section 2, we explore the basics of our concept, starting such as orchestrator technology and Redis databases. In Section 3, we present related work, both from the area of Kubernetes-based FL approaches, and those dealing with the single point of failure problem. In Section 4, we present our own method. In Section 5, we describe the experiment that aims to show that our concept guarantees reliable FL systems. Section 6 shows the results of the experiment in Section 5. Finally, in Section 7, we summarize the contributions and give an insight into our future work.

2 Fundamentals

In the following, we summarize the basic concepts of Kubernetes & Redis. For general information about FL, see, Kairouz et al. (2021), McMahan et al. (2016), and Yang et al. (2019).

2.1 Kubernetes

Kubernetes is an open source orchestrator for deployment of container-based applications, developed, and introduced by Google (Bernstein, 2014) in 2014. The major goal of Kubernetes is the deployment of scalable and reliable systems, consisting of so-called microservices (Google, 2022). In the following, we summarize the basic concepts of Kubernetes. All the information is available at Google (2022), as well as in learning materials such as Burns et al. (2022). For more information on container technology, see Docker (2022).

2.1.1 Orchestration

To understand the concept of orchestration, we introduce several, commonly used terms.

Definition 1. A microservice is a small, self-contained code that serves as a semi-independent application. Several microservices are usually part of a larger unit known as an application. Microservices are deployed in pods. A pod is a single or a group of containers with additional layers, allowing the integration in Kubernetes architecture (Google, 2022).

To differentiate between a pod and an application, consider the following scenario: An application typically encompasses a graphical user interface (GUI) and a backend component. In this scenario, we could deploy a separate pod for each of these components, with the combination of both constituting the application as defined in this study.

Definition 2. A node describes a virtual or physical machine. In Kubernetes, we use master and worker nodes. Master nodes are special because of their responsibility to take care of the overall network. Usually, user-written microservices only run on the worker nodes, whereas components used to deploy Kubernetes itself run on the master nodes (ibid.).

Definition 3. A cluster describes the overall network, consisting of all nodes, cluster components, and microservices. A cluster is managed by the master nodes, which work as control entities (ibid.).

With definitions 1–3, orchestrator can be introduced according to the commonly used definition:

Definition 4. An orchestrator is a software system that orchestrates microservices. Rather than initiating a microservice deployment independently, it is initiated by the orchestrator in an orchestrator-based system. In that instance, the orchestrator searches for an available node with the needed resources to deploy the application. Furthermore, the orchestrator ensures that all deployed applications are available (ibid.).

Kubernetes provides all the components and objects required to deploy such a microservice architecture. These objects are useful for deploying containers, isolating them, configuring load balancing, and various other tasks (ibid.).

2.1.2 Replicas

One of the most important aspects of orchestrators and the main reason for the reliability of such a system is the deployment of replicas (Google, 2022). For the robust execution of microservices, Kubernetes allows the deployment of numerous replicas of the same microservice to be distributed across several nodes. Because of the distributed replicas–even if one of the nodes fails – the microservice remains available and productive (ibid.). If one of the pods fails, Kubernetes automatically manages the full process of rescheduling new pods of the same microservice (ibid.). That makes such a system reliable for both, users and developers (ibid.). Note that even if no replica is set, Kubernetes reschedules failed pods. Nonetheless, there is some downtime due to the gap between failing and restarting of the microservice. To reduce downtime, deploying replicas is a (ibid.).

In the following, two fundamental terms in the context of replicas are introduced, see (ibid.): stateful and stateless applications.

Definition 5. Stateful applications are applications that retain the current state of some data by storing it (Hausenblas, 2022). The most common example of such an application is a database, e.g., MySQL, PostgreSQL, and Redis (Google, 2022).

Definition 6. Stateless applications are applications that do not track any state (Hausenblas, 2022). Whenever these applications are accessed, the accesses are thus unrelated and independent requests or queries. Such applications have a pure functional behavior (Google, 2022). An example of such an application would be a GUI.

In a typical infrastructure, there are both: stateful and stateless applications that may interact with each other. A straightforward example is a GUI (stateless) that displays the current view of a database (stateful). Distinguishing these two types of applications is critical for Kubernetes deployments. To understand the significance, we assume in the following example that there is a stateful application, such as a database. Further, we assume that two basic database replicas are deployed, pod A and pod B. The principal pod, to which the user has accessed, is pod A. If the user changes some entries in the database, only pod A tracks these state changes. Whenever pod A fails, the current state of the database is lost. The redirection to pod B ensures the availability of the database but not the timeliness of the data (ibid.). To solve that issue, Kubernetes provides different deployment strategies for different sorts of applications.

Definition 7. To create replicas of stateless applications deployments are used. A deployment is a higher-level concept that manages ReplicaSets. ReplicaSets are used to maintain a set of replicas of stateless applications. For ReplicaSets all replicated pods are identical and interchangeable.

Fig. 1 illustrates deployments. Pods that are deployed within a deployment are identical. As a consequence, the pods are interchangeable without changing the overall state (Google, 2022).

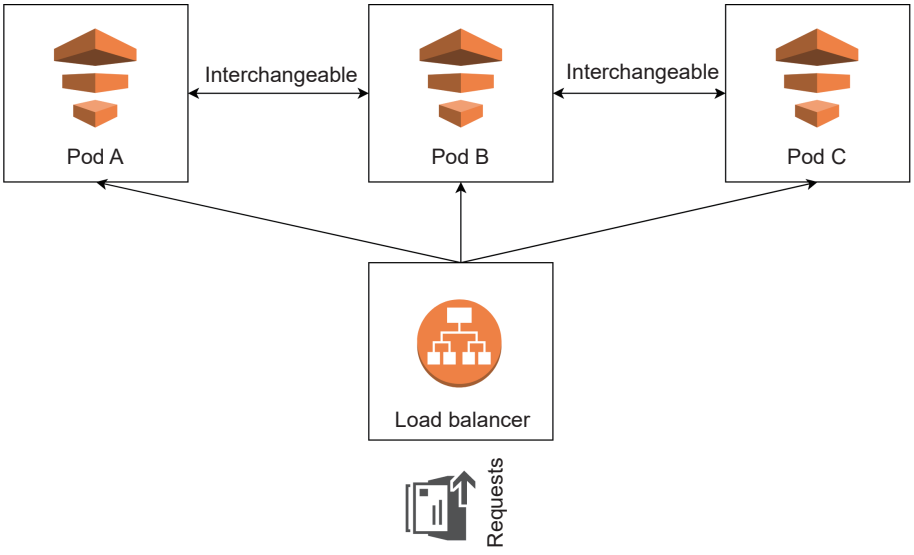

Fig. 1 ReplicaSet: Whenever a request comes in, the Load Balancer imparts the request to one of the pods (Google, 2022).

Definition 8. To deploy a stateful application, the StatefulSet component of Kubernetes is provided. A StatefulSet is an abstraction of one or multiple pods that allows replications in the context of stateful applications. In StatefulSets, all pods are not interchangeable, have their own identity, and are ordinal (ibid.).

In the following, we will describe how Kubernetes StatefulSets can be used, to deploy a reliable stateful application with the example of Redis.

2.2 Redis

Redis is an in-memory database that is intended to guarantee high-performance access. It is used for caching, as a message broker, streaming engine, or simply as a database. Redis's data structures are comprehensive, such as strings, and hashes. Despite its in-memory character, Redis also offers persistence by periodically writing the dataset to disk or continually appending changes to a log file stored on the disk. In addition, Redis offers the possibility to create high availability clusters, whereby the database is replicated over multiple nodes (Redis, 2022). Only one Redis instance can modify the data in the storage to avoid data inconsistency. By other Redis instances, only read-actions are permitted. Furthermore, all instances have access to different physical storage (Google, 2022). To achieve data consistency, the constant synchronization of all Redis instances must be ensured, to keep all storage up-to-date, which Redis does in an automated fashion. As a result, the backup instances are only responsible for updating their storage, while the principal instance initiates the actual data changes. The mechanism for replicating the data functions as follows: As soon as changes are made to the dataset of the principal Redis instance, instructions are sent from the principal instance to the backup instances on how to change the respective data. If this process fails at any point due to network errors, it is detected and the complete dataset from the principal instance to the backup instances will be copied by transferring an

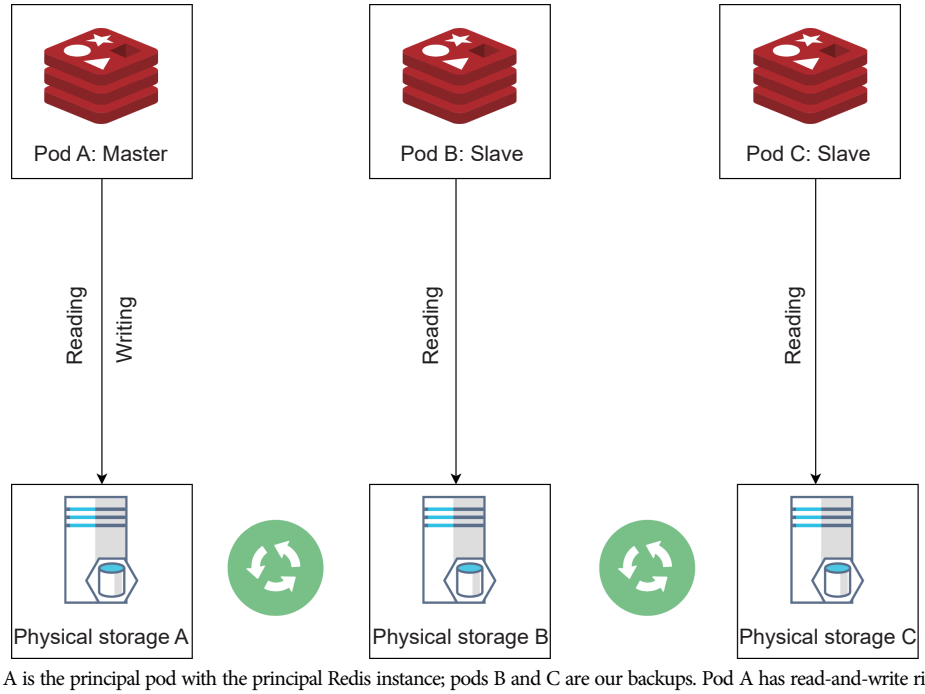

Fig. 2 StatefulSets: Pod A is the principal pod with the principal Redis instance; pods B and C are our backups. Pod A has read-and-write rights to the storage, while pods B and C only have read rights. Every pod has its own physical storage. By continuously synchronizing, every storage has the same content (state) (Google, 2022).

RDB file (Redis, 2022).

To deploy such a Redis cluster in Kubernetes, StatefulSets needs to be used. Fig. 2 illustrates a StatefulSet for setting up a Redis Cluster at Kubernetes.

To successfully deploy a Redis cluster, Sentinel is required. It guarantees high availability in a Redis cluster which is given by constant monitoring of all Redis instances. If Sentinel detects that the current principal pod is no longer reachable, Sentinel selects a new principal pod and downgrades the former one to a backup pod.

3 Related works

The usage of Kubernetes-based FL systems has become popular in recent years. FATE (Liu et al., 2021) is a project that allows institutions and businesses to work together in order to train common models without sharing data. Unlike other frameworks, FATE supports all components for development and production deployment. Regression, artificial neural networks, and tree-based algorithms are examples of ML algorithms that can be used. The findings show that FATE can reduce the overall cost of large-scale training.

KubeFL, a framework proposed by Kim et al. (2021), used Kubernetes to deploy models over multiple clients. The architecture is split into master and worker nodes. The worker nodes are the clients that train the global model deployed in pods. The master receives model updates and aggregates the global model.

Zhuang et al. (2022) developed a platform called EasyFL for simple and low-code usage of FL using Kubernetes or Docker as a standalone solution. EasyFL provides users with different levels of expertise access to FL while ensuring high customization paired with many easy-to-use out-of-the-box features. A plugin architecture allows developers to develop new algorithms and applications to address open problems in FL. Easy FL supports three training methods: standalone processing for simulation on single devices, distributed learning for increasing training speed using multiple devices, and remote training to allow FL live systems.

To overcome the issue of a single point of failure in server-based FL, Pokhrel and Choi (2020a) introduced on-vehicle ML (oVML) to allow single vehicles to fine-tune globally trained models. Given this approach, every vehicle is able to make decisions even if the communication to the rest of the network fails. Nonetheless, learning solely based on locally generated data is usually not enough to train accurate models that are trans-regionally applicable. For that reason, the exchange of information is still required. To do so (ibid.), utilized blockchain technology to store the updates of the global model aggregating the single client model updates. Access to such information is now more reliable because of the general properties of blockchain systems. Instead of retrieving the global model from the server, the vehicle now requests model update information from the blockchain.

Further, Awan et al. (2019) demonstrate the advantages of blockchain-based FL for secure aggregation of model updates. By using the blockchain, the system record information flows while executing FL tasks, such as updating the models. By tracking these flows, in contrast to other FL approaches, the participation of all network entities is more trustworthy.

To our knowledge, there is no approach that uses the concept of orchestration to create redundancy for the central server in an FL network. Since orchestrators such as Kubernetes are widely used in the area of system architectures (Cloud Native Computing Foundation, 2022), including our infrastructure-enhanced autonomous driving architecture deployed in the project BeIntelli, a Kubernetes-based solution is better for the application of FL. Furthermore, it should be mentioned that blockchain-based approaches have limitations, such as high energy consumption and the multitude of ways to maliciously manipulate them (Golosova and Romanovs, 2018). Due to the general characteristics of blockchain technologies, it will cause additional computation time and thus delay data transfers (Pokhrel and Choi, 2020b). In the following, we will propose our approach to overcoming these shortcomings.

4 Method

FL has gained significant traction in recent years as a decentralized machine learning paradigm. Without the need to share raw data, multiple clients can collaboratively train a machine learning model. Clients and servers are considered singular entities within the overall network in FL-based model training. The operation of the network can be significantly affected by the failure of these individual entities. In the beginning, FL-based model training was conceptualized with a relatively simplistic approach to the communication between clients and servers. The server was responsible for orchestrating the model training process, while the clients were considered passive contributors to the learning process. This approach proved to be problematic; however, as both clients and servers were prone to failures that would render the network inoperable. In general, a failure can be defined as any event that interrupts the normal functioning of a client or server and results in its inability to participate in the learning process. Scenarios may include but are not limited to:

- Loss of Internet connectivity: Due to network outages or temporary unavailability, a client or server may be unable to communicate with his clients.
- Sudden shutdown: Hardware or software problems can cause a client or server to shut down unexpectedly and be unable to take part in the learning round.
- Overloaded resources: A client or server may experience resource constraints, such as high CPU or memory utilization, which prevent it from participating effectively in the learning session.

If a client failed, it became impossible for it to participate in learning, and the data it held became inaccessible. In the majority of FL applications, the number of clients is significant, and the failure of one client can be compensated for by the other clients. However, this process requires a reliable workload redistribution mechanism to ensure that the remaining clients are assigned the tasks of the failed client. In contrast, failing servers can be catastrophic to the entire network. It orchestrates the entire training process and ensures that each client receives the appropriate model parameters. In the event of a server failure, the network is inoperable and the training process grinds to a halt. In some cases, the speed and efficiency of the entire training process may be significantly impacted because the network may require manual intervention to recover from the server failure. Companies such as Google recognize the importance of implementing fault tolerance concepts, particularly in distributed systems that rely on centralized servers in their seminal work on the Google File System (GFS) (Ghemawat et al., 2003). Distributing information across different subsystems, such as different machine racks, to keep them available even in case of failure, is a key aspect of their

approach. Furthermore, modern frameworks allow the distribution of information and functionalities on different levels: on different machines of the same computing center, but also on entirely different geographical locations. Therefore, the failure of one node is not mandatory equal to the failure of a whole system, including potential backups. However, such approaches might increase bandwidth usage. Therefore, bandwidth optimization in FL networks is a wide area of research. For more information, see Zhao et al. (2022). Despite the increasing bandwidth usage, the trade-off is robustness and resilience to unexpected server failures. Ghemawat et al. (2003) state that it is worth to make the compromise. For large systems like Google's datacenters, but also for smaller architectures in average enterprises, DevOps engineers' daily experience has consistently shown the importance of deploying software components across multiple machines to ensure availability and resilience. This is a key factor in the success of Kubernetes. Kubernetes has been widely adopted due to its inherent characteristics of using multiple worker nodes rather than relying on a single one Burns et al. (2022). In our experience, it is common for a node to fail for various reasons – for example, a misconfigured log file can consume excessive memory, leaving the server unable to process requests or even issue warning alerts. Our observations confirm Google's position on the importance of designing multi-machine systems for high availability and fault tolerance, and emphasize the value of such strategies for both large and small deployments.

To address the challenge of ensuring high availability and reliability of FL-based model training, we propose a novel architecture in this study (Fig. 3). To overcome the bottleneck and provide high availability for the FL network, our architecture is designed. In particular, we propose a stateful set-based architecture that is composed of three pods, but is not limited to just three pods: pod A, which is the main pod that contains the ML model; and pods B and C, which are backup pods that store their own ML models. Pod A has read and write access to the storage and is responsible for receiving updates directly from

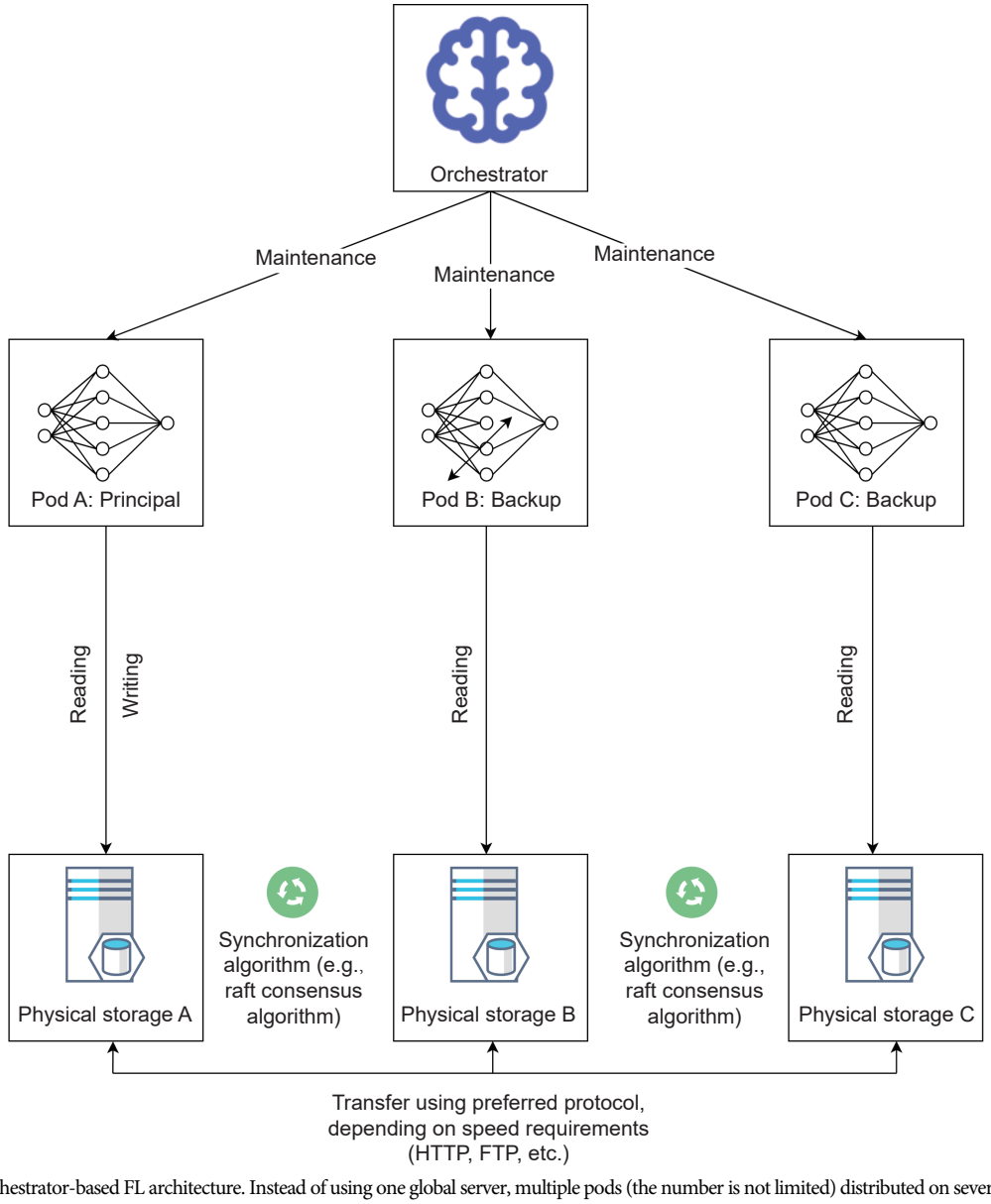

Fig. 3 General Orchestrator-based FL architecture. Instead of using one global server, multiple pods (the number is not limited) distributed on several nodes contain the global model to ensure high availability. Applying a consensus algorithm ensures synchronization. Depending on the synchronization algorithm used, the specific information flow varies. In case of using Redis databases, Pod A is updating Pods B and C by sending update instructions to both. For scaling up the performance of Redis clusters, additional shards can be added, allowing distribution of write requests to multiple shards rather than one. For more information, see Redis (2022).

clients. Conversely, pods B and C have read-only access to storage. Each pod has its own physical storage. The storage devices are constantly synchronized to ensure that they contain the same ML model. As opposed to other FL approaches, our architecture replicates the global model across multiple nodes, making it available even if one node fails. The novel contribution of our architecture is its ability to provide a highly available and reliable FL network, even in large-scale applications. This is achieved by using a stateful set-based architecture. This ensures that the ML model is replicated across multiple nodes. This approach provides a significant improvement in the reliability of the FL network, making it more resilient to failures and reducing the risk of downtime. Additionally, our architecture is highly scalable and easily deployed on various cloud-based platforms. This is due to the inherent scalability of the stateful set-based architecture. This allows us to seamlessly add or remove nodes from the network. This scalability ensures that our architecture is suitable for a wide range of FL-based model training applications, from those on a small scale to those on a large scale in the enterprise.

Using Kubernetes StatefulSets ensures:

- Fast and efficient deployment;
- Reliable network;
- Persistent storage;
- Ease in updating the global model.

As storage systems, we utilize Redis databases. However, it is not mandatory to use Redis, as other database systems are available. We choose Redis because it provides out-of-the-box solutions for implementing data synchronization, whereas other database systems require additional development for this purpose. This can be achieved, for instance, by using the Raft Consensus algorithm (Diego and Ousterhout, 2015), but utilizing Redis eliminates this requirement. The crucial idea is to make the central server redundant, by employing orchestration tools for managing a large number of servers and implementing synchronization algorithms, such as Raft, to update and synchronize all servers. Furthermore, given the substantial community using Kubernetes, our concept enables effortless integration into existing infrastructure networks (Cloud Native Computing Foundation, 2022).

5 Experiment

In the following, we will introduce the experiments conducted for analysing our proposed architecture. As a baseline method for comparison, we use the classic FL architecture, consisting of a single server and the clients accordingly. To build our Kubernetes cluster, we use Kind. Kind is a container-based Kubernetes cluster creation tool that enables the development of multi-nodal clusters without the need to prepare multiple machines. The most notable feature of Kind is its simple cluster configuration, which can be specified using a YAML configuration file that determines the number of nodes and their role within the cluster. For more information on Kind, see Kubernetes (2022). Clusters created with Kind mimic multi-machine clusters. Therefore, this simulation can be directly applied to real-world scenarios. See Table A1 inAppendix for the hardware specifications of the machine running the experiment.

Furthermore, we used several YAML files to setup the following cluster components:

- 3 Redis pods with 3 different Redis instances: redis-0, redis-1, and redis-2;
- 3 Sentinel pods with 3 different Sentinel instances: sentinel-0, sentinel-1, and sentinel-2;
- 3 interfaces, that allow communication with the Redis instances: interface-0, interface-1, and interface-2. To do that, we set up REST-APIs for each Redis instance. By calling endpoints of the REST-API, we enable communication to the Redis instances.

With the setup of the communication interfaces, we implemented the upload process of our model into the Redis database. For interface-0 only, we place the model in the redis-0 instance. By default, the replication functionality of Redis will automatically transfer the model to redis-1 and redis-2. As a model, we used a fully connected multi-layer perceptron consisting of a few layers and weights.

To ensure a proper network setup, we need to set some options. Redis defaults to listening on all interfaces on the server. However, by binding specific IP addresses, it is possible to specify specific interfaces. Binding to all interfaces can be a security risk because it exposes the instance to anyone on the Internet if the Redis server is directly accessible from the Internet. To avoid this, we can configure Redis to listen only on the IPv4 loopback interface address, allowing it to accept connections solely from clients on the same server. In high throughput environments, it is important to have a high backlog to avoid having clients connect slowly. The Linux kernel can silently reduce the backlog to a default value, so we need to increase the value to ensure connection reliability. However, if the connection between the principal and backup pods is lost, the backup pod may still respond to client requests with potentially outdated data or no data at all if it is still in the process of replicating the dataset. It is necessary to properly handle these connection losses to ensure that the network remains available.

To properly set up the saving behavior, we need to configure two parameters: One that determines the number of write operations required before saving, and one that determines the number of seconds that must elapse before saving is initiated. The current snapshot of the database will be saved only if these two conditions are met. We will set up the following rules to determine when to save: After 900 s, if at least one key has been changed; after 300 s, if at least 10 keys have been changed; and after 60 s, if at least 10,000 keys have been changed.

We will perform a series of experiments to thoroughly evaluate the effectiveness of our proposed solution in achieving high availability with our chosen architecture.

Experiment 1 To get a preliminary understanding of our setup, we will start with a simple experiment. To do this, we initiate our Redis and Sentinel instances and interfaces. This will allow us to store the simple model described earlier in the Redis databases. We will then test how the system responds to changing weights by incrementing each weight by 1. In a practical scenario, this would be done using the Federated Averaging algorithm or a similar approach to build a comprehensive model from client updates. Next, we will do an intentional removal of the redis-0 instance. By default, the first primary pod always has the lowest numbering. When removed, Sentinel will promptly select a new Redis instance to serve as the primary pod. A new redis-0 pod is then scheduled to be backed up. If the model is modified by the newly selected Redis instance, each weight is again incremented by 1, we expect the system to start with the updated model instead of the original if our architecture works. We can thoroughly verify our concept's functionality by repeating the weight incrementing, primary pod deletion, and weight-checking process. The algorithm of our experiment is described in Algorithm 1. To increase the

complexity of the experiment, we repeat it with 2 clients and instead of just increment the model weights, we apply a serious learning on both clients, for approximating the solution of a linear regression. The algorithm of the linear regression version is described in Algorithm 2.

Algorithm 1 Procedure for Testing Model Architecture: Basic

1: Initialize model stored in principal pod
2: **repeat**
3: Increment each weight of the model by 1
4: Delete principal pod
5: Select new Redis instance as principal pod
6: Increment each weight of the model by 1
7: Verify that the model starts with modified weights, not initial weights
8: **until** User stops.

Algorithm 2 Procedure for Testing Model Architecture: Linear Regression

1: Initialize model stored in principal pod
2: **repeat**
3: Do a learning round on both clients
4: Both clients send their gradients to the server, the server aggregates them
5: Delete principal pod
6: Select new Redis instance as principal pod
7: Clients retrieve the current model form the current principal pod
8: Verify that both clients have the correct, updated model
until User stops.

Experiment 2 In addition, we use the memtier benchmark (Memtier-benchmark, 2022) to evaluate the performance of our architectural design. The tool mimics clients and measures the latency and throughput of requests sent to the system. We evaluate the architecture's performance under different workloads by simulating multiple client threads and connections. By default, memtier benchmark mixes GET and SET requests. The tool provides comprehensive information about the system's performance, such as the number of operations per second, cache hit percentage, and request latencies. These statistics can be used to identify any performance-impacting bottlenecks or problems within the system. To perform the performance test, several variables must be determined, including:

- Host IP of our current Redis principal pod;
- Port of our current Redis principal pod;
- Clients;
- Threads;
- Test-time.

The level of parallelism and concurrency of the benchmark is controlled by the clients and threads options in the memtier benchmark utility. The test-time parameter specifies the duration of the benchmark in seconds. The number of client connections to be used is specified by the client's parameter. Each client connection mimics a different client contacting the server. The overall load on the server increases as the number of client connections increases. The number of worker threads to use for each client connection is specified by the threads parameter. Each worker thread has the ability to make requests to the server independently of the other worker threads. Adding more worker threads increases the parallelism of the benchmark. However, it may also increase the competition for resources between client connections. The benchmark will continue to run until the specified time has elapsed or the specified number of requests has been served, whichever comes first. In general, adding client connections and worker threads puts more load on the server and makes it easier to detect performance bottlenecks at higher concurrency levels. However, finding the ideal balance between concurrency and parallelism is critical because overloading the server can result in poor performance. The size and capabilities of the system, as well as the expected workload of the architecture, will influence the values of the clients, threads, and chosen test-time parameters. In our experiment, we used the following settings in Table 1.

Table 1 Setup Memtier parameters. Thread represents the number of worker threads used to generate requests in the benchmark; client is the number of connections that each worker thread maintains to the server; test-time is the duration of the benchmark

Benchmark	Client	Thread	Test-time
1	50	1	30
2	100	2	30
3	400	8	60

Experiment 3 Finally, we measure the time it takes to switch between the current principal pod and a new one in case of failure to test the performance of our architecture. In comparison, we will measure the time between pod failure and re-creation of a new pod, for the single server baseline method. This is critical to ensure that the FL network remains up and running in a timely manner.

6 Results

The effectiveness of our architecture was validated by the results of Experiment 1. Our model remained current, even in the case of failure, for both the base implementation and the linear regression. Figs. 4a–4c show the latency distribution of our benchmark for the three benchmarks, benchmarks 1–3, respectively applied on our cluster setup. Accordingly, Figs. 5a–5c show the latency distribution of our benchmark for the three benchmarks, benchmarks 1–3, respectively applied on our single server setup. The value pair of 9.54 ms and 100% GET/SET requests – for instance – indicates that 100% of the GET/SET requests had a latency up to 9.54 ms. Each value pair in the curve corresponds to a specific range of latencies, and the *y*-axis indicates the percentage of requests that had a latency within that range. The values are cumulative, so the final value shows the percentage of requests that had a requests that had a latency at the upper bound of the latency/*x*-axis value, but it also includes all requests that had latencies in the lower ranges (0.295–0.791 ms, 0.791–0.879 ms, etc.). If we look at the ops/sec metric, we can see that the Redis cluster configuration consistently achieves a higher total number of operations per second when compared to the single Redis instance, see Table 2. In particular, in benchmarks 1, 2, and 3, the cluster achieves 85,665.40, 65,566.04 and 38,455.37 ops/sec (totals), respectively. In contrast, the single instance configuration achieves only 67,409.53, 61,804.27, and 36,749.25 ops/sec, respectively. This indicates that Redis Cluster provides superior throughput, due to its ability to distribute load across multiple nodes, improving overall performance. Regarding latency, the average and percentile latencies vary depending on the benchmark scenario. However, our result has shown that Redis Cluster has superior performance in terms of latency compared to single instances (Table 2). The results show that the cluster setup provides a significant reduction

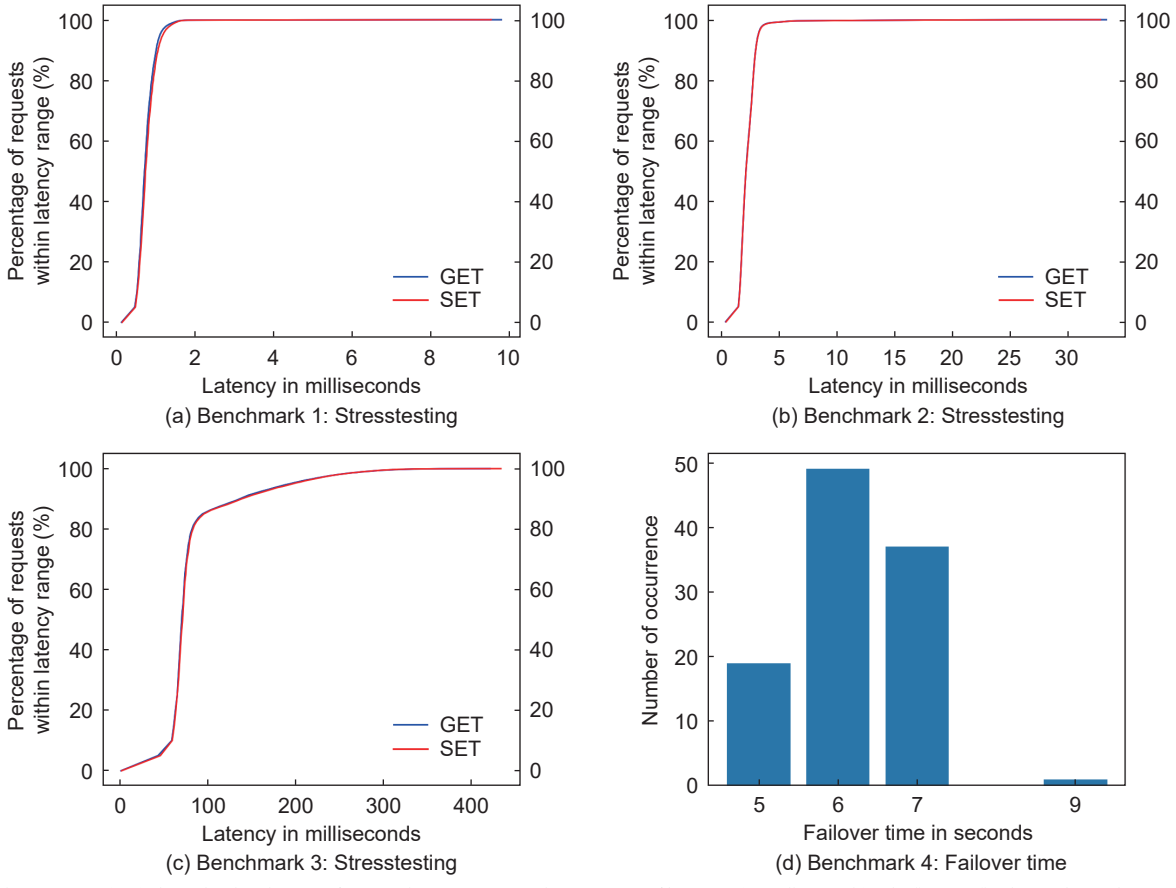

Fig. 4 Cluster Setup. (a–c) show the distribution of request latencies, grouped into ranges of latencies (in milliseconds). The "Percent" column shows the percentage of requests that had a latency within the corresponding range. (d) shows the failover time between the failure of the current principal pod and the election of a new principal pod.

in the execution time of operations compared to single instances. Looking at the misses/sec for GET operations, there is no significant difference between the Redis cluster and single instance setups. This is an indication that both configurations are equally effective in terms of cache hit ratio. The higher throughput of Redis is even more attractive for heavy workloads. Redis Cluster consistently transfers more data per second in all benchmarks when looking at the KB/sec metric in Table 2. This further underscores the ability of the cluster to handle higher workloads and provide better performance. With a high rate of operations per second and reasonably low latencies, the findings demonstrate good performance for both SETs and GETs on our cluster setup. The fact that there are far more GETs than SETs suggests that our Redis instance is mostly used for read operations. As we can see in Figs. 4a–4c, the behavior of GET and SET requests is approximately equal, therefore the workload was focused on GET requests rather than SET requests, to retrieve better results in terms of Misses/sec. It is important to note that both SETs and GETs have 99th percentile latencies significantly higher than the mean latencies. This shows that a small subset of queries likely have latencies that are longer than those of most requests. The number of operations per second has dropped for benchmarks 2 and 3, respectively for both setups. This is due to the increased number of threads and connections per thread in this test. Having more threads and connections per thread places a greater load on the Redis instance/cluster, causing it to become more stressed and potentially impacting its performance.

Overall the results of the three benchmarks 1–3 for the cluster setup, showed remarkable performance with high operations per second and relatively low latencies, despite the weak hardware used for the tests. Across three different test scenarios, the benchmark results provide important information about the performance and latency behavior of the system. These findings have critical implications for designing, configuring, and optimizing systems in a variety of application contexts. The observations suggest that the system may experience increased latency under heavier workloads or non-optimal configurations. This may affect the overall user experience and response time. Monitoring latency and adjusting system settings to balance performance and responsiveness based on the specific use case is therefore critical. In addition, the number of cache misses remains relatively low in all scenarios. This indicates efficient cache management. Nevertheless, some cache misses still occur. This underscores the need for continuous monitoring and fine-tuning of caching strategies to minimize their impact on performance, as well as effective cache management in the maintenance of high performance systems..

For Experiment 3, the failover process for the cluster setup was measured approximately 100 times in order to determine the average time it takes to elect a new principal pod after the previous one fails, see Fig. 4d. In 19 of the experiments, the process took only 5 s to complete. In 49 of the experiments, the process took 6 s, and in 37 of the experiments, it took 7 s. There was also one experiment in which the process took 9 s. These results suggest that the failover process is generally efficient, with a majority of the experiments taking 5–7 s to complete. However, the failover time can be drastically reduced, by using proper hardware rather than our simulation environment with weak hardware, see Appendix. To compare, we tested the single-server setup by measuring the

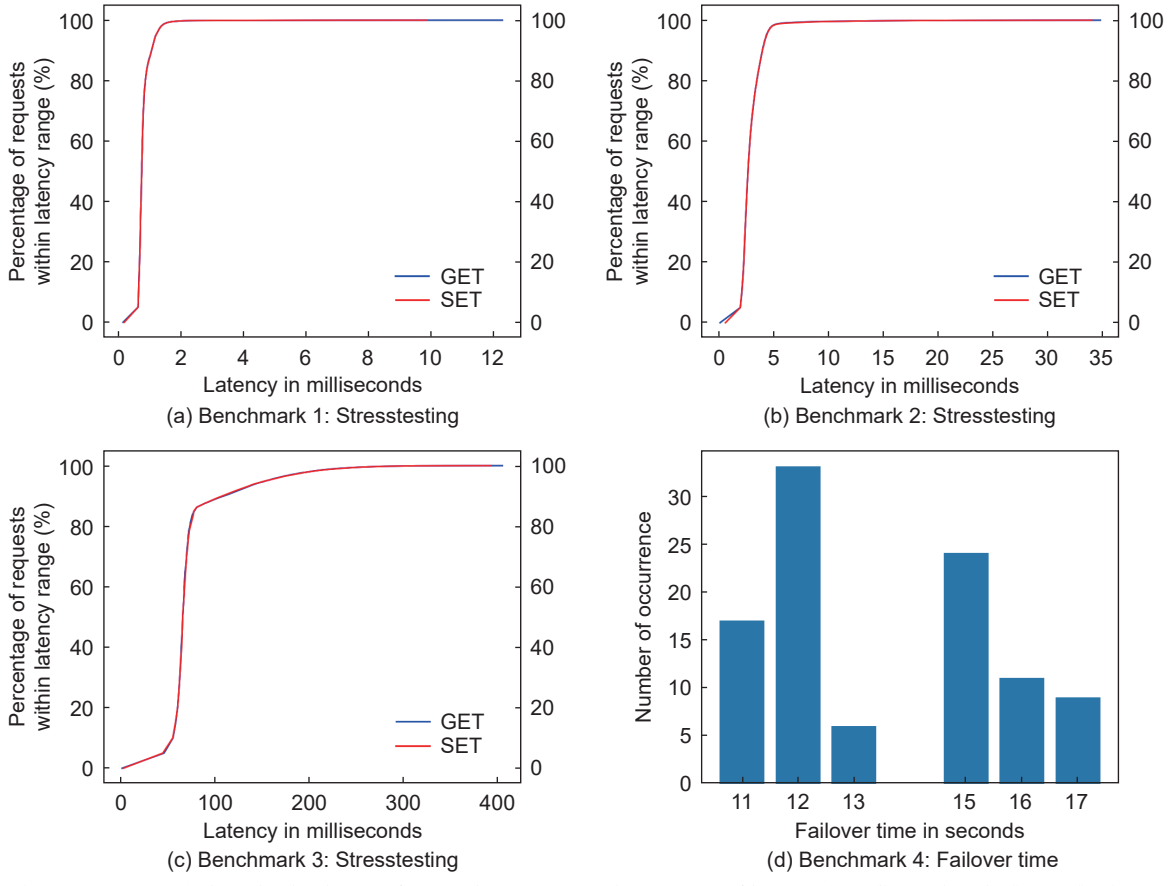

Fig. 5 Single-server Setup. (a–c) show the distribution of request latencies, grouped into ranges of latencies (in milliseconds). The "Percent" column shows the percentage of requests that had a latency within the corresponding range. (d) shows the time between the failure of the current pod and the creation of a new pod.

Table 2 Stresstesting Redis. 'Ben.' indicates the benchmark. The setup of those benchmarks can be found in Table 1. 'Ops/sec' is the number of operations (e.g., SETs or GETs) that the benchmark completed per second. 'Misses/sec' is the number of cache misses (i.e., unsuccessful GETs) that the benchmark completed per second. 'Avg. Latency' is the average time taken for an operation (in milliseconds). 'p50 latency', 'p99 latency', and 'p99.9 latency' are the 50th, 99th, and 99.9th percentile latency (in milliseconds), respectively. This means that 50%, 99%, or 99.9% of the operations took this time or less. "KB/sec" is the number of kilobytes transferred per second. All values are rounded to the second decimal point. The first set of rows contains the result for the benchmark that was run on the redis cluster, and the second set contains the result for the benchmark that was run on the single redis instance.

Ben.	Type	Ops/sec	Misses/sec	Latency				KB/sec
				Avg.	p50	p99	p99.9	
1	Sets	7,790.81	—	0.79	0.75	1.50	2.91	459.26
1	Gets	77,874.59	4.00	0.76	0.73	1.44	2.94	2,322.00
1	Totals	85,665.40	4.00	0.76	0.73	1.45	2.94	2,781.26
2	Sets	5,961.38	—	2.34	2.19	4.42	18.30	600.08
2	Gets	59,604.66	8.59	2.33	2.19	4.32	17.79	3,034.07
2	Totals	65,566.04	8.59	2.33	2.19	4.35	18.01	3,634.15
3	Sets	3,852.33	—	75.87	67.58	231.42	303.10	296.36
3	Gets	34,603.04	4.78	75.95	67.58	228.35	301.06	1,491.05
3	Totals	38,455.37	4.78	75.94	67.58	229.38	301.06	1,787.42
1	Sets	6,131.05	—	0.81	0.76	1.53	3.25	432.90
1	Gets	61,278.48	5.60	0.81	0.76	1.52	3.28	2,188.75
1	Totals	67,409.53	5.60	0.81	0.76	1.52	3.25	2,621.65
2	Sets	5,619.34	—	2.97	2.72	6.08	18.43	472.19
2	Gets	56,184.93	5.16	2.97	2.72	5.79	18.30	2,387.49
2	Totals	61,804.27	5.16	2.97	2.72	5.79	18.43	2,859.68
3	Sets	3,486.47	—	84.65	72.19	278.53	335.87	268.20
3	Gets	33,262.78	4.04	83.74	71.17	278.53	342.02	1,348.48
3	Totals	36,749.25	4.04	83.83	71.17	278.53	339.97	1,616.68

time it takes to restart a new Redis instance after a failure instead of switching to one of the backups as in the clustered setup (Fig. 5). The results prove that it takes significantly longer to schedule a new instance. We measured a time between 11 and 17 s, with the majority between 12 and 15 s. However, it is important to note that in real-world applications, when a central server fails, for usual, it takes much longer than our measured time because it typically requires manual intervention to fix the failure and restore the server.

7 Conclusions

In recent years, especially in applications where privacy is a concern, the use of FL as a decentralized approach to training machine learning models has gained significant traction. However, to fully realize the potential of FL, as with any new technology, there are still open challenges that need to be addressed. In this study, we have focused on the issue of availability in server-based FL networks. Availability can be compromised by a single point of failure.

In order to address this issue, we have proposed a modern architecture model that adds redundancy to the central server and applies a consensus algorithm to keep the replications up to date. We have shown that the proposed architecture works as expected. Moreover, even for our weak hardcore components selected for the simulation, the network downtime decreases and the response rate of our architecture is high, see Appendix. Our study leverages the work of Chen et al. (2016), which demonstrates the potential of Redis clusters to deliver superior results, especially on suitable hardware. Building on the results of their work, we emphasize the importance of optimizing hardware resources in order to maximize the performance and scalability of Redis clusters for data processing tasks. In the context of infrastructure-enhanced autonomous driving, where our proposed solution enables the deployment of high-availability FL networks, our developed architecture is particularly relevant.

Some limitations should be mentioned despite the success of this architecture:

- This approach works well especially if enough Redis instances are deployed. If the number is too low, it can happen that all instances fail, which means that the entire network is no longer available.
- Individual changes to the record may be lost. This is because the changes are only abandoned at the principal pod. Before updating the backup pods, the principal pod replies to the user that it has received the data. If the principal pod fails before sending the updates to the backup pods, the network loses the information about the last change while the user assumes that everything was transmitted correctly (Redis, 2022).

However, it should be noted that if enough Redis instances are deployed, the losses are reduced to small amounts of data. Due to the generally higher availability provided, the network is still active and robust against data loss.

It should also be noted that although we have conducted performance tests, our architecture in principle has a flexible performance depending on the configuration:

- Which protocol is used between the respective pods for data transfer (FTP, HTTP, etc.);
- What hardware resources the respective servers have;
- Which synchronisation algorithm is used.

The efficiency and robustness of the architecture can be further improved by increasing hardware resources such as processing power, memory, and storage capacity. The result is faster model convergence, lower communication latency and more efficient handling of larger numbers of clients. In addition, better hardware can lead to improved generalisation and overall performance of the federated learning system by enabling the exploration of more complex and computationally intensive models. Therefore, in the context of large-scale and high-performance computing scenarios, investment in appropriate hardware can play a significant role in unlocking the full potential of our approach. For performance tests of Kubernetes and Docker, see Ferreira and Sinnott (2019) and Xie et al. (2017). Consequently, our results represent a critical step forward in the wider adoption of FL, as they address a key challenge that has limited the applicability of this technology to date. By enabling model training at the edge, in the cloud, and in vehicles, our solution enables the development of a wide range of applications, including object detection and trajectory prediction models, while maintaining the privacy of data owners.

Synchronising the global model across multiple nodes is costly, thus in an environment without any server failures, our architecture will be slower. Though, it should be examined at which server failure rate our architecture becomes more efficient, in order to provide a rule of thumb for researchers and practitioners consideration between FL as introduced as the basis and our architecture.

With FL, other practical issues and obstacles for the distributed training of models can be addressed and thus trained more efficient. Especially, for use cases in which transferring the training data is more costly than transferring the (aggregated) model updates, this is in particularly the case for devices that have slow internet connection, e.g., vehicles connected via cellular or V2X connection in a connected and autonomous driving use case.

This study is based on a two tier FL framework, though this concept can also be extended to a multi layer FL framework, for example dispersed FL, see Khan et al. (2022), in which clients are for example regionally linked to a (sub-) server which aggregated the model updates that occur in the respective region and these to a global server. This can address especially for large scale use cases, e.g., with trans-regional or global reaching the FL performance. Our concept can be adapted for these FL approaches, in order to achieve more reliable middle level server performance.

Appendix

For setting up the cluster, we used a notebook with the following resources in Table A1.

Table A1 Ressources used in the experiment

OS	Ubuntu 20.04
Kubernetes version	v1.23.0
Docker version	20.10.17
ContainerD version	1.6.7
Disk	300 GB
RAM	8 GB
CPU	Intel i7-8550U CPU @ 1.80 GHz

Replication and data sharing

The code for the experiment is available on GitHub: https://github.com/BenjaminAcar/FLAvailability.

Acknowledgements

This study is based on the research conducted in the BeIntelli

project, funded by the German Federal Ministry for Digital and Transport (BMDV) (No. 01MM20004).

Declaration of competing interest

The authors have no competing interests to declare that are relevant to the content of this article.

References

Alpaydin, E., 2021. *Machine Learning*. Cambridge, MA: The MIT Press, 2021.

Augusto, M. G., Hessler, A., Keiser, J., Masuch, N., Albayrak, S., 2021. Towards intelligent infrastructures and AI-driven platform ecosystems for connected and automated mobility solutions. https://itsworldcongress.com/the-book-of-abstracts

Awan, S., Li, F., Luo, B., Liu, M., 2019. Poster: A reliable and accountable privacy-preserving federated learning framework using the blockchain. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, 2561–2563.

Bernstein, D., 2014. Containers and cloud: From LXC to docker to kubernetes. IEEE Cloud Comput, 1, 81–84.

Burns, B., Beda, J., Hightower, K., 2022. *Kubernetes: Up and running*. 3rd edn. Sebastopol, USA: O'Reilly Media, Inc., 1–202.

Chen, S., Tang, X., Wang, H., Zhao, H., Guo, M., 2016. Towards scalable and reliable in-memory storage system: A case study with redis. In: 2016 IEEE Trustcom/BigDataSE/ISPA, 1660–1667.

Docker, 2022. https://www.docker.com

Ferreira, A. P., Sinnott, R., 2019. A performance evaluation of containers running on managed kubernetes services. In: 2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), 199–208.

Cloud Native Computing Foundation, 2022. CNCF Annual Survey 2021. https://www.cncf.io/reports/cncf-annual-survey-2021

Ghemawat, S., Gobioff, H., Leung, S. T., 2003. The Google file system. In: Proceedings of the nineteenth ACM symposium on Operating systems principles, 29–43.

Golosova, J., Romanovs, A., 2018. The advantages and disadvantages of the blockchain technology. In: 2018 IEEE 6th Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE), 1–6.

Google, 2022. Kubernetes. https://kubernetes.io/de

Hausenblas, M., 2022. Kubernetes: State and Storage. https://cloud.redhat.com/blog/kubernetes-state-storage

Kairouz, P., Brendan McMahan, H., Avent, B., Bellet, A., Bennis, M., Bhagoji, A. N., *et al*., 2021. Advances and open problems in federated learning. Foundations and Trends in Machine Learning, 14, 1–210.

Khan, L. U., Tun, Y. K., Alsenwi, M., Imran, M., Han, Z., Hong, C. S., 2022. A dispersed federated learning framework for 6G-enabled autonomous driving cars. IEEE Trans Netw Sci Eng, 1–12.

Kim, J., Kim, D., Lee, J., 2021. Design and implementation of kubernetes enabled federated learning platform. In: 2021 International Conference on Information and Communication Technology Convergence (ICTC), 410–412.

Kubernetes, 2022. https://kind.sigs.k8s.io

Li, Y., Tao, X., Zhang, X., Liu, J., Xu, J., 2022. Privacy-preserved federated learning for autonomous driving. IEEE Trans Intell Transport Syst, 23, 8423–8434.

Liu, Y., Fan, T., Chen, T., Xu, Q., Yang, Q., 2021. FATE: An Industrial Grade Platform for Collaborative Learning With Data Protection. J Mach Learn Res, 22, 1–6.

McMahan, H. B., Moore, E., Ramage, D., Hampson, S., Arcas, B. A. Y., 2016. Communication-efficient learning of deep networks from decentralized data. https://arxiv.org/abs/1602.05629.pdf

Memtier-benchmark, 2022. https://github.com/RedisLabs/memtier_benchmark

Nakanoya, M., Im, J., Qiu, H., Katti, S., Pavone, M., Chinchali, S., 2021. Personalized federated learning of driver prediction models for autonomous driving. https://arxiv.org/abs/2112.00956.pdf

Nguyen, A., Do, T., Tran, M., Nguyen, B. X., Duong, C., Phan, T., *et al*., 2022. Deep federated learning for autonomous driving. In: 2022 IEEE Intelligent Vehicles Symposium (IV), 1824–1830.

Diego, O., Ousterhout, J., 2015. The raft consensus algorithm. Lecture Notes CS 190.

Pokhrel, S. R., Choi, J., 2020a. A decentralized federated learning approach for connected autonomous vehicles. In: 2020 IEEE Wireless Communications and Networking Conference Workshops (WCNCW), 1–6.

Pokhrel, S. R., Choi, J., 2020b. Federated learning with blockchain for autonomous vehicles: Analysis and design challenges. IEEE Trans Commun, 68, 4734–4746.

Redis, 2022. https://redis.io

Xie, X. L., Wang, P., Wang, Q., 2017. The performance analysis of Docker and rkt based on Kubernetes. In: 2017 13th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD), 2137–2141.

Yang, Q., Liu, Y., Cheng, Y., Kang, Y., Chen, T., Yu, H., 2019. Federated learning. Synth Lect Artif Intell Mach Learn, 13, 1–207.

Zhang, H., Bosch, J., Olsson, H. H., 2021. End-to-end federated learning for autonomous driving vehicles. In: 2021 International Joint Conference on Neural Networks (IJCNN), 1–8.

Zhao, Z., Xia, J., Fan, L., Lei, X., Karagiannidis, G. K., Nallanathan, A., 2022. System optimization of federated learning networks with a constrained latency. IEEE Trans Veh Technol, 71, 1095–1100.

Zhuang, W., Gan, X., Wen, Y., Zhang, S., 2022. EasyFL: A low-code federated learning platform for dummies. IEEE Internet Things J, 9, 13740–13754.

Benjamin Acar holds his M.S. degree in Technomathematics (Mathematics with a minor in Physics) from the Karlsruhe Institute of Technology (KIT), Germany. Previously, he worked as a Risk Analyst in one of the largest German banks. Currently, he is pursuing his Ph.D. degree in Computer Science, focusing on Multi-Agent Systems. His research interests encompass Distributed Systems, Machine Learning, and Software Engineering.

Marius Sterling joined a M.S. program in Berlin, a cooperation between Humboldt-, Technical and Free University of Berlin, and in Statistics for Smart Data from 'Ecole Nationale de la Statistique et de l' Analyse de l'Information (ENSAI), Rennes, France. Currently, he leads the sub-project responsible for platform development and creating selected showcase services within the BeIntelli research project. His research interests encompass Data Science, applied Machine and Deep Learning, and CCAM, with a particular interest in CCAM based Smart and Efficient Parking Systems.