Advancing Serverless ML Training Architectures via Comparative Approach

Amine Barrak, Ranim Trabelssi, Fabio Petrillo and Fehmi Jaafar

***Abstract*—The field of distributed machine learning (ML) faces increasing demands for scalable and cost-effective training solutions, particularly in the context of large, complex models. Serverless computing has emerged as a promising paradigm to address these challenges by offering dynamic scalability and resource-efficient execution. Building upon our previous work, which introduced the Serverless Peer Integrated for Robust Training (SPIRT) architecture, this paper presents a comparative analysis of several serverless distributed ML architectures. We examine SPIRT alongside established architectures like ScatterReduce, AllReduce, and MLLess, focusing on key metrics such as training time efficiency, cost-effectiveness, communication overhead, and fault tolerance capabilities. Our findings reveal that SPIRT provides significant improvements in reducing training times and communication overhead through strategies such as parallel batch processing and in-database operations facilitated by RedisAI. However, traditional architectures exhibit scalability challenges and varying degrees of vulnerability to faults and adversarial attacks. The cost analysis underscores the long-term economic benefits of SPIRT despite its higher initial setup costs.**

This study not only highlights the strengths and limitations of current serverless ML architectures but also sets the stage for future research aimed at developing new models that combine the most effective features of existing systems.

***Index Terms*—Distributed Machine Learning, Peer-to-Peer (P2P), Serverless Computing, Fault Tolerance, Robust Aggregation.**

I. Introduction

The field of machine learning (ML) has experienced a significant transformation, primarily due to the increasing complexities and growing amounts of data associated with modern ML models. Conventional single-machine learning frameworks, which were previously dominant, now struggle to meet these growing demands. In this scenario, distributed machine learning is seen as a highly efficient approach that utilizes a network of computational nodes to divide and process the workload in parallel[1].

Multiple distributed ML architectures have been introduced over the years, many of which are fundamentally based on the structures of the Parameter Server (PS) and Peer-to-Peer (P2P) topolgies [2]. Each of these represents a unique methodology for orchestrating the management and distribution of tasks and data across nodes in a distributed system, with their own set of benefits and challenges [3], [4]. In the parameter server, for instance, the worker nodes perform computations on their respective data partitions and communicate with the parameter server (PS) to update the global model [5]. In contrast, peer-to-peer (P2P) distributes the model parameters and computation across all nodes in the network, eliminating the need for a central coordinator [6].

However, independent of the topology, traditional computing models within these architectures often struggle with inflexible resource allocation, leading to underutilized infrastructure or resource shortages during peak demands (i.e, training). This can escalate operational costs and hinder scalability [7]. Moreover, the intricacies of managing distributed systems for deep learning, necessitate substantial expertise and effort. This can cause a shift in focus away from the core aspects of machine learning development [8].

In response to these challenges, serverless computing has emerged as a revolutionary solution within the domain of distributed ML architectures [9], [10], [11]. Platforms such as Amazon Lambda[12], Google Cloud Functions[13], and Azure Functions[14] provide dynamic scalability and a cost-effective model. This paradigm empowers ML practitioners to prioritize model development over infrastructure management complexities. Subsequent research has introduced frameworks that are proven to enhance cost reduction [15], scalability [16], [17], and training time [18].

Even with the significant strides made by integrating serverless computing, the field continues to grapple with two primary challenges: **(1) Database Dependency and Communication Latency**: A significant aspect of serverless architectures is their reliance on databases for communication. This necessity arises from the stateless nature of serverless architectures and limitations in directly transmitting large data sizes[19], [20]. As a result, communication latency can arise, especially during the iterative process of model update and gradient aggregation[20], [21]. **(2) Fault Tolerance and Security Concerns**: Ensuring robust fault tolerance in distributed environments is another significant concern in distributed Machine Learning (ML) environments [22]. One issue is the absence of mechanisms to authenticate new nodes joining the network [23], [24]. Unauthenticated or malicious nodes could join the network, potentially causing significant disruption to the training process. Beyond the threat of unauthenticated new nodes, even existing, trusted nodes can become compromised, introducing malicious gradients intended to sway the model training in harmful directions [25], [23], [26]. This risk emphasizes the importance of robust aggregation methods [27], [28], [29], which can help mitigate the effect of such outliers on the overall model's performance.

In our prior work [30], we addressed these challenges by presenting the Serverless Peer Integrated for Robust Training

Manuscript created October, 2023; This work was developed by the IEEE Publication Technology Department. This work is distributed under the LaTeX Project Public License (LPPL) (http://www.latex-project.org/) version 1.3. A copy of the LPPL, version 1.3, is included in the base LaTeX documentation of all distributions of LaTeX released 2003/12/01 or later. The opinions expressed here are entirely that of the author. No warranty is expressed or implied. User assumes all risk.

(SPIRT) architecture. This proposed solution, a robust peer-to-peer (P2P) serverless ML training architecture, is characterized by several key contributions: (i) SPIRT introduces a fully automated ML training workflow within a P2P architecture using orchestrated workflow coordination, e.g., AWS Step Functions, maximizing scalability while reducing operational complexities. (ii) Recognizing the critical role of databases in serverless ML training systems, SPIRT incorporates a modified RedisAI to facilitate in-database model updates. This enhancement streamlines communication and reduces overhead, facilitating more efficient serverless ML operations. (iii) SPIRT architecture ensures fault tolerance by integrating secure, new participant integration, and robust aggregation mechanisms to establish a fault-tolerant environment.

SPIRT's groundbreaking role in establishing the first fully serverless peer-to-peer (P2P) architecture marks a significant milestone, yet it represents just one facet of a rapidly evolving domain. As various architectures emerge, they all share the common goal of distributed ML training, with each framework distinguishing itself by proposing unique communication patterns. For instance, the LambdaML Architecture [20] introduced distributed machine learning (ML) training on Function-as-a-Service (FaaS) platforms, proposing two communication patterns: ScatterReduce and AllReduce. In ScatterReduce, each gradient is divided among the workers, each responsible for aggregating the assigned portion. In AllReduce, one worker aggregates the total, and then the consolidated results are distributed to all workers. They implemented checkpoints to ensure continuity, addressing the short lifespan of serverless operations. Compared to traditional IaaS setups, LambdaML stands out for its accelerated training speeds, though it acknowledges that cost advantages are not always guaranteed. Conversely, the MLless Architecture [15] introduced two key features for distributed training on Function-as-a-Service platforms: first, a significance filter that enables the sharing of only significant model updates among workers; and second, a scale-in scheduler that dynamically adjusts the number of serverless functions based on workload. Each of these proposed frameworks has significantly contributed to advancing distributed machine learning in serverless environments. To fully understand each framework's unique contributions, it is crucial to compare them through extensive experimentation at various stages of ML training, such as synchronization, communication, and aggregation. This comparison will illuminate how each framework is tailored to enhance certain aspects of the training process in a distributed, serverless context, thereby advancing the overall efficacy of machine learning training in such environments.

In this work, we extend our research beyond the initial implementation of the Serverless Peer Integrated for Robust Training (SPIRT) architecture to include a wider range of serverless distributed machine learning (ML) architectures. We aim to conduct an extensive comparative study of these architectures to understand how serverless frameworks can optimize and enhance distributed ML training. We provide a replication package for this study [1].

[1] https://sites.google.com/view/spirt-paper/

This study provides valuable insights for practitioners in selecting the most suitable serverless distributed ML architecture for specific scenarios, based on empirical evidence. Concurrently, the study lays a foundational pathway for researchers, encouraging the creation of a hybrid framework that combines the most effective features from diverse architectures, fostering advancements in the field of machine learning.

II. Related work

This section reviews literature on distributed machine learning and serverless computing, focusing on decentralized ML on P2P paradigm and fault tolerance mechanisms .

A. Distributed Machine Learning Architectures

Distributed machine learning has gained prominence due to the intense computational requirements and extensive data in training sophisticated ML models [31], [2], [32]. Within this context, serverless computing, with its modular, function-based (FaaS) computational units, has emerged as a key technology, particularly in distributed training scenarios. This is exemplified by the work of Jiang et al. [20], who conducted a comparative study exploring the trade-offs between FaaS-based and IaaS-based systems in training distributed ML models. Their findings highlight the potential of serverless computing to accelerate certain aspects of ML training. Building on all this, a significant and growing body of research has focused on its application in training distributed ML models [33], [19], [34], [17], [35], [18]. Within this research, various architectural have been proposed, each employing serverless computing in distinctive ways to distribute computational tasks. In some designs [15], a hierarchical structure is employed, where certain serverless workers function as supervisorS. These workers are primarily responsible for overseeing the process, making critical decisions, and coordinating the efforts of other workers. Alternatively, other architectures [30] adopt a more balanced approach, where all serverless workers share equal responsibility. In these setups, tasks and decision-making processes are distributed evenly. Moreover, there are also architectures [20], [16] where, while maintaining a level of equality among workers, some are assigned additional tasks beyond the standard workload. These tasks, however, do not include decision-making responsibilities. Instead, they might involve more complex computations, data handling.

B. Communication and Model Updating in Serverless based Distributed ML

In the context of distributed training with serverless architectures, employing channels like queues and databases is essential due to the inherent stateless nature of serverless functions [36]. They enable the collection and dissemination of data generated during the distributed learning process, ensuring that the coordination and aggregation of computational tasks can be carried out.

LambdaML[20] and SMLT [16] exemplify this approach by utilizing a centralized database where all worker nodes store their computed gradients. This central database forms

the core communication hub, allowing workers to access and utilize gradients computed by others. However, both of these designs are generally of low efficiency due to the bandwidth bottleneck over the central database as well as over the one function responsible of making the aggregation of gradients of the differnet workers. The use of 'scatterReduce' method used in [20] and [16] aimed to mitigate this problem and instead of a single central function aggregating all gradients, the task is distributed among multiple nodes. Liu et al. in their solution FuncPipe [37] tried to reduce bottleneck communication over the functions responsible of making the aggregation by utilizing both y utilizes both uplink and downlink bandwidth of serverless functions

Contrastingly, MLLess [38] adopts a more composite strategy, integrating both a central database and individual queues for each worker. In this hybrid model, the central database is employed for storing gradients and facilitating the communication of these gradients among different nodes. The individual queues associated with each worker serve a distinct purpose – they act as notification channels, informing each worker about new updates and changes. To reduce overhead over the aggregated function MLLess only send significant updates.

Differentiating from these works, our architecture uses both queues and individual databases for each worker, rather than relying on a central database. Our design also incorporates gradients accumulation throw parallel gradients in order to further reduce the communication overhead. Additionally, we modify RedisAI [39] for serverless ML training systems that rely on databases, introducing in-database model updates ton eliminate the traditional fetch-process-reupload cycle.

C. Security and Robustness in Distributed ML

Fault tolerance in distributed ML is facilitated by techniques like Availability Zones, retries, architectural decisions, and checkpointing mechanisms [40], [41], [42], [19], [20]. The heartbeat technique is also used for failure detection [43]. Several studies also propose securing communication within distributed systems to prevent data leakage [17], [44], [23], [45]. In addition, robust aggregation techniques such as KRUM [27], MULTI-KRUM [27], GeoMed [28], MarMed [28], and ZENO [29] are used to address Byzantine faults [46]. These robust aggregations rules have been adapted to P2P architectures in works like the BRIDGE framework [26] and a blockchain-based solution by Xu *et al.* [23].

In the field of distributed machine learning, LambdaML [20] introduces a novel fault tolerance approach within individual workers. This system monitors execution to preemptively address a potential 15-minute timeout. As the timeout nears, LambdaML pauses execution, saving a checkpoint to the storage service, which includes the latest local model parameters. Execution is then resumed with a new worker trigger. Conversely, Mlless [38] enhances training robustness through a supervisor mechanism. This supervisor automatically removes workers whose contribution to model convergence is minimal or negative, primarily due to escalated communication costs.

We expand on this body of work by implementing a secure, scalable peer-to-peer communication, new participant integration into the ML training network, and incorporating robust aggregation mechanisms for reliable distributed ML.

III. Design Architecture of Logical Peer to Peer training ML

In this section, we delve into the design architecture of a serverless, peer-to-peer machine learning training system. The discussion encompasses a broad overview of the proposed architecture, an in-depth examination of the core components, and an exploration of the operational dynamics driving the system's overall functionality and performance.

A. Comprehensive Overview of the Proposed Architecture

Our proposed serverless P2P distributed training architecture, as illustrated in Figure 1, kicks off with system initialization, followed by the authentication of new peers, if any. In this configuration, every peer in the network is uniquely identified by the IP address and port tied to its corresponding stateful component - a dedicated Redis database. A heartbeat monitoring system ensures the constant availability of all peers.

Within the system, the assigned dataset of each peer is divided into smaller shards (batches). The peer computes gradient for each shard, averages them, and stores the result in its Redis database. These averaged gradients are then collectively aggregated among all peers, filtering out any outliers in the process by applying robust aggregation. The resultant set of trusted, aggregated gradients is used to update the model parameters. The system performs periodic checks for model convergence *i.e.,*every ten epochs, assuring optimal progress during the learning phase. Data integrity and confidentiality are ensured through secure peer communication.

The entire process is coordinated using AWS Step Functions, which seamlessly orchestrates the flow of each epoch within the training process of each peer.

This architecture is deployed on Amazon AWS for its unique benefits like the 15-minute timeout and 10GB RAM from AWS Lambda [47]. Notably, comparable services on platforms like Google Cloud, Azure, and IBM Cloud enable possible architecture replication.

B. Deep Dive into Core Architectural Components

Following the initial overview, we will now go over each key facet of our proposed Peer-to-Peer (P2P) serverless architecture.

*1) **Training Dataset Management and Partitioning**:* Individual peers have the ability to pull data from multiple distributed storage systems, including Amazon S3. The specific data each peer is responsible for is determined by its unique rank. This data is then divided into smaller units, or shards, to enable batch processing.

*2) **Leveraging Serverless Computing Across Peer Training Tasks**:* The cornerstone of our architecture is serverless computing, embodied by Amazon Lambda functions. Incorporated throughout the peer training workflow—from peer authentication to model updates—it offers benefits such as isolation

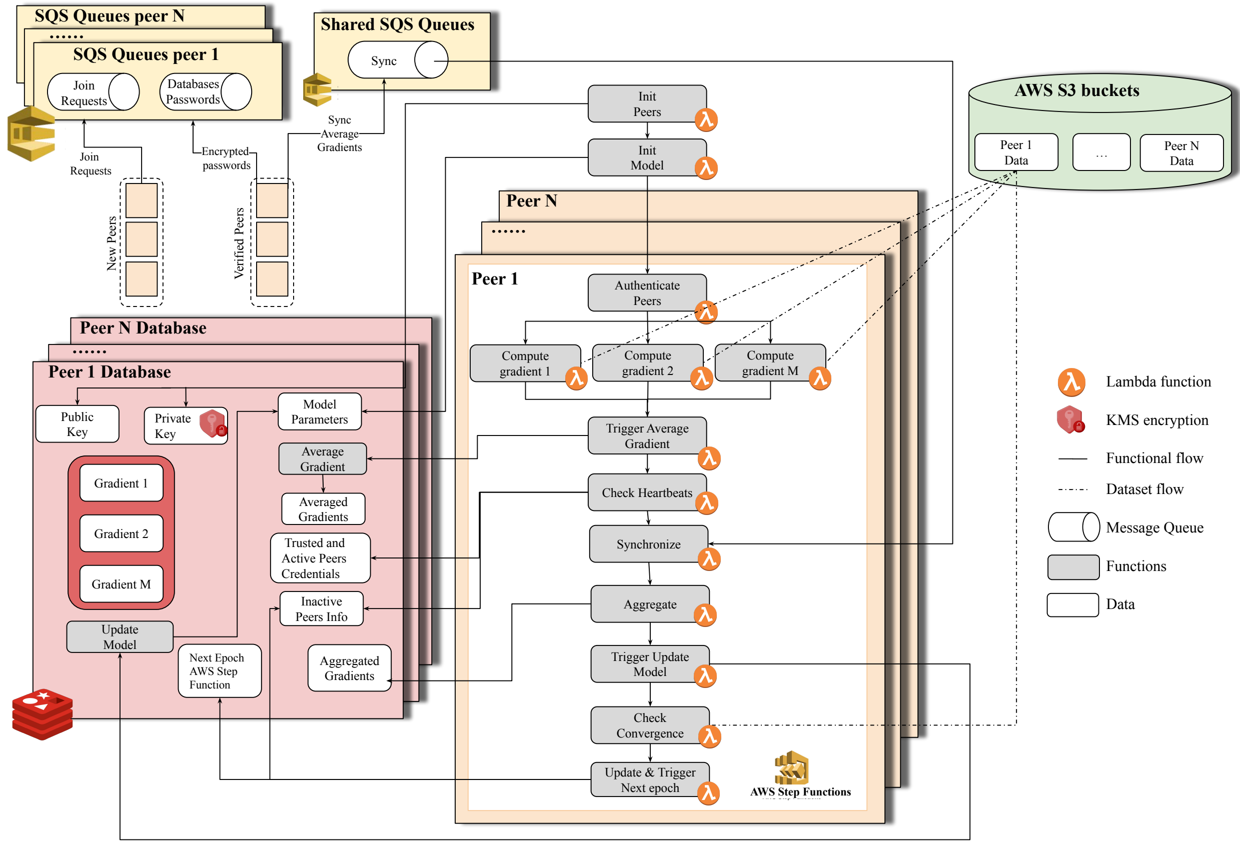


Fig. 1. Overview of the proposed Peer To Peer training based on Serverless computing

for uninterrupted operations, capacity for managing compute-intensive tasks like gradient computations, and scalability for workload fluctuations.

*3) **Training Workflow Orchestration**:* Due to the serverless structure of our system, where functions operate independently, their orchestration is crucial. To manage this, we use AWS Step Functions, a powerful serverless workflow service. It coordinates the entire machine learning training process within each peer during each epoch, including tasks like peers authentication, gradient computation and averaging, model updates, and convergence assessment. Notably, our architecture integrate continuous invocation of step functions for each epoch, which effectively mitigates AWS Lambda's cold start delays, thereby enhancing overall performance and minimizing latency during ML training.

*4) **State Management and Processing in Database**:* In our architecture, we use Redis, an open-source in-memory data store, for quick access to machine learning artifacts such as model parameters and gradients - a key requirement for any stateless distributed ML system.

Beyond a simple key-value store, we utilize the RedisAI module which supports various deep learning backends, enabling in-database ML operations, and minimizing data transfer latency. RedisAI is especially efficient at serving models at scale and in real-time. Unique to our architecture is the extension of RedisAI's capabilities to directly modify model parameters within the database, eliminating the traditional process of external processing, our routine performs these operations inside the database itself.

*5) **Synchronization between Peers**:* Within our proposed architecture, achieving synchronization amongst peers is paramount for ensuring the correctness of the distributed training process. To manage this aspect of distributed computation, we employ the AWS Simple Queue Service (SQS).

Once a peer completes gradient computation for its data shards and averages local gradients, it sends a notification message to a designated synchronization queue, the "Sync Queue", signifying the task completion. If a peer doesn't respond or acknowledge within a designated timeout period, others proceed without waiting indefinitely. The unresponsive peer is identified as a failed node in the next epoch by our heartbeat monitoring system.

We note that the messages inside the "sync queue" will be deleted by any peer in initialisation phase.

*6) **Secure Communication: Safeguarding Data Integrity and Confidentiality**:* Our architecture employs stringent secure communication protocols to ensure data integrity and confidentiality during inter-peer interactions, accomplished through the RSA algorithm for asymmetric encryption, with unique public and private keys for each peer. Beyond encryption, unique digital signatures derived from private keys authenticate sender identity and verify data integrity.

Each peer's private key is safeguarded by encryption using a unique key from AWS Key Management Service (KMS), with access strictly limited to few authorized services (Lambda functions), enhancing security against unauthorized access.

C. Operational Dynamics of the Proposed Architecture

Within this subsection, we take a closer look on the operational dynamics that power our proposed architecture. Moving beyond the standalone examination of the key components, to their interactions, collaborative processes, and the mechanisms that drive the overall system performance and functionality.

*1) **Peers Initialization and Authentication**:* Our architecture relies on Amazon SQS for two main operations: *peers' initialization* and *new peers' integration*. Each peer has two distinct SQS queues - the first for join requests and the second for receiving encrypted passwords of other peers' databases. Every peer also maintains an AWS Key Management Service (KMS) encryption key, securing its private key within its database, and ensuring exclusive key access.

Peers Initialisation This phase involves the setup of individual peers. The process below describe the peers initialisation.

1) At the onset, the admin initiates the peers, by providing each their own KMS encryption key, URLs of their neighboring peers' "join requests" SQS queues, and a unique rank. Each peer then generates private and public keys. The public key is stored in its plain form, while the private key is encrypted using the KMS encryption key before storage in the respective database.
2) Each peer then generates a digital signature and broadcasts this, along with its public key, database IP address, port, and the URL of its "databases passwords" queue inside the other peer's "join requests" queue.
3) Verification ensues as each peer validates the other's signature
4) Upon successful verification, peers mutually exchange their encrypted database passwords. Furthermore, they save each other details, including their rank, into their respective databases.

Novel peer integration In this phase, we illustrate the process of integrating a new peer into an existing training network. The steps are as follows:

1) The process is initiated when the admin provisions the new peer (Peer 3) with the URLs of the "join requests" SQS queues of existing peers and their corresponding public keys, and their ranks. Peer 3, in response, generates a pair of public and private keys. These keys are stored in its database, with the private key encrypted using the designated KMS encryption key.
2) The new peer (Peer 3) generates a digital signature, broadcasting it and its public key, database port and IP address, URL of its "databases passwords" queue, and rank. It sends its password, encrypted with the recipient's public key, via their "join request" queues.
3) Once the broadcasting phase is completed, Peer 3 waits for validation from the existing peers. These peers undertake the task of validating Peer 3's authenticity. They accomplish this by rigorously comparing the signature provided by Peer 3 against the information contained in its public key, ensuring a match.
4) After Peer 3's successful validation, Peers 1 and 2 send back their individual signatures, and their databases passwords encrypted with Peer 3's public key, into Peer 3's "databases passwords" queue. Furthermore, they incorporate Peer 3's details into their respective databases.
5) Finally, Peer 3 validates the sender peers based on their signatures and public keys. Upon successful validation, Peer 3 records the details of Peers 1 and 2 into its own database.

*2) **Model Initialisation**:* The model initialization phase involves the establishment of a unified model to serve all initialized peers. This model can be initialized with random parameters or pre-trained ones, depending on the requirements. The chosen model, which could be a specific ML or deep learning model, is then stored in each peer's Redis database using RedisAI. This ensures a consistent starting point for distributed learning.

*3) **Distributed Gradient Computation**:* Leveraging the serverless concurrent abilities of AWS Lambda, we implement a parallel gradient computation within our architecture. Each peer partakes in this distributed process by calculating gradients on assigned data batches and storing these computed gradients in its local Redis database. To expedite the gradient computation, data, segmented into smaller batches, is fetched from S3 storage, and model parameters are retrieved from Redis.

*4) **Averaged Gradient Computation**:* Once the gradients have been computed, an embedded Lua script calculates the gradients' average within the Redis database environment, capitalizing on in-database programming. This approach eliminates costly external data transfers. Once local averaging is complete, peers send a completion message to the ”sync queue“, notifying others of the task's conclusion.

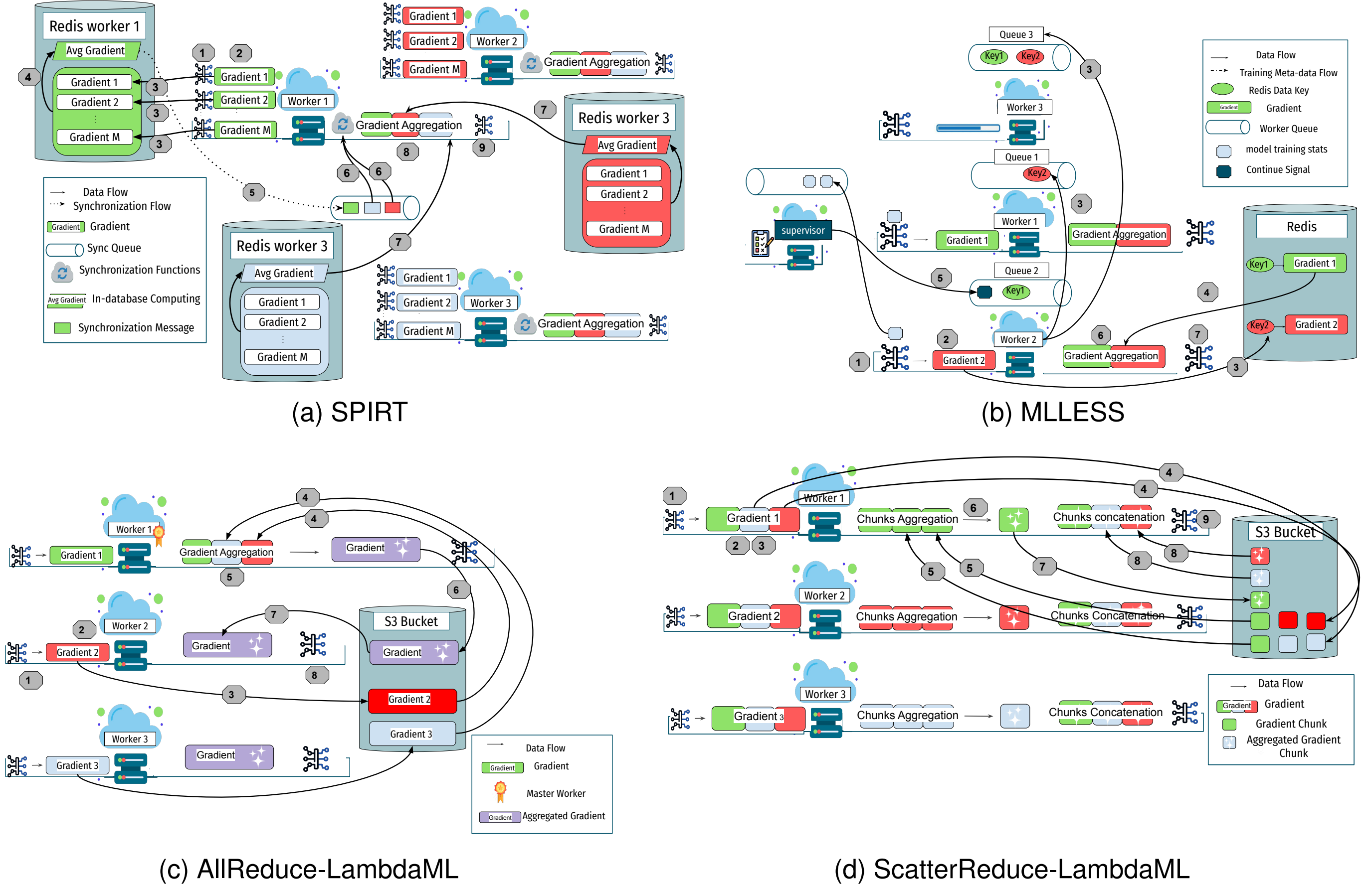
*5) **Heartbeat Monitoring**:* Our system incorporates a 'heartbeat' mechanism designed to be triggered every epoch, where each peer checks the operational status of other peers databases. Peers send a signal, waiting for responses to confirm others' activeness. Failure to respond within a set timeframe and a number of trials denotes a peer as ”inactive“, subsequently removed from the trusted peers list and added to the ”inactive peers info“ list. This continuous health check is vital for network integrity and uninterrupted peer-to-peer communication.

*6) **Peers Synchronization**:* To collate the individually computed average gradients from each participating peer and derive the aggregated gradient, a specially designated Lambda function ”synchronize“ is instantiated. This function serves as a synchronization barrier that waits until the count of messages in the queue equals the current number of active peers, those determined by the preceding heartbeat check.

*7) **Gradient Aggregation**:* Once all peers are synchronized, the gradient aggregation phase begins. Each peer fetches the average gradients from the databases of all active peers in the network. An aggregation function then amasses these gradients, utilizing robust algorithms to discard outlier gradients. The final aggregated gradient is then stored in each peer's Redis database. This approach ensures the integrity and accuracy of our gradient aggregation process.

*8) **Model Update**:* Utilizing the in-database programming feature of RedisAI, we directly update each peer's model

Fig. 2. Comparative Overview of Serverless Machine Learning Frameworks: SPIRT, MlLess, LambdaML Scatter Reduce, and LambdaML AllReduce



(a) SPIRT

(b) MLLESS

(c) AllReduce-LambdaML

(d) ScatterReduce-LambdaML

parameters stored in the Redis database using the aggregate gradient, thereby bypassing the conventional read-process-write cycle.

*9) **Convergence Checking**:* Upon the completion of model parameter updates, a Lambda function is intermittently called (e.g., after every tenth iteration) to check model convergence. This approach reduces unnecessary function calls, as significant model changes aren't anticipated after each iteration.

*10) **Update and Trigger new epoch**:* In our AWS Step Function workflow, each function manages a single epoch. A Lambda function triggers a new Step Function at each epoch's end to continue training. This Lambda initializes the Step Function with the next epoch number, parallelism level for gradient tasks, and a convergence check flag, updating it as needed. The new function's ARN is stored in a Redis database for reuse.

The same Lambda function updates the inactive peer list through a consensus approach, marking a peer as inactive only if all nodes list it as such.

*11) **Fault Tolerance and Peer Data Redistribution**:* A core strength of our architecture lies in its fault-tolerance mechanisms. This resilience is primarily manifested through our approach to handle instances of peer inactivity or failure. When one or many peers become inactive, our dedicated Lambda function first identifies these instances using the consensus-based approach described previously. Once inactive peers are identified, the data originally assigned to the downed peers is segmented and distributed among the active peers based on a predefined ranking system. Here, each peer, according to their rank, inherits a corresponding portion of the data from the inactive peer.

Following this data reassignment, the "Update and Trigger new epoch" Lambda function adjusts the configuration of the new Step Function to account for the change in data distribution and the increased workload for each active peer. This adjustment may include tuning the degree of parallelism for gradient computation tasks to accommodate the additional data batches.

IV. Comparative Analysis of Serverless ML Frameworks

In this section, we conduct a systematic comparison of serverless machine learning training frameworks, specifically SPIRT [30], MLLess [15], and the two proposed communication patterns proposed by LambdaML [20]. By illuminating their unique architectural elements, we establish a foundation for comprehensive analysis and comparative evaluation to break down and elucidate their communication protocols and interaction patterns.

A. Serverless ML Training Workflow

Training in a serverless computing environment entails distinct characteristics due to the stateless nature of serverless

TABLE I
COMPARATIVE ANALYSIS OF SERVERLESS TRAINING FRAMEWORK ARCHITECTURES: AN OVERVIEW OF KEY TRAINING COMPUTATIONAL STAGES

Training Stages	SPIRT [30]	MLLESS [15]	Scatter Reduce [20]	ALL Reduce [20]
Fetch Dataset	1. Fetch a worker's minibatches.	1. Fetch a minibatch.	1. Fetch a minibatch.	1. Fetch a minibatch.
Compute Gradients	2. Compute in parallel the gradient of each minibatch. 3. Send gradients to own database. 4. Calculate the average of computed gradients within database.	2. Compute gradient of minibatch.	2. Compute gradient of minibatch.	2. Compute gradient of minibatch.
Synchronisation	5. Notify the queue that the gradient is ready. 6. Poll the sync queue until the messages count matches the number of workers. 7. Fetch all averaged gradients from workers. 8. Aggregate retrieved gradients and save it in database.	3. Store gradient in database and send its key to other workers' queues. 4. Listen to the queue and store received update keys 5. Wait until the expected updates from supervisor arrives. 6. Fetch and aggregate gradients until all expected updates are received.	3. Divide the gradient into chunks for each worker. 4. Retain own chunk, send others to database. 5. Fetch and aggregate assigned chunks, gathered from others. 6. Send the aggregated chunk to the database. 7. Retrieve all aggregated chunks. 8. Concatenate aggregated chunks to assemble the full gradient.	3. Send gradient to database. 4. Master worker retrieves all gradients from the database 5. Master Worker performs the aggregation process. 6. Master worker sends aggregated gradient to the shared database. 7. Fetch aggregated gradient.
Model Update	9. Update the model.	7. Update the model.	9. Update the model.	8. Update the model.

functions. These functions necessitate an external storage solution for preserving intermediate data that must be accessible across different workers or for storing training checkpoints prior to the termination of a function. The following algorithm delineates the serverless ML training workflow.

Algorithm: Typical Serverless ML Training workflow

1: **Data Fetch:** Each worker fetches its dataset partition.
2: **Parallel Training:** Independently, workers train on local batches and compute gradients.
3: **Gradient Sharing:** Workers upload their computed gradients to a shared database.
4: **Gradient Collection:** Workers fetch gradients from the shared database.
5: **Gradient Aggregation:** Aggregate all fetched gradients to compute a global update.
6: **Model Update:** Update each local model with the aggregated gradients.

B. Framework-Specific Communication Mechanisms

Serverless computing has significantly transformed distributed machine learning (ML) with innovative frameworks like SPIRT, LambdaML, and MLLess, each uniquely addressing the complexities of serverless architectures. Figure 2 illustrates the distinct communication mechanisms employed by these frameworks to enhance distributed ML tasks in a serverless environment.

In **SPIRT (Figure 3a)**, each worker is considered as a peer. Each worker has its own database and a serverless workflow orchestrated by aws step function to perform the following operations: **(1)** each worker fetches the minibatches assigned to it, **(2)** each minibatch is then utilized to compute gradients in parallel, **(3)** these gradients are subsequently stored in the worker's own database. The process continues with **(4)** the averaging of these gradients within each worker's database. Following this, **(5)** a notification is sent to the synchronisation queue indicating the completion of gradient averaging. Workers **(6)** poll the synchronization queue until the message count aligns with the number of peers involved. Subsequently, **(7)** peers retrieve the averaged gradients from each other's databases, **(8)** aggregate these averages and save it within database, and finally, **(9)** proceed to update their local models within their databases.

In **MLLESS** (Figure 3b), the workflow begins with **(1)** each worker fetching a minibatch. Following this, **(2)** the workers compute the gradient of their respective minibatches. Upon identifying a significant update, **(3)** the gradient is stored in a shared database, and its key is sent to the queues of other workers, while simultaneously notifying the supervisor of the update by posting in the supervisor's queue. The next phase, **(4)**, involves workers continuously monitoring their queues to accumulate the keys of received updates from other workers and to read from supervisor the expected updates. **(5)** Workers then wait until all the expected updates, as communicated by the supervisor, have arrived. **(6)** They then fetch the corresponding gradients from the database and aggregate them once all expected updates, as indicated by the supervisor, have been received. Finally, **(7)** the aggregated gradients are used to update the model.

In **ScatterReduce-LambdaML (Figure 3d)**, each worker begins by **(1)** fetching a minibatch of the dataset to process. They then **(2)** compute the gradients of this minibatch and **(3)** divide the computed gradient into chunks, each intended for a different worker. Each worker **(4)** keeps its respective chunk and sending the others to a shared database. Workers then **(5)** fetch and aggregate the chunks assigned to them, which have been gathered from other workers, This aggregated chunk is **(6)** sent back to the database. Each worker **(7)** retrieves all the aggregated chunks from the database, **(8)** concatenates these aggregated chunks to assemble the full gradient, and finally, **(9)** updates the model with this complete gradient.

In **AllReduce-LambdaML (Figure 3c)**, each worker begins

TABLE II
COMPARATIVE ANALYSIS OF SERVERLESS MACHINE LEARNING FRAMEWORKS: FEATURE COMPARISON ACROSS MLLESS, LAMBDAML, AND SPIRT

	SPIRT [30]	MLLess [15]	AllReduce [20]	ScatterReduce [20]
Gradients Storage	Each worker computes gradients and stores them in their own designated database.	Gradients are stored in a central database and identified with unique keys.(Redis Database)	Gradients are stored in a central database. (S3 bucket)	
Communication Channels	Workers employ a central queue to receive completion notifications and directly access gradients stored in other workers' databases.	Workers employ a single queue for both supervisor instructions and receiving gradient keys from other workers.	Workers transmit their individual gradients to the central database for storage and access the aggregated gradient computed by the master worker.	Workers transmit their calculated gradient chunks to the central database and collect corresponding chunks computed by other workers.
Communication Overhead Reduction	Parallel accumulative learning and in-database batch averaging, coupled with enhanced RedisAI for in-database model updates.	Workers accumulate local updates and share them with other workers only upon reaching a significance threshold.	Scatter-reduce was proposed as a solution to the bottleneck issue inherent in AllReduce, by decentralizing the aggregation process and thus distributing the workload more evenly among workers.	
Synchronization Barrier	Workers are blocked until the central synchronization queue's notification count aligns with the number of workers.	Workers are blocked from advancing until their synchronization queue has the expected number of messages from the supervisor, ensuring all gradients are computed.	Workers wait until the master node uploads the aggregated gradients in the database	Workers pause proceeding only when the number of gradient chunks matches the worker count
Batch Processing	Each worker is assigned a dataset segment, organized into minibatches in the cloud, leveraging parallel batch processing.	MLLess processes batches by first preprocessing and storing the data as minibatches in cloud storage. Each worker then sequentially processes their assigned minibatches.	The dataset is preprocessed and segmented based on the number of workers, is processed in minibatches by each worker using the training loader.	
Fault Tolerance	**Worker down:** training continue with existing workers. **Recovery:** involves detecting inactive workers through heartbeat monitoring. Active workers then redistribute the data initially assigned to the unavailable peer among themselves.	**Worker down:** training continue with existing workers. **Supervisor down:** workers blocked until supervisor come back. **Recovery:** None.	**Worker down:** Master worker will be blocked until the worker come back. **Master Worker down:** workers blocked until Master worker come back. **Recovery:** None.	**Worker down:** All workers will be blocked until the worker come back. **Recovery:** None.
Auto-Scaling	A worker can be added at any point of the training without affecting the other workers.	None.	None.	None.
Security Measures	Implemented robust aggregation techniques, ensured worker authentication upon network entry, and secured communications through encryption.	None.	None.	None.

by **(1)** fetching a minibatch from the dataset. Following this, they **(2)** compute the gradients of the minibatch. Once the gradients are computed, workers **(3)** send these gradients to a shared database. One of the workers will be designed as master (usually worker with ID 1). The master worker then **(4)** retrieves all the gradients from the database and **(5)** performs the aggregation process to combine these gradients into a single, unified gradient. After the aggregation, the master worker **(6)** sends this aggregated gradient back to the shared database. Subsequently, each worker **(7)** fetches the aggregated gradient from the database, and finally, **(8)** updates their local models with this aggregated gradient.

Exploring these frameworks in a detailed comparative study will highlight the unique conceptual approaches they adopt for orchestrating machine learning training processes.

C. Comparative Review of Serverless ML Frameworks

We evaluate the frameworks using a set of criteria that highlight their distinctive operational and architectural characteristics, specifically addressing the intricacies of serverless computing and ML training, as follows:

- **Gradient Storage:** This criterion evaluates the strategies utilized for the preservation and accessibility of computed gradients within distributed machine learning systems.
- **Communication Channels:** This aspect evaluates the mechanisms through which workers communicate with each other. It looks at the types of channels used (e.g., queues, direct database access) and how it facilitate the transfer and retrieval of gradients.
- **Communication Overhead Reduction:** This criterion examines the strategies and techniques used to minimize the amount of communication required between workers.
- **Synchronization Barrier:** This assesses the synchronization mechanisms used to coordinate task progression among distributed workers, including conditions and constraints like conditional waits.
- **Batch processing:** This describe how workers manage data batches during training, from sequential to parallel processing. In serverless environments like AWS Lambda, which use CPU-based execution [48], the parallelization within Lambda functions is constrained by their single-threaded design, resulting in sequential processing.
- **Fault Tolerance:** This evaluates the robustness of a distributed system against worker or supervisor node fail-

ures, detailing strategies for operational continuity, state recovery methods, and resilience mechanisms that enable the system to maintain or swiftly restore its functions.

- **Auto-Scaling:** This measures the system's ability to dynamically adjust the number of active workers during the training process without requiring a restart, allowing for seamless integration or removal of workers.
- **Security Measure**: This criterion assesses the measures to protect data integrity and confidentiality, including encryption protocols, authentication mechanisms, and other practices to prevent unauthorized access and breaches.

V. Evaluation

In order to evaluate the performance of different serverless ML training architectures, including SPIRT, we design a series of experiments to evaluate the performance of various CNN models across different datasets on the proposed architectures.

A. Experimental Setup

1) Datasets: We utilized two public datasets:

MNIST: The MNIST Handwritten Digit Collection [49] consists of 60,000 handwritten digit samples, each belonging to one of ten classes.

CIFAR: The CIFAR Image Dataset [50] encompasses 60,000 color images spanning ten distinct classes, such as automobiles, animals, and objects. Each category contains 6,000 images that are evenly distributed.

2) Model Architectures and Hyperparameters: The experiments involve three different CNN models:

MobileNet V3 Small: A lightweight CNN developed for mobile and edge devices, it features inverted residual blocks, linear bottlenecks, and squeeze-and-excitation modules, with roughly 2.5 million trainable parameters [51].

MobileNet: MobileNet (MN) is a neural network model that uses depth-wise separable convolutions to build lightweight deep neural networks. The size of each input image is 224×224×3, and the size of model parameters is 12MB.

ResNet-18: A deep learning CNN model with approximately 11.7 million parameters, featuring 18 layers and using "skip connections" to aid training of deeper networks [52].

ResNet-50: A more advanced deep learning CNN architecture, ResNet-50 encompasses approximately 25.6 million parameters across 50 layers [52].

B. AWS Lambda Configuration

Several AWS Lambda functions, discussed in Section III, were set up for the training procedure. Dependencies such as PyTorch, NumPy, Redis, RedisAI, and sshtunnel were necessary. Managing these dependencies posed a challenge due to AWS Lambda's deployment package size limit. The unzipped files cannot exceed 250MB.

C. Dataset Division for Workers and Batches

During experiments with datasets like MNIST, the dataset is distributed among the workers, with each receiving a subset. Workers then split their subset into batches for processing. For instance, if MNIST is shared among 4 workers using a batch size of 128, each worker processes about 15,000 images across roughly 118 batches to compute gradients.

VI. Results

In order to assess the effectiveness of various serverless ML training architectures, such as SPIRT, we design a series of experiments to evaluate the performance of various CNN models across different datasets on the proposed architectures.

A. Serverless Frameworks Training Time

Motivation: The motivation behind this experiment is to delve into the intricate temporal dynamics that characterize the various architectures. By benchmarking the duration of the distinct ML training stages outlined in Table IV-A, we aim to uncover the inefficiencies and inherent strengths of each architecture. This comparative analysis can guide practitioners in selecting the most suitable framework for their specific needs.

Approach: To evaluate training times across serverless architectures, we conducted a single-epoch training using 4 workers, tracking the time spent at each stage. Depending on the architecture, the serverless lambda maximum memory size was determined based on tests. We executed the lambda function, which subsequently reported the maximum memory usage observed during the operation. This allowed for a precise assessment of the memory demands of the training process. We trained the MobileNet model on the CIFAR dataset, providing a basis for comparing architectural efficiency.

Results: We observed distinct performance characteristics across the stages of fetching the dataset, computing gradients, synchronization, and model updating.

Spirt has the advantages to speed up its epoch by parallelizing the batch processing, with notably rapid dataset fetching (1.2s) and gradient computation (18.055s). To complete one epoch, Spirt needs one synchronisation between workers (6.03s) and one model update (0.28s).

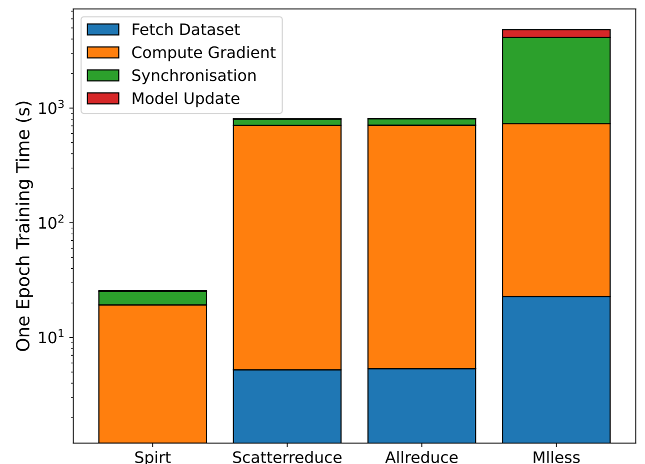


Fig. 3. Training time for one epoch across serverless training frameworks, depicted on a logarithmic scale.

TABLE III
COST ESTIMATIONS OF ONE EPOCH TRAINING USING MOBILENET ON CIFAR WITH 4 WORKERS ACROSS VARIOUS ML TRAINING FRAMEWORKS IN SERVERLESS COMPUTING, EMPLOYING A BATCH SIZE OF 256 TO YIELD 49 BATCHES

Framework	Training Stage	Time (s)	RAM	Cost/Worker (USD)	Cost 4 Workers (USD)
SPIRT [30]	Initialisation	5.23	128	<0001	<0001
	Fetch Dataset + Compute Gradient	15.44	3048	0.0298*	0.1193
	Trigger Average Gradients	3.815	128	<0001	<0001
	Synchronisation	3.273	128	<0001	<0001
	Aggregation	2.759	168	<0001	<0001
	Trigger Model Update	0.28	128	<0001	<0001
	Total per Epoch			0.0298	**0.1194**
ScatterReduce [20]	Fetch Dataset	5.232			
	Compute Gradient	49 * 14.343			
	Synchronisation	49 * 1.920			
	Model Update	49 * 0.163			
	Total per Epoch	810	4880	0.0514	**0.2056**
AllReduce [20]	Fetch Dataset	5.349			
	Compute Gradient	49 * 14.382			
	Synchronisation	49 * 1.9516			
	Model Update	49 * 0.12			
	Total per Epoch	811.5757	4986	0.0526	**0.2104**
MLLEss [15]	Fetch Dataset	49 * 0.463			
	Compute Gradient	49 * 14.472			
	Synchronisation	49 * 69.425			
	Model Update	49 * 10.291			
	Total for Worker	4638	5389	0.3254	1.3016
	Total for Server	4638	204	0.0123	0.0123
	Total per Epoch				**1.3139**

*Cost per worker, multiplied by 49 for concurrent lambdas.

For the remaining frameworks—Mlless, Scatterreduce, and Allreduce—the times were calculated based on executing 49 discrete steps for each of the training stages.

The MLLess framework exhibited the longest training epoch duration, particularly due to the time-consuming synchronization process. Each of the 49 steps required for synchronization took about 69.425 seconds. This was because MLLess needed to communicate updates with a supervisor and other workers, verify its queue, and begin retrieving shared update keys from other workers. It then waited to receive all the expected updates, processed each one sequentially, fetched the update from the database using the received key, and sent it for model update. The model update for MLLess took 10.29 seconds for each of the 49 steps, since every time it fetched a gradients update from the database, it updated the model parameters.

Each worker in ScatterReduce and Allreduce are fetching the dataset assigned for him, taking about 5 seconds. Compute gradient were reported similar to reach almost 14 seconds for each of the 49 steps, the synchronisation as well, is almost the same with 1.9 seconds.

ScatterReduce and Allreduce presented similar performance profiles, with each worker responsible for fetching its portion of the dataset, taking about 5 seconds on average. The computation of gradients was consistent across these frameworks, with each of the 49 steps taking nearly 14 seconds. Synchronization times were comparable as well, at approximately 1.9 seconds for each step.

B. Serverless Frameworks Cost analysis

Motivation: The motivation behind this experiment is to analyze the cost implications of training distributed machine learning models across various frameworks. We focus on the detailed cost associated with each architectural element. Specifically, we examine the pricing of compute instances (e.g., AWS Lambda functions), database interactions (e.g., read/write operations in Redis), and communication services (e.g., RabbitMQ). The objective is to provide a granular cost breakdown that highlights the financial impact of architectural decisions within serverless environments.

Approach: Building on the insights from our previous training time experiment, we assessed the costs associated with operating serverless functions, taking into account the allocated memory RAM and execution duration. Additionally, we evaluated the expenses related to various components, including data storage for training datasets, database communication channels, state management functions, and communication

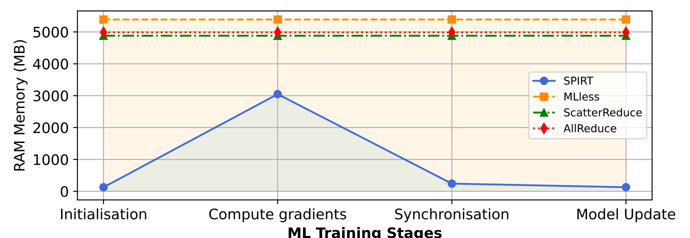


Fig. 4. Memory Allocation in Serverless Computing for Training Workflows: A Comparison between SPIRT and Other Serverless Training Frameworks.

TABLE IV
COST ESTIMATION OF ONE EPOCH TRAINING FOR VARIOUS ARCHITECTURAL COMPONENTS OF SERVERLESS TRAINING FRAMEWORKS FOR MOBILENET ON CIFAR WITH 4 WORKERS, UTILIZING A BATCH SIZE OF 256.

Framework	Component	Estimation Cost (USD)
SPIRT	Data Storage (S3)	/
	Redis Communication Channel	0.0014 (0.17 / hour) * 4
	Lambda Functions	0.1194
	Step Function	0.001
	Queue (RabbitMQ)	0.0002 (0.027 / hour)
	Total	**0.1306**
ScatterReduce	Data Storage (S3)	/
	S3 Communication Channel	0.14
	Lambda Functions	0.2056
	Total	**0.3456**
AllReduce	Data Storage (S3)	/
	S3 Communication Channel	0.16
	Lambda Functions	0.2104
	Total	**0.3704**
MLLess	Data Storage (S3)	/
	Redis Communication Channel	0.0618 (0.048 / hour)
	Lambda Functions	1.3139
	Queue (RabbitMQ)	0.0348 (0.027 / hour)
	Total	**1.4105**

queues. Notably, for the SPIRT architecture, we considered the cost implications of using an EC2 instance to host the sophisticated Redis database.

Serverless computing usage varied across the studied frameworks. Before executing a function, it is necessary to preset the desired memory size for the function. For instance, in SPIRT, a different memory size was utilized for each stage of the training workflow. In contrast, other frameworks employed a uniform memory allocation for the entire training workflow. Previous work [18] revealed that gradient computation is the most resource-intensive stage of the training workflow. Using a single function for the entire training would necessitate allocating a function with memory that may not be fully utilized across different training stages. The maximum RAM usage was determined for each framework through testing, where each framework's lambda function reported its peak memory usage. Figure 4 illustrates this process.

In our evaluation, we utilized a setup comprising 4 workers and employed the MobileNet model, trained on the CIFAR dataset, as a benchmark to compare cost efficiency across different architectural configurations. It's important to note that all cost estimations for Lambda are based on AWS's publicly available pricing information [53]. Additionally, we used the AWS Pricing Calculator to precisely estimate the costs associated with each framework [54].

Tables III and IV provide detailed cost analyses of serverless functions and training architectures across various machine learning frameworks, respectively. Both tables focus on the specific scenario of training MobileNet for one epoch on the CIFAR dataset using four workers with a batch size of 256.

Results: In the SPIRT framework, we a detailed the execution time and the memory requirements for each function involved in the training stages. The most costly operation was the parallel computation of gradients, priced at $0.0298 per function. This cost was then multiplied by 49 to account for the concurrent parallel functions actively computing the gradients, which contributed significantly to the final recorded Lambda function cost of $0.1194. We note that SPIRT was relying on the database to realise several computations which explain the low RAM overhead on the serverless computing.

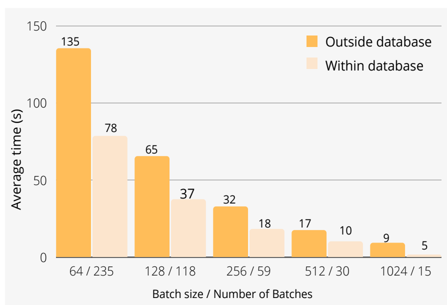
In contrast, the ScatterReduce, AllReduce, and MLLess frameworks utilized Lambda functions differently, assigning a function to each worker for the execution of the entire training stage. This methodology required tracking the total training time for one epoch and the maximum memory usage to estimate the costs. Specifically, ScatterReduce incurred a Lambda cost of $0.2056, AllReduce was slightly higher at $0.2104, and MLLess was more costly at $1.3139.

In the total framework cost evaluation, we did not consider the cost of Data Storage (S3), since we are using the same CIFAR dataset to evaluate all the frameworks.

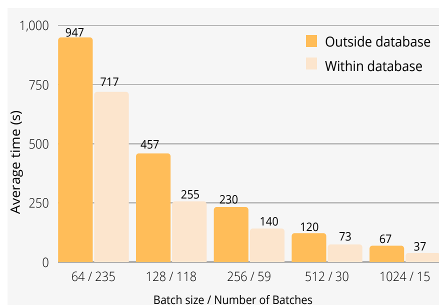
The SPIRT architecture employs several components, each contributing to an overall cost of $0.1306, computed based on the total epoch execution time of 30.797 seconds. It utilizes a modified RedisAI database on a c5.xlarge instance, which includes 4vCPUs and 8GB of memory, costing $0.17 per hour and amounting to approximately $0.0014 for four hours of usage. Lambda functions in the architecture cost $0.1194. Workflow orchestration is managed through AWS Step Functions, with 9 step transitions across 4 workers, costing a total of $0.001. Additionally, RabbitMQ is used for queue management, costing $0.027 per hour, with a brief usage cost of about $0.0002.

To accurately estimate S3 costs, we calculate both the volume of data exchanged and the number of queries involved. In the ScatterReduce architecture with four workers, each worker uploads 4 chunks per step (3 initial and 1 aggregated), resulting in $4n$ uploads per step, where n is the number of workers. Each worker also downloads 6 chunks per step (3 initial and 3 aggregated), leading to $6n$ downloads per step. Over an epoch consisting of 49 steps, this amounts to $196n$ uploads ($4n \times 49$) and $294n$ downloads ($6n \times 49$). Given each chunk is approximately 3.075 MB (from a gradient divided into four parts of 12.3 MB total), the total upload data per epoch is approximately $602.8n$ MB and the download data is approximately $904.2n$ MB. For four workers, this results in $2.41\,$GB uploaded and $3.62\,$GB downloaded, culminating in a total data transfer of $6.03\,$GB per epoch.

In the AllReduce architecture with four workers, the formulas governing data transfers per epoch can be described as follows: Each worker uploads its gradient to S3, and the rank 0 worker aggregates these and uploads the result, resulting in 5 uploads per step ($n+1$, where n is the number of workers). Each worker, except rank 0, downloads the aggregated gradient while rank 0 downloads the gradients from the other workers, totaling 6 downloads per step ($2 \times (n-1)$). Over an epoch of 49 steps, this translates to 245 uploads (5×49) and 294 downloads (6×49). Given each gradient is 12.3 MB, the total upload data per epoch is 3013.5 MB and the download data



(a) MobileNet V3 Small Model



(b) Resnet-18 Model

Fig. 5. Time taken for calculating gradient averages within and outside the database

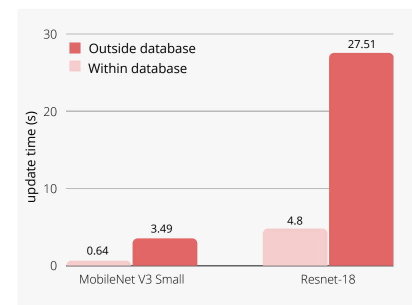


Fig. 6. Time taken for Model Update within and outside the database

is 3616.2 MB, resulting in a total data transfer of 6.63 GB.

The higher total data transfer in AllReduce stems from the fact that each data transfer involves larger amounts of data (whole gradients), compared to ScatterReduce where the data is broken into smaller chunks. These calculations are essential for estimating costs related to data storage and transfer on S3, considering both the volume of data exchanged and the number of operations performed.

MLLess scored the highest price due to the lengthy duration of 4638 seconds required to complete one epoch, which in turn increased the usage of cloud services. The breakdown shows costs for various components: no significant cost for Data Storage (S3), moderate expenses for Lambda Functions at $1.3139, and smaller amounts for the Redis Communication Channel and RabbitMQ Queue, at $0.0618 and $0.0348 respectively. The total accumulated cost reached $1.4105.

C. Communication Overhead Reduction

1) SPIRT Parallel Batch Processing:

Motivation: In training ML models, the learning process involves incrementally updating the model by processing data in batches. These batches are then processed in sequence, requiring frequent updates to be communicated between workers—a process that can introduce significant overhead.

Our proposed methodology, however, takes a different path. After distributing the initial dataset among workers, we further split these segments into minibatches. The key difference lies in processing these mini-batches in parallel within each worker. Rather than communicating updates after processing each mini-batch, we aggregate the gradients from all mini-batches and communicate these accumulated updates only once all parallel processing is complete.

TABLE V
COMPARATIVE ANALYSIS OF TIME AND COST WITH SEQUENTIAL VS PARALLEL BATCH PROCESSING

Batch size / # of batches	Time (sec)		Cost (USD)	
	Sequential	**Parallel**	Sequential	**Parallel**
64 / 235	394.8	**10.5**	0.01017	**0.05435**
128 / 118	330.4	**12.9**	0.00851	**0.03451**
512 / 30	278.4	**28.1**	0.00717	**0.03069**
1024 / 15	258	**47.8**	0.00665	**0.03567**

Approach: In previous work [18], we evaluated the performance and cost-efficiency of training the VGG11 model on the MNIST dataset using two distinct computational architectures. The first architecture employs a traditional sequential approach, where the entire training process is conducted on a single base instance, relying on sequential processing. The second architecture adopts a parallel, serverless-based approach, leveraging distributed batch processing. In the cost comparison analysis, the estimated cost per worker was calculated as follows:

$$\begin{aligned}\text{Cost per worker}_{\text{parallel}} = &[\text{Lambda Cost} \times \text{Num of batches} \\ &+ \text{Trigger Instance Cost}] \\ &\times \text{Computation Time}\end{aligned} \quad (1)$$

$$\text{Cost per worker}_{\text{sequential}} = \text{Instance Cost} \times \text{Computation Time} \quad (2)$$

Results: Our results in Table V highlight the speed advantage of serverless parallel processing. For instance, with a batch size of 64, the serverless approach curtailed computation time from 394,8 seconds (as seen in the single-machine approach) to a mere 10,5 seconds. This significant reduction in computation time persisted across all batch sizes. However, this efficiency comes at a slightly higher cost, with the serverless approach incurring $0.05435 per peer for the same batch size, compared to $0.01017 per peer in the traditional model.

2) SPIRT within Database Operations:

Motivation: As we delve deeper into the intricate world of distributed serverless environments, characterized by numerous autonomous and stateless services, the challenges associated with frequent database retrieval loads during training phases emerge. While works such as previous work [18], LambdaML [20], SMLT [16], and MLLess [38] utilize a database as a communication channel, storing and retrieving model parameters as necessary, the exploration of the communication overhead these operations create remains largely unexplored. Such conditions often precipitate a significant increase in communication overhead, consequently undermining the overall training performance. However, within our unique architectural framework, we incorporate a customized Redis. This component allows us to perform average and update operations

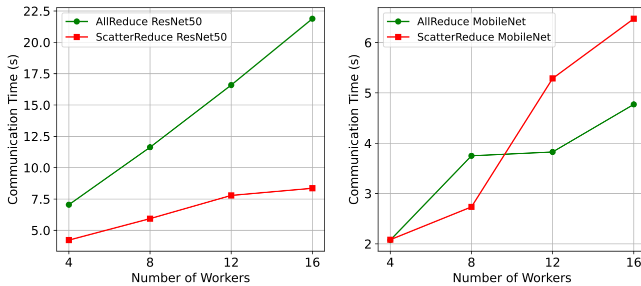


Fig. 7. Comparison of communication times between scatter reduces, and all reduce patterns as a function of the number of workers for 1 training step.

directly within Redis. Through this investigation, we aim to shed light on how our approach can reduce communication overhead and subsequently enhance training performance.

Approach: In our experiment, we first explore the communication overhead gains that can be realized by conducting model updates directly within RedisAI. Following this, we delve into the benefits of calculating gradient averages within Redis. For a comprehensive evaluation that considers the impact of model size on the overhead, we will use two distinct models - MobileNetV3 Small and ResNet18 - and run these models on the MNIST dataset. The insights from these tests will then be compared with the traditional method of iterative fetch-update-store operations, typically used with standard Redis, as outlined in other serverless training ML frameworks.

Results: An in-depth analysis of the experimental results provides persuasive evidence of the significant efficiency improvements facilitated by in-database operations, as depicted in Figures 5 and 6.

Figure 5 concisely illustrates the substantial reduction in time for gradient averaging calculations within the database for both the MobileNetV3 Small and the ResNet-18 models. As the plot underscores, the in-database approach results in a reduction of overall average computation time across various batch sizes. The MobileNetV3 Small model, for example, experiences an impressive decrease in computation time from 135.29 seconds outside the database to 78.52 seconds within, for a batch size of 64. The gradients averaging time continues to escalate as the batch size increases, with the largest batch size of 1024 resulting in a striking 82% improvement, requiring only 5.4 seconds for in-database computations.

A similar pattern is observed with the larger ResNet-18 model, where gradients averaging computations conducted within the database halved the processing time to 37.41 seconds, a stark reduction from the 67.32 seconds required when computed outside the database, for the batch size of 1024. This trend equates to a remarkable 44.43% improvement in processing efficiency, proving that our approach's benefits are applicable even for larger, more demanding models.

Figure 6 delves into the time taken for model updates within and outside the database. It illustrates a decrease in model update times when processed within the database. MobileNetV3 Small model updates reached an 82% reduction, shrinking from 3.49 seconds outside the database to a mere 0.64 seconds within. Likewise, ResNet-18 updates experienced an approximately 83% improvement, with update times reducing from 27.5 seconds to 4.8 seconds.

3) LambdaML ScatterReduce Vs ALLReduce:

Motivation: In LambdaML's study [20], authors explored the impact of communication patterns between scatter reduce and all reduce. Their evaluation was conducted with 10 workers on two models, MobileNet and ResNet, demonstrating that as the model size increases, scatter reduce becomes faster because communication becomes heavier and the single reducer (i.e., aggregator) in AllReduce becomes a bottleneck. However, the number of workers can significantly impact communication, as this number determines how many chunks are communicated over the network in ScatterReduce approach. Varying the number of workers will provide insights into when each framework —scatter reduce or all reduce— performs optimally. **Approach:** We replicated LambdaML's communication patterns experiment by testing MobileNet and ResNet-50 models on the CIFAR dataset, adjusting the worker count from 4 to 16 in 4-worker increments.

Result: As illustrated in Figure 7, the communication time for the ResNet50 model, which has a significant parameter footprint of approximately 89MB, displayed a linear increase with the AllReduce strategy as more workers were introduced. This time escalated from roughly 7.05 seconds with 4 workers to 21.88 seconds at 16 workers, highlighting a scalability bottleneck, likely due to the centralized aggregation process. Conversely, the ScatterReduce strategy, by distributing the aggregation process more effectively, managed to maintain lower communication times, with the peak reaching only about 8.36 seconds with 16 workers. For the smaller MobileNet model, about 12MB in size, AllReduce demonstrated better scalability at higher worker counts, maintaining a lower communication time of 4.77 seconds with 16 workers, compared to ScatterReduce's 6.47 seconds.

4) MLLEss Significant Updates:

Motivation: The Significant Updates Filter introduces an efficient strategy for managing communication while maintaining accuracy in distributed learning systems. Instead of transmitting every minor update to the model parameters across workers, this approach advocates for aggregating these updates locally until they collectively reach a significant level based on a predefined threshold. Once this threshold is exceeded, the accumulated history of insignificant updates is packed into a single transmission, reducing the overall communication burden.

Approach: The MLLess [15] method for significant update calculation was initially designed for lightweight ML models like sparse logistic regression and matrix factorization. We extended MLLess to support deep learning models, enhancing its applicability across various scenarios. This expansion addresses the substantial challenge posed by the vast number of parameters in deep learning models compared to lightweight ML models. To tackle this, we leveraged tensor's norm in our filter implementation to provide a singular value that captures the overall magnitude of the model. In this case, the significant update is computed by accumulation of gradient norm variation across the model parameters over the training

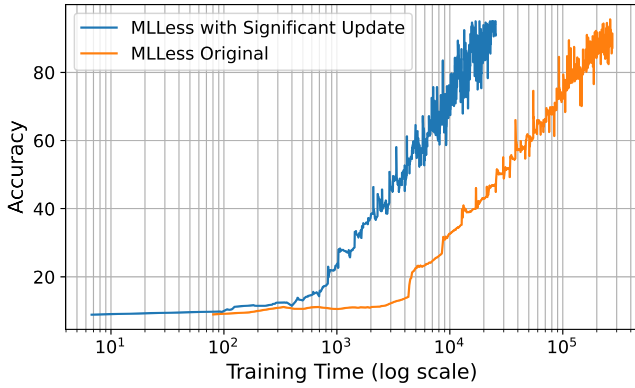


Fig. 8. Significant Update Evaluation on MLLess

steps until the ratio exceeds a predefined threshold.

Formally, the refined, significant update criterion for a deep learning model with parameters Θ. can be represented by the following equation:

$$\sum_{t'}^{t} \left(\frac{\| \sum_{t'}^{t} \nabla\Theta \|}{\|\Theta\|} \right) > \text{threshold}$$

where:

- t denotes the current step of the model and t' is the step number of the last propagation time,
- $\| \cdot \|$ denotes the norm (e.g., the Euclidean norm) of a vector.
- $\sum_{t'}^{t} \nabla\Theta$ represents the accumulated gradients of the parameters from step t' to step t.
- The *threshold* is a predefined value that determines the significance of an update.

We note that higher threshold decreases update frequency but may miss minor updates, while a lower threshold increases frequency and overhead to catch smaller updates.

Results: As illustrated in Figure 8, the implementation of a significance filter within our experimental framework has improved the convergence rates over traditional training methods. Utilizing this filter, convergence was achieved in a significantly reduced time frame of 8667 seconds, in stark contrast to the 113379 seconds necessitated by conventional training approaches. This achieved a 13-fold improvement in the rate of training convergence, significantly reducing the time required to reach optimal model performance.

D. Fault Tolerance and Auto-scaling

Motivation: In the dynamic realm of distributed machine learning (ML) peer-to-peer training, disruptions, such as peer failures, new peers joining, and potential Byzantine behaviors,

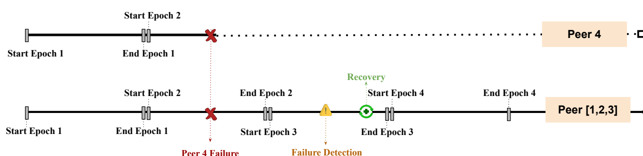


Fig. 9. Recovery Time when a Peer Fails

are inevitable, and if not effectively managed, can impact the reliability and performance of the ML training. This provides the impetus for our experiment, which is meticulously designed to evaluate the robustness, scalability, and fault tolerance of a serverless, peer-to-peer ML training architecture. In this architecture, peers are carefully identified to form a secure training network, and its capacity to handle peer failures, seamlessly integrate new peers, and resist Byzantine attacks will be critically evaluated to ensure long-running distributed ML training sessions can continue to deliver accurate results amidst these disruptions.

Approach: In our experimental approach, we center on the evaluation of three crucial aspects using the Mobilenet V3 Small model trained on the MNIST dataset: *peer failures*, the *addition of new peers*, and *Byzantine attacks*.

We initiate with the simulation of ***peer failures*** where we begin training with four peers, each processing 15 compute gradients. A peer failure is then artificially induced, leading us to measure the time taken by the remaining peers to identify the failure. In response to this failure, these remaining peers incorporate the data of the failed peer into their following epochs, augmenting the compute gradients to 20. This simulation aids in testing the system's resilience and ability to adapt in response to computational loss during peer failures.

To assess the system's scalability, we execute the ***addition of new peers*** scenario where a new peer is introduced into the network. We proceed to calculate the time taken by the existing peers to recognize and integrate this newcomer into the ongoing training process.

Lastly, our approach includes countering potential ***Byzantine attacks*** using robust aggregation algorithms, Zeno and Meamed. Zeno [29] ensures resilience by using a validation set to score and exclude any potentially adversarial local updates. The medians-based approach, or Meamed [28], combats Byzantine attacks by creating a vector that minimizes the overall distance to all local updates, thereby mitigating the influence of adversarial elements. By simulating adversarial scenarios of one malicious peer, such as a sign flipping attack [55] where the malicious peer inverts and amplifies its local gradient, and a noise attack [56] where the malicious peer introduces Gaussian noise to its local updates, we track the training progression to convergence. This comprehensive approach enables us to gauge the architecture's resilience and robustness against adversarial behavior.

results:

1) ***Peer Failure:*** The flow of the experiment are depicted in Figure 9 for a more visual understanding. Initially, we began with four peers, each designed to process 15 batches per epoch. The first epoch proceeded unimpeded, with the total training time recorded at 52.6 seconds.

A simulated peer failure was introduced at the beginning of the second epoch, immediately after a health check had validated the 'failed' peer as operational. This timing extended the detection period. Nonetheless, the remaining peers recognized the failed peer within the ongoing epoch, taking a total of 50.9 seconds to align with the next epoch's heartbeat step.

Post this single-peer-level detection, the remaining peers reached a consensus on the failed peer within 9.66 seconds.

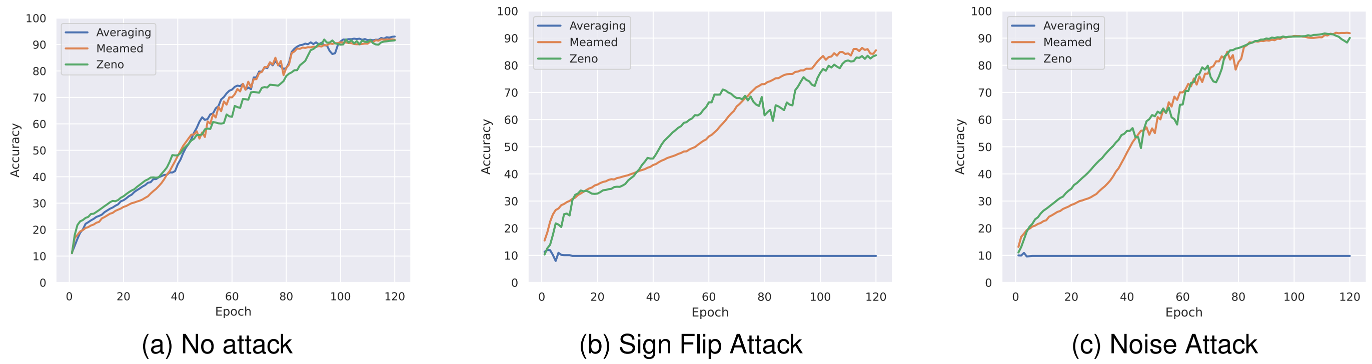


Fig. 10. Evaluation of network accuracy using Averaging, Zeno, and Meamed aggregation methods under conditions: (a) Normal training with no attack, (b) Training under sign flip attack, and (c) Training amidst a noise attack, where random Gaussian noise is added to local updates.

This aggregated the total detection time to 61.56s seconds of the failed peer. we note that, despite the deferred detection of the simulated failure in the third epoch, the training process was not interrupted, thus maintaining the progression towards model convergence. Upon consensus on the peer failure, the recovery process was triggered. This involved creating a new AWS Step Function to redistribute the failed peer's computational workload, which increased the remaining peers' load from 15 to 20 batches per epoch. The complete recovery process, encapsulating the creation, deployment, and database entry of the new AWS Step Function, required an extra 1.94 seconds. After recovery, the remaining peers continued training in the 4th epoch with an adjusted workload to account for the lost peer. By the end of the fourth epoch, the total training time post-recovery was registered as 55,06 seconds.

Our analysis shows a slight increase in total training time from 52.6 to 55.06 seconds, as the system transitions from four to three peers. This can be attributed to the increased number of gradients being averaged due to the redistribution of the failed peer's workload.

*2) **Adding a new peer**:* We found that in the scenario of *adding a new peer*, it took approximately 7.2 seconds for the existing three peers to recognize and integrate the new peer into their list of trusted peers.

*3) **Tolerating Byzantine attacks**:* The adoption of Meamed and Zeno aggregation algorithms introduced a computational overhead, leading to an approximate increase of 8.2 and 5.9 times in computational time, respectively, in contrast to the average aggregation method.

In a scenario without adversarial attacks, all three aggregation methods – Averaging, Zeno, and Meamed – achieved an accuracy above 90% within about 100 epochs. During a sign flip attack, the robust aggregations, Zeno and Meamed, managed to converge to almost 85%, while the normal averaging method did not. In a noise attack scenario, both Zeno and Meamed converged above 90% after nearly 90 epochs, but the normal averaging method remained divergent.

E. Performance Evaluation of Training Accuracy

Motivation: After exploring the training durations and studying the cost implications of various serverless frameworks, it is important to evaluate the accuracy to understand the trade-offs between speed, expense, and model performance. This step ensures a balanced assessment, highlighting frameworks that offer the best combination of efficiency, cost-effectiveness, and high-quality results.

Approach: In our experiment aimed at comparing the accuracy of machine learning training in different serverless environments, we tailored the data processing approach to align with the operational characteristics of each framework. For MLless and SPIRIT, we divided the dataset into 196 batches, with each of the four workers directly processing 49 batches to complete a single epoch. In contrast, for AllReduce and Scatterduce, the dataset was divided based only on the number of workers. Each worker then acted as a dataloader, processing the dataset batch by batch. To detect convergence, we utilized the method of early stopping.

Results: The analysis of serverless machine learning frameworks reveals varied convergence patterns: SPIRT converges the quickest, achieving an accuracy of 83.2% within approximately 61.96 minutes. ScatterReduce, in contrast, begins its convergence process after 1,652.49 minutes, ultimately reaching an accuracy of 82.1%, which suggests a slower and steadier learning trajectory. MLless Significant records a lower accuracy of 83.48%, but it takes significantly longer to converge, approximately 189.68 minutes, indicating potential inefficiencies or data reporting anomalies. Lastly, AllReduce starts converging after 1,367.01 minutes and achieves an accuracy of 85.05%, outperforming ScatterReduce in both speed and accuracy.

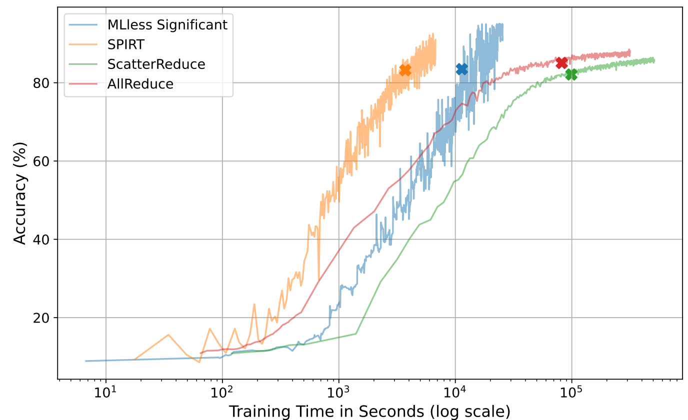


Fig. 11. Comparative Accuracy Evaluation of Serverless Training Frameworks

VII. Discussions

This section provides an analysis of the findings from our comparative study of serverless machine learning (ML) training frameworks. We discuss the implications of these findings for the field of serverless ML training and explore potential avenues for future research.

A. Serverless Frameworks: Training Time

Our comparative analysis of four serverless architectures—SPIRT, ScatterReduce, ALLReduce, and MLLess—reveals varied training times.

MLLess faces substantial delays during synchronization and aggregation, attributed to its queue-based communication. It uses complex tasks, such as workers scaling down based on predictive models and planning when training can be stopped. This suggests a need for optimization in queue management and task coordination. ScatterReduce and ALLReduce show balanced performance but are affected by synchronization overhead as the number of workers increases, indicating scalability limits in their current synchronization methods. SPIRT successfully reduces training times by leveraging its streamlined parallel batch processing to accelerate gradient computations and dataset retrieval. This makes SPIRT particularly suitable for low-latency applications.

B. Serverless Frameworks: Cost Implication

Cost saving is one of the main reasons to use serverless computing, allowing you to pay only for what you execute.

The multi-database architectural design of Spirit, although typically more expensive than other frameworks, achieves cost reduction by minimizing training time and dividing the machine learning training workflow stage into separate serverless functions, allocating resources exclusively to each function. This contrasts with other frameworks that commonly use a single serverless function with continuous memory allocation throughout the training workflow.

Adapting SPIRT to utilize a single database may result in cost savings and architectural simplification; however, this would be accompanied by a decrease in fault tolerance. This trade-off must be carefully considered, especially in applications where reliability is paramount.

C. Serverless Frameworks: Communication Overhead

Each framework has proposed a method to reduce communication overhead. The SPIRT Parallel Batch Processing technique enhances communication by conducting parallel minibatch processing within each worker and then consolidating these updates into a single communication round. Additionally, SPIRT's RedisAI integration boosts communication for serverless machine learning tasks that often interact with databases due to their stateless nature.

LambdaML's ScatterReduce vs. ALLReduce communication patterns exhibit varying performance based on model size and worker count. ScatterReduce excels in handling larger models by efficiently managing heavy communication loads, thus outperforming ALLReduce regardless of the number of workers and avoiding bottlenecks. In contrast, for smaller models with fewer workers, both methods are equally effective. However, as the number of workers increases, ALLReduce gains due to its synchronized updating mechanism.

MLLess's significant updates filter employs a strategy to manage communication overhead between workers. It aggregates insignificant updates locally and only transmits significant updates once they surpass a predefined threshold. This approach effectively reduces communication load and makes the overall training faster.

D. Security and Fault Tolerance in ML Architectures

The fault tolerance capabilities were taken into consideration by the different frameworks, each with varying levels of severity. For example, in the *MLLess* framework, training can continue with existing workers even if one worker goes down, while a supervisor's failure blocks the workers until the supervisor returns, with no recovery options provided. Similar scenarios are seen in the *AllReduce* and *ScatteReduce* frameworks, where the entire system is blocked until the downed worker or master worker returns. Conversely, the *SPIRT* framework allows training to continue with existing workers and involves active detection and redistribution of data among workers when a peer is inactive.

SPIRT has adopted security measures that include cryptographic mechanisms to ensure data integrity, authenticity, and confidentiality. To prevent model deviation that can be triggered by an intruder, SPIRT employs robust aggregation techniques that securely consolidate gradients from multiple workers. While this robust aggregation process may extend the aggregation time, its capacity to safeguard against Byzantine attacks and guarantee model convergence is invaluable.

E. Lessons Learned: Serverless Computing for Training ML

Serverless computing is typically chosen for lightweight, event-triggered functions and parallel processing. It handles high demand by running concurrent functions, enabling dynamic scaling without server infrastructure management. Furthermore, its pay-as-you-go pricing model reduces the complexities of server-side operations. However, serverless computing has inherent limitations, including restrictions on package size, limited execution times, and stateless nature. These constraints necessitate reliance on external databases for saving results and managing communication between functions, making serverless less ideal for complex tasks. To effectively use serverless for complex operations, such as training machine learning, task logic must be divided to meet these functional constraints. Most serverless machine learning frameworks, such as MLLess and LambdaML, operate by running training within these functions and saving the status before timeout to trigger subsequent functions. Our approach with SPIRT, however, involves splitting the training workflow stages into manageable serverless functions. This method allowed us to parallelize gradient computations and integrate machine learning operations directly within the database. As a result, SPIRT emerged as the fastest and most cost-effective framework.

VIII. Conclusion

In this study, we present the Serverless Peer Integrated for Robust Training (SPIRT), a serverless machine learning (ML) architecture, and conduct an extensive evaluation of various serverless distributed ML architectures, assessing their efficiency in training times, cost management, communication overhead, and resilience in fault tolerance and security. The insights derived from this comparative study are invaluable for practitioners and researchers aiming to optimize ML training processes within serverless environments. Our findings delineate clear distinctions among the analyzed architectures, each presenting unique benefits and challenges. Among the architectures assessed, SPIRT architecture was notably superior in reducing training times and communication overhead through parallel batch processing and in-database operations using RedisAI.

In contrast, architectures like AllReduce faced scalability challenges with increasing worker counts, particularly under the load of large model parameters, revealing potential bottlenecks in their centralized aggregation processes. MLLess, while innovative in its approach to minimize communication overhead through a significant updates filter, exhibited longer training times and higher costs due to intensive data interactions. Our cost analysis revealed that despite SPIRT's higher setup costs, its efficient resource management translated into long-term savings. Furthermore, SPIRT demonstrated robust fault tolerance and security features, effectively mitigating risks associated with Byzantine attacks and peer failures.

In our future research, we plan to explore the impact of memory allocations on the performance and cost-efficiency of ML training serverless functions. Our goal is to determine the most cost-effective memory settings that still deliver optimal performance. By systematically adjusting memory allocations and varying batch sizes, we aim to find a balance where the increase in computational speed and training efficiency offsets the cost.

References

[1] B. Yuan, C. R. Wolfe, C. Dun, Y. Tang, A. Kyrillidis, and C. Jermaine, "Distributed learning of fully connected neural networks using independent subnet training," *Proc. VLDB Endow.*, vol. 15, no. 8, p. 1581–1590, apr 2022. [Online]. Available: https://doi.org/10.14778/3529337.3529343

[2] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, and J. S. Rellermeyer, "A survey on distributed machine learning," *Acm computing surveys (csur)*, vol. 53, no. 2, pp. 1–33, 2020.

[3] S. Alqahtani and M. Demirbas, "Performance analysis and comparison of distributed machine learning systems," *arXiv preprint arXiv:1909.02061*, 2019.

[4] T. Sun, D. Li, and B. Wang, "Decentralized federated averaging," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022.

[5] M. Li, L. Zhou, Z. Yang, A. Li, F. Xia, D. G. Andersen, and A. Smola, "Parameter server for distributed machine learning," in *Big learning NIPS workshop*, vol. 6, no. 2, 2013.

[6] R. Šajina, N. Tanković, and I. Ipšić, "Peer-to-peer deep learning with non-iid data," *Expert Systems with Applications*, vol. 214, p. 119159, 2023.

[7] A. Ulanov, A. Simanovsky, and M. Marwah, "Modeling scalability of distributed machine learning," in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 2017, pp. 1249–1254.

[8] X. Liu, D. Gu, Z. Chen, J. Wen, Z. Zhang, Y. Ma, H. Wang, and X. Jin, "Rise of distributed deep learning training in the big model era: From a software engineering perspective," *ACM Transactions on Software Engineering and Methodology*, 2023.

[9] A. Barrak, F. Petrillo, and F. Jaafar, "Serverless on machine learning: A systematic mapping study," *IEEE Access*, vol. 10, pp. 99 337–99 352, 2022.

[10] Y. Yang, L. Zhao, Y. Li, H. Zhang, J. Li, M. Zhao, X. Chen, and K. Li, "Infless: a native serverless system for low-latency, high-throughput inference," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 768–781.

[11] A. Bhattacharjee, Y. Barve, S. Khare, S. Bao, A. Gokhale, and T. Damiano, "Stratum: A serverless framework for the lifecycle management of machine learning-based data analytics tasks," in *2019 USENIX Conference on Operational Machine Learning (OpML 19)*. Santa Clara, CA: USENIX Association, May 2019, pp. 59–61. [Online]. Available: https://www.usenix.org/conference/opml19/presentation/bhattacharjee

[12] "Serverless computing - aws lambda - amazon web services," https://aws.amazon.com/lambda/, (Accessed on 04/20/2023).

[13] "Cloud functions — google cloud," https://cloud.google.com/functions, (Accessed on 01/26/2022).

[14] "Cloud computing services — microsoft azure," https://azure.microsoft.com/en-us/, (Accessed on 01/26/2022).

[15] P. Gimeno Sarroca and M. Sánchez-Artigas, "Mlless: Achieving cost efficiency in serverless machine learning training," *Journal of Parallel and Distributed Computing*, vol. 183, p. 104764, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S074373152300134X

[16] A. Ali, S. Zawad, P. Aditya, I. E. Akkus, R. Chen, and F. Yan, "Smlt: A serverless framework for scalable and adaptive machine learning design and training," *arXiv preprint arXiv:2205.01853*, 2022.

[17] A. Grafberger, M. Chadha, A. Jindal, J. Gu, and M. Gerndt, "Fedless: Secure and scalable federated learning using serverless computing," *arXiv preprint arXiv:2111.03396*, 2021.

[18] A. Barrak, R. Trabelssi, F. Jaafar, and F. Petrillo, "Exploring the impact of serverless computing on peer to peer training machine learning," *International Conference on Cloud Engineering*, 2023.

[19] P. G. Sarroca and M. Sánchez-Artigas, "Mlless: Achieving cost efficiency in serverless machine learning training," *arXiv preprint arXiv:2206.05786*, 2022.

[20] J. Jiang, S. Gan, Y. Liu, F. Wang, G. Alonso, A. Klimovic, A. Singla, W. Wu, and C. Zhang, "Towards demystifying serverless machine learning training," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 857–871.

[21] D. Chahal, M. Mishra, S. C. Palepu, R. K. Singh, and R. Singhal, "Pay-as-you-train: Efficient ways of serverless training," in *2022 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2022, pp. 116–125.

[22] H. Wang, L. Muñoz-González, M. Z. Hameed, D. Eklund, and S. Raza, "Sparsfa: Towards robust and communication-efficient peer-to-peer federated learning," *Computers & Security*, vol. 129, p. 103182, 2023.

[23] M. Xu, Z. Zou, Y. Cheng, Q. Hu, D. Yu, and X. Cheng, "Spdl: A blockchain-enabled secure and privacy-preserving decentralized learning system," *IEEE Transactions on Computers*, 2022.

[24] M. Shayan, C. Fung, C. J. Yoon, and I. Beschastnikh, "Biscotti: A ledger for private and secure peer-to-peer machine learning," *arXiv preprint arXiv:1811.09904*, 2018.

[25] R. Guerraoui, A. Guirguis, J. Plassmann, A. Ragot, and S. Rouault, "Garfield: System support for byzantine machine learning (regular paper)," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2021, pp. 39–51.

[26] C. Fang, Z. Yang, and W. U. Bajwa, "Bridge: Byzantine-resilient decentralized gradient descent," *arXiv preprint arXiv:1908.08098*, 2019.

[27] P. Blanchard, E. M. El Mhamdi, R. Guerraoui, and J. Stainer, "Machine learning with adversaries: Byzantine tolerant gradient descent," *Advances in neural information processing systems*, vol. 30, 2017.

[28] C. Xie, O. Koyejo, and I. Gupta, "Generalized byzantine-tolerant sgd," *arXiv preprint arXiv:1802.10116*, 2018.

[29] C. Xie, S. Koyejo, and I. Gupta, "Zeno: Distributed stochastic gradient descent with suspicion-based fault-tolerance," in *International Conference on Machine Learning*. PMLR, 2019, pp. 6893–6901.

[30] A. Barrak, M. Jaziri, R. Trabelsi, F. Jaafar, and F. Petrillo, "Spirt: A fault-tolerant and reliable peer-to-peer serverless ml training architecture," in *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*. IEEE, 2023, pp. 650–661.

[31] E. Haußmann, "Accelerating i/o bound deep learning on shared storage," 2018.

[32] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica *et al.*, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.
[33] D. Barcelona-Pons, P. Sutra, M. Sánchez-Artigas, G. París, and P. García-López, "Stateful serverless computing with crucial," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 3, pp. 1–38, 2022.
[34] F. Xu, Y. Qin, L. Chen, Z. Zhou, and F. Liu, "λ dnn : Achieving predictable distributed dnn training with serverless architectures," *IEEE Transactions on Computers*, 2021.
[35] M. Sánchez-Artigas and P. G. Sarroca, "Experience paper: Towards enhancing cost efficiency in serverless machine learning training," in *Proceedings of the 22nd International Middleware Conference*, 2021, pp. 210–222.
[36] T. Naumenko and A. Petrenko, "Analysis of problems of storage and processing of data in serverless technologies," *Technology audit and production reserves*, vol. 2, no. 2, p. 58, 2021.
[37] Y. Liu, B. Jiang, T. Guo, Z. Huang, W. Ma, X. Wang, and C. Zhou, "Funcpipe: A pipelined serverless framework for fast and cost-efficient training of deep learning models," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 6, no. 3, pp. 1–30, 2022.
[38] P. G. Sarroca and M. Sánchez-Artigas, "Mlless: Achieving cost efficiency in serverless machine learning training," *arXiv preprint arXiv:2206.05786*, 2022.
[39] "Redisai - a server for machine and deep learning models," https://oss.redis.com/redisai/#quick-links, (Accessed on 06/20/2023).
[40] "Cross-region replication in azure — microsoft learn," https://learn.microsoft.com/en-us/azure/reliability/cross-region-replication-azure, (Accessed on 05/29/2023).
[41] "Configuring a lambda function to access resources in a vpc - aws lambda," https://docs.aws.amazon.com/lambda/latest/dg/configuration-vpc.html, (Accessed on 05/29/2023).
[42] A. Qiao, B. Aragam, B. Zhang, and E. Xing, "Fault tolerance in iterative-convergent machine learning," in *International Conference on Machine Learning*. PMLR, 2019, pp. 5220–5230.
[43] Y. Bouizem, N. Parlavantzas, D. Dib, and C. Morin, "Active-standby for high-availability in faas," in *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, 2020, pp. 31–36.
[44] T. Wink and Z. Nochta, "An approach for peer-to-peer federated learning," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2021, pp. 150–157.
[45] H. Bourreau, E. Guichet, A. Barrak, B. Simon, and F. Jaafar, "On securing the communication in iot infrastructure using elliptic curve cryptography," in *2022 IEEE 22nd International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*. IEEE, 2022, pp. 758–759.
[46] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," in *Concurrency: the works of leslie lamport*, 2019, pp. 203–226.
[47] "Lambda quotas - aws lambda," https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html, (Accessed on 06/24/2023).
[48] "Operating lambda: Performance optimization – part 2 — aws compute blog," https://aws.amazon.com/fr/blogs/compute/operating-lambda-performance-optimization-part-2/, (Accessed on 02/06/2024).
[49] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE signal processing magazine*, vol. 29, no. 6, pp. 141–142, 2012.
[50] A. Krizhevsky and G. Hinton, "Convolutional deep belief networks on cifar-10," *Unpublished manuscript*, vol. 40, no. 7, pp. 1–9, 2010.
[51] B. Koonce and B. Koonce, "Mobilenetv3," *Convolutional Neural Networks with Swift for Tensorflow: Image Recognition and Dataset Categorization*, pp. 125–144, 2021.
[52] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
[53] "Serverless computing – aws lambda pricing – amazon web services," https://aws.amazon.com/lambda/pricing/, (Accessed on 04/11/2024).
[54] "Create estimate: Configure aws lambda," https://calculator.aws/#/createCalculator/Lambda, (Accessed on 04/11/2024).
[55] L. Li, W. Xu, T. Chen, G. B. Giannakis, and Q. Ling, "Rsa: Byzantine-robust stochastic aggregation methods for distributed learning from heterogeneous datasets," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, 2019, pp. 1544–1551.
[56] S. Li, Y. Cheng, W. Wang, Y. Liu, and T. Chen, "Learning to detect malicious clients for robust federated learning," *arXiv preprint arXiv:2002.00211*, 2020.

Amine Barrak is currently a Ph.D. candidate in Software Engineering at University of Quebec at Chicoutimi. His doctoral research focuses on the suitability between serverless computing and machine learning pipelines. Prior to this, he was a Master's student at Polytechnique Montreal, where he studied the occurrence of security vulnerability changes in software code. He was the recipient of the best student paper award in CASCON 2018. He has currently published two in major refereed international conferences. Amine's prior research includes cloud computing, MLOps, application of machine learning techniques, analysis of software repositories, data science, data analytics, and natural language processing.

Ranim Trabelssi recently started her PhD in the domain of machine learning at the University of Quebec at Chicoutimi. Before this, she completed her Master's degree at Polytechnique Tunisia, focusing on machine learning and IoT (Internet of Things). She excelled during her internship at the University of Quebec at Chicoutimi, where she published two papers in international conferences on serverless training for machine learning.

Fabio Petrillo is an associate professor in the Department of Software Engineering Information Technology at École de Technologie Supérieure (Canada). He obtained his Ph.D. degree in Computer Science in 2016 from Federal University of Rio Grande do Sul (Brazil) and he were a postdoctoral fellow at Concordia University (Canada). In his research career, he has worked on Empirical Software Engineering, Software Quality, Debugging, Service-Oriented Architecture, Cloud Computing, and Agile Methods. He has been recognized as a pioneer and an international reference on Software Engineering for Computer Games. I'm the creator of Swarm Debugging, a new collaborative approach to support debugging activities. Dr. Petrillo has published several papers in international conferences and journals, including TSE, EMSE, JSS, IST, IEEE Software, QRS, ICPC, SAC, ICSOC, and VISSOFT. He has served on the program committees of several international conferences including QRS, CHI, SIGCSE, ICPC, VISSOFT, GAS and has reviewed for top international journals such as TSE, TOSEM, JSS, EMSE and IST.

Fehmi Jaafar is currently is an Associate Professor at Quebec University at Chicoutimi and an Affiliate Professor at Laval University and Concordia University. Previously, he was Researcher at the Computer Research Institute of Montreal, an Adjunct Professor at Concordia University of Edmonton, and a post-doctoral research fellow at Queen's University and Polytechnique Montreal. Dr. Fehmi Jaafar received his PhD from the Department of Computer Science at Montreal University, Canada. He is interested in the Internet of Things security, and in the application of machine learning techniques in cybersecurity. His researches have been published in top venues in computer sciences, including the Journal of Empirical Software Engineering (EMSE) and the Journal of Software: Evolution and Process (JSEP). He established externally funded research programs in collaboration with Defence Canada, Safety Canada, NSERC, MITACS, etc.