UQAC
Université du Québec
à Chicoutimi

SERVERLESS ARCHITECTURES FOR SCALABLE PEER-TO-PEER MACHINE LEARNING TRAINING

BY AMINE BARRAK

THESIS PRESENTED TO L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI IN PARTIAL FULFILLMENT OF THE REQUIERMENTS FOR THE DEGREE OF PHILOSOPHIÆ DOCTOR (PH.D.) IN THE SUBJECT OF INFORMATIQUE

QUÉBEC, CANADA

ABSTRACT

Serverless computing has revolutionized cloud-computing paradigms by allowing developers to focus on application functionality without managing the underlying infrastructure. It achieves this through automatic resource allocation and a pay-as-you-go pricing model, which optimizes costs and enhances scalability. However, the application of serverless architectures to machine learning (ML) training introduces significant challenges due to the stateless and ephemeral nature of serverless functions, which complicates tasks requiring persistent state and extensive communication.

Different ML training architectures, such as centralized and distributed systems, present unique challenges in a serverless environment. Centralized training, which relies on a single powerful machine or cluster, introduces a single point of failure, making it vulnerable to disruptions and reducing reliability. Distributed training systems, including peer-to-peer networks, require significant coordination and data transfer between nodes, which is challenging to manage with serverless functions due to the need for continuous state synchronization and the overhead of communication. Despite these challenges, the dynamic resource allocation, scalability, and cost benefits of serverless computing present significant opportunities for optimizing ML training. These benefits are particularly valuable in application domains such as real-time data processing, IoT, and edge computing, where demand can be highly variable and unpredictable.

This dissertation investigates the hypothesis that serverless computing can significantly enhance ML training efficiency through advanced techniques that address these inherent challenges. The research is structured around four sub-hypotheses: (1) Serverless functions can accelerate parallel gradient computation during ML training, although they introduce challenges related to state management, synchronization, and communication overhead; (2) In-database ML operations, such as gradient aggregation and model updates, can effectively reduce communication overhead in serverless distributed training environments by minimizing data transfer, with tools like RedisAI adapted for these operations; (3) Implementing key mechanisms for fault tolerance and secure communication protocols improves the reliability and security of serverless ML training; and (4) The proposed SPIRT architecture, a fully serverless training workflow, offers superior training speed, performance, and fault tolerance compared to existing serverless ML training frameworks.

To explore these hypotheses, a comprehensive review of over 150 studies was conducted to understand the integration of serverless computing within ML pipelines, starting from data preprocessing and model training to deployment and inference. The findings reveal an increasing adoption of serverless functions for various ML tasks, particularly in the training phase. Additionally, serverless architectures offer the ability to scale resources which making it well-suited for parallel gradient computations, significantly accelerating the ML training pro-

cess. However, the stateless nature of serverless computing necessitates advanced techniques for efficient resource management, state synchronization, and fault tolerance.

Addressing communication overhead, this research proposes in-database operations to minimize data transfer latencies. Due to the stateless nature of serverless functions, relying on a database is a necessity. By performing gradient averaging and model updates within the database, specifically through our modifications to RedisAI to use stored gradients, we significantly improve training efficiency.

Furthermore, several key mechanisms were implemented to ensure fault tolerance and secure communication protocols in serverless ML training. Firstly, secure communication between workers was ensured, particularly when adding new workers to the training network, using encryption protocols to protect data integrity and confidentiality. A heartbeat mechanism was employed to monitor the status of each worker, allowing for prompt detection and response to any failures. To address fault tolerance, a mechanism was developed to handle worker failures; if a worker goes down, its assigned dataset is redistributed among the remaining workers, ensuring continuous training over all dataset. Additionally, to protect against Byzantine attacks, a method to exclude gradient outliers was incorporated. These implementations collectively enhance the reliability and security of the serverless ML training process.

The SPIRT architecture, inspired by peer-to-peer mechanisms, was designed to optimize serverless ML training by orchestrating training tasks across multiple serverless functions and peers. We proposed two dimensions of scalability: increasing the number of serverless functions to compute gradients within a peer and adding more peers. This architecture avoids single points of failure and optimizes resource utilization, ensuring scalability and resilience. Comparative evaluations with state-of-the-art frameworks demonstrate that SPIRT achieves superior training time efficiency, cost-effectiveness, and fault tolerance.

In conclusion, this research advances the field of serverless computing for machine learning by addressing critical challenges and proposing innovative solutions. The findings highlight the potential of serverless architectures to optimize ML training processes, providing practical frameworks for real-world applications and setting the stage for future advancements in serverless ML training.

RÉSUMÉ

L'informatique sans serveur a révolutionné les paradigmes de l'informatique en nuage en permettant aux développeurs de se concentrer sur la fonctionnalité des applications sans avoir à gérer l'infrastructure sous-jacente. Elle y parvient grâce à l'allocation automatique des ressources et à un modèle de tarification à l'utilisation, qui optimise les coûts et améliore la scalabilité. Cependant, l'application des architectures sans serveur à l'apprentissage automatique (ML) pose des défis importants en raison de la nature sans état et éphémère des fonctions sans serveur, ce qui complique les tâches nécessitant un état persistant et une communication intensive.

Différentes architectures de formation ML, telles que les systèmes centralisés et distribués, présentent des défis uniques dans un environnement sans serveur. La formation centralisée, qui repose sur une machine puissante ou un cluster unique, introduit un point de défaillance unique, la rendant vulnérable aux perturbations et réduisant la fiabilité. Les systèmes de formation distribués, y compris les réseaux peer-to-peer, nécessitent une coordination et un transfert de données importants entre les nœuds, ce qui est difficile à gérer avec des fonctions sans serveur en raison de la nécessité d'une synchronisation continue de l'état et de la surcharge de communication. Malgré ces défis, l'allocation dynamique des ressources, la scalabilité et les avantages en termes de coûts de l'informatique sans serveur offrent des opportunités significatives pour optimiser la formation ML. Ces avantages sont particulièrement précieux dans les domaines d'application tels que le traitement des données en temps réel, l'IoT et l'informatique de périphérie, où la demande peut être très variable et imprévisible.

Cette thèse examine l'hypothèse selon laquelle l'informatique sans serveur peut améliorer de manière significative l'efficacité de la formation ML grâce à des techniques avancées qui répondent à ces défis inhérents. La recherche est structurée autour de quatre sous-hypothèses : (1) Les fonctions sans serveur peuvent accélérer le calcul parallèle des gradients pendant la formation ML, bien qu'elles introduisent des défis liés à la gestion de l'état, à la synchronisation et à la surcharge de communication ; (2) Les opérations ML en base de données, telles que l'agrégation des gradients et la mise à jour du modèle, peuvent réduire efficacement la surcharge de communication dans les environnements de formation distribués sans serveur en minimisant les transferts de données, avec des outils comme RedisAI adaptés à ces opérations ; (3) La mise en œuvre de mécanismes clés pour la tolérance aux pannes et les protocoles de communication sécurisés améliore la fiabilité et la sécurité de la formation ML sans serveur ; et (4) L'architecture SPIRT proposée, un flux de travail de formation entièrement sans serveur, offre une vitesse de formation, des performances et une tolérance aux pannes supérieures par rapport aux cadres de formation ML sans serveur existants.

Pour explorer ces hypothèses, une revue complète de plus de 150 études a été réalisée pour comprendre l'intégration de l'informatique sans serveur dans les pipelines ML, allant

du prétraitement des données et la formation du modèle à son déploiement et son inférence. Les résultats révèlent une adoption croissante des fonctions sans serveur pour diverses tâches ML, particulièrement dans la phase de formation. De plus, la capacité des architectures sans serveur à évoluer dynamiquement s'est révélée particulièrement adaptée aux calculs parallèles des gradients, accélérant significativement le processus de formation ML. Cependant, la nature sans état de l'informatique sans serveur nécessite des techniques avancées pour une gestion efficace des ressources, une synchronisation de l'état et une tolérance aux pannes.

Pour aborder la surcharge de communication, cette recherche propose des opérations en base de données pour minimiser les latences de transfert de données. En raison de la nature sans état des fonctions sans serveur, il est nécessaire de s'appuyer sur une base de données. En effectuant la moyenne des gradients et les mises à jour du modèle dans la base de données, spécifiquement grâce à nos modifications de RedisAI pour utiliser les gradients stockés, nous améliorons significativement l'efficacité de la formation.

En outre, plusieurs mécanismes clés ont été mis en œuvre pour assurer la tolérance aux pannes et les protocoles de communication sécurisés dans la formation ML sans serveur. Tout d'abord, la communication sécurisée entre les travailleurs a été assurée, en particulier lors de l'ajout de nouveaux travailleurs au réseau de formation, en utilisant des protocoles de cryptage pour protéger l'intégrité et la confidentialité des données. Un mécanisme de battement de cœur a été employé pour surveiller le statut de chaque travailleur, permettant une détection et une réponse rapides à toute défaillance. Pour répondre à la tolérance aux pannes, un mécanisme a été développé pour gérer les défaillances des travailleurs ; si un travailleur tombe en panne, son jeu de données assigné est redistribué parmi les travailleurs restants, garantissant une formation continue sur l'ensemble du jeu de données. De plus, pour se protéger contre les attaques byzantines, une méthode pour exclure les outliers des gradients a été incorporée. Ces mises en œuvre améliorent collectivement la fiabilité et la sécurité du processus de formation ML sans serveur.

L'architecture SPIRT, inspirée des mécanismes peer-to-peer, a été conçue pour optimiser la formation ML sans serveur en orchestrant les tâches de formation entre plusieurs fonctions sans serveur et pairs. Nous avons proposé deux dimensions de scalabilité : augmenter le nombre de fonctions sans serveur pour calculer les gradients au sein d'un pair et ajouter plus de pairs. Cette architecture évite les points de défaillance uniques et optimise l'utilisation des ressources, garantissant scalabilité et résilience. Des évaluations comparatives avec des cadres de pointe démontrent que SPIRT atteint une efficacité de temps de formation, un rapport coût-efficacité et une tolérance aux pannes supérieurs.

En conclusion, cette recherche fait progresser le domaine de l'informatique sans serveur pour l'apprentissage automatique en abordant des défis critiques et en proposant des solutions innovantes. Les résultats mettent en évidence le potentiel des architectures sans serveur pour optimiser les processus de formation ML, fournissant des cadres pratiques pour des

applications réelles et préparant le terrain pour de futures avancées dans la formation ML sans serveur.

TABLE OF CONTENTS

ABSTRACT ii

RÉSUMÉ iv

LIST OF TABLES xiii

LIST OF FIGURES xv

LIST OF ABBREVIATIONS xviii

ACKNOWLEDGEMENTS xix

PREFACE xx

CHAPTER I – INTRODUCTION 1

1.1 THESIS HYPOTHESIS 4

1.2 SUMMARY OF THESIS CONTRIBUTIONS 7

1.2.1 CONTRIBUTIONS OF CHAPTER 3 (SUB-HYPOTHESIS 1) 7

1.2.2 CONTRIBUTIONS OF CHAPTER 4 (SUB-HYPOTHESIS 1) 8

1.2.3 CONTRIBUTIONS OF CHAPTER 5 (SUB-HYPOTHESIS 2) 10

1.2.4 CONTRIBUTIONS OF CHAPTER 6 (SUB-HYPOTHESIS 3) 10

1.2.5 CONTRIBUTIONS OF CHAPTER 7 (SUB-HYPOTHESIS 4) 11

1.2.6 CONTRIBUTIONS OF CHAPTER 8 (SUB-HYPOTHESIS 4) 12

1.3 ORGANIZATION OF THE THESIS 12

CHAPTER II – LITERATURE REVIEW 14

2.1 CHAPTER OVERVIEW 14

2.2 DISTRIBUTED MACHINE LEARNING: AN OVERVIEW 16

2.3 SERVERLESS COMPUTING IN THE CONTEXT OF DISTRIBUTED ML . 18

2.4 COMMUNICATION ARCHITECTURES IN DISTRIBUTED ML 28

2.4.1 PEER-TO-PEER (P2P) SYSTEMS 28

2.4.2 PARAMETER SERVER SYSTEMS 30

2.4.3 COMMUNICATION AND MODEL SYNCHRONIZATION IN SERVERLESS DISTRIBUTED ML 31
2.5 FAULT TOLERANCE IN DISTRIBUTED MACHINE LEARNING 33
2.5.1 GENERAL FAULT TOLERANCE TECHNIQUES IN DISTRIBUTED ML 33
2.5.2 FAULT TOLERANCE IN DISTRIBUTED TRAINING OF ML MODELS 34
2.5.3 FAULT TOLERANCE IN CLOUD AND SERVERLESS COMPUTING 34
2.6 APPROACHES TO TOLERATING BYZANTINE FAULTS IN ML 35
2.7 EVALUATION OF DISTRIBUTED ARCHITECTURES 37
2.8 CHAPTER SUMMARY 40
CHAPTER III – SERVERLESS ON MACHINE LEARNING: A SYSTEMATIC MAPPING STUDY 42
3.1 CHAPTER OVERVIEW 42
3.2 STUDY DESIGN 44
3.2.1 RESEARCH QUESTIONS 44
3.2.2 DOMAIN EXPLORATION 45
3.2.3 SEARCH AND SELECTION PROCESS 47
3.3 RESULTS 48
3.3.1 WHAT ARE THE **PUBLICATION TRENDS** OF RESEARCH STUDIES ABOUT SERVERLESS ON MACHINE LEARNING? 50
3.3.2 WHAT IS THE **FOCUS OF RESEARCH** OF APPLIED MACHINE LEARNING ON SERVERLESS COMPUTING ? 52
3.3.3 WHAT ARE THE **POTENTIAL CHALLENGES OF ADOPTING** MACHINE LEARNING ON SERVERLESS COMPUTING? 59
3.4 THREATS TO VALIDITY 71
3.5 CHAPTER SUMMARY 73
CHAPTER IV – EXPLORING THE IMPACT OF SERVERLESS COMPUTING ON PEER TO PEER TRAINING MACHINE LEARNING 74
4.1 CHAPTER OVERVIEW 74
4.2 METHODOLOGY AND SYSTEM DESIGN 77

4.2.1 DESIGN ARCHITECTURE OF PEER TO PEER TRAINING MACHINE LEARNING BASED ON SERVERLESS COMPUTING 78
4.2.2 PEER TO PEER TRAINING MACHINE LEARNING 80
4.2.3 SERVERLESS COMPUTING TO REDUCE A PEER OVERLOAD COMPUTING 86
4.3 EXPERIMENTAL SETUP 86
4.3.1 DATASETS 86
4.3.2 MODEL ARCHITECTURES AND HYPERPARAMETERS 87
4.3.3 EC2 INSTANCES CONFIGURATION FOR PEERS 87
4.3.4 SERVERLESS CLIENT FUNCTIONS CONFIGURATION 88
4.4 EXPERIMENTAL RESULTS 90
4.4.1 IDENTIFY TASKS NEEDING EXPENSIVE COMPUTATIONAL LEVEL 91
4.4.2 EVALUATION OF SERVERLESS INFRASTRUCTURE FOR GRADIENT COMPUTING 92
4.4.3 PEER TO PEER TRAINING AND COMMUNICATION BARRIER SYNCHRONISATION 97
4.5 DISCUSSION 98
4.5.1 BENEFITS AND CHALLENGES OF SERVERLESS INFRASTRUCTURE 98
4.5.2 IMPACT OF MODEL ARCHITECTURE AND DATASET CHOICES . 99
4.6 CHAPTER SUMMARY 100
CHAPTER V – REDUCING COMMUNICATION OVERHEAD IN SERVERLESS DISTRIBUTED TRAINING 101
5.1 CHAPTER OVERVIEW 101
5.2 COMPRESSION AND COMMUNICATION OVERHEAD 103
5.2.1 COMPUTATION AND COMMUNICATION OVER WORKERS 103
5.2.2 COMPRESSION / DECOMPRESSION 104
5.2.3 GRADIENT COMPRESSION FOR COMMUNICATION IMPROVEMENT 105
5.3 MACHINE LEARNING OPERATIONS WITHIN DATABASE 105

5.3.1 MOTIVATION 105
5.3.2 GRADIENTS AVERAGE WITHIN REDIS 106
5.3.3 MODEL UPDATES WITH REDISAI 108
5.4 CHAPTER SUMMARY 111
CHAPTER VI – SECURITY & FAULT TOLERANCE IN SERVERLESS DISTRIBUTED TRAINING 112
6.1 CHAPTER OVERVIEW 112
6.2 SECURING COMMUNICATION AND INTEGRATION 113
6.2.1 PEERS INITIALIZATION AND AUTHENTICATION 114
6.2.2 ADDING AND AUTHENTICATING NEW PEERS 116
6.2.3 PRIVATE KEYS MANAGEMENT ON CLOUD 117
6.3 SECURE MACHINE LEARNING TRAINING FROM BYZANTINE ATTACKS 118
6.3.1 DETERMINING THE NUMBER OF BYZANTINE PEERS 119
6.3.2 APPLYING ROBUST AGGREGATION ALGORITHMS 121
6.4 FAULT-TOLERANCE AGAINST PEERS FAILURE 127
6.4.1 FAILURE SIMULATION APPROACH 129
6.4.2 EXPERIMENTAL RESULTS 130
6.5 CHAPTER SUMMARY 131
CHAPTER VII – SPIRT: A FAULT-TOLERANT AND RELIABLE PEER-TO-PEER SERVERLESS ML TRAINING ARCHITECTURE 133
7.1 CHAPTER OVERVIEW 133
7.2 DESIGN ARCHITECTURE OF LOGICAL PEER TO PEER TRAINING ML 137
7.2.1 COMPREHENSIVE OVERVIEW OF THE PROPOSED ARCHITECTURE 137
7.2.2 DEEP DIVE INTO CORE ARCHITECTURAL COMPONENTS 138
7.2.3 OPERATIONAL DYNAMICS OF THE PROPOSED ARCHITECTURE 141
7.3 EXPERIMENTAL SETUP 145
7.3.1 DATASETS 146
7.3.2 MODEL ARCHITECTURES AND HYPERPARAMETERS 146

7.3.3 REDIS AND REDISAI CONFIGURATION 146
7.3.4 AWS LAMBDA CONFIGURATION 147
7.4 EFFICIENCY AND PERFORMANCE IN SERVERLESS P2P ARCHITECTURE 147
7.4.1 MOTIVATION: 147
7.4.2 APPROACH: 148
7.4.3 RESULTS: 148
7.5 DISCUSSIONS 150
7.5.1 SERVERLESS P2P FOR ML TRAINING 151
7.5.2 SECURITY, RELIABILITY, AND FAULT TOLERANCE IN MACHINE LEARNING ARCHITECTURE 151
7.6 CHAPTER SUMMARY 152
CHAPTER VIII – ADVANCING SERVERLESS ML TRAINING ARCHITECTURES VIA COMPARATIVE APPROACH 155
8.1 CHAPTER OVERVIEW 155
8.2 COMPARATIVE ANALYSIS OF SERVERLESS ML FRAMEWORKS . . . 158
8.2.1 SERVERLESS ML TRAINING WORKFLOW 159
8.2.2 FRAMEWORK-SPECIFIC COMMUNICATION MECHANISMS . . . 159
8.2.3 COMPARATIVE REVIEW OF SERVERLESS ML FRAMEWORKS . . 166
8.3 EVALUATION 169
8.3.1 EXPERIMENTAL SETUP 169
8.3.2 AWS LAMBDA CONFIGURATION 171
8.3.3 DATASET DIVISION FOR WORKERS AND BATCHES 171
8.3.4 SERVERLESS FRAMEWORKS REPLICATIONS 171
8.4 RESULTS 172
8.4.1 SERVERLESS FRAMEWORKS TRAINING TIME 172
8.4.2 SERVERLESS FRAMEWORKS COST ANALYSIS 176
8.4.3 COMMUNICATION OVERHEAD REDUCTION 182

8.4.4 PERFORMANCE EVALUATION OF TRAINING ACCURACY 188
8.5 DISCUSSIONS 190
8.5.1 SERVERLESS FRAMEWORKS: TRAINING TIME 190
8.5.2 SERVERLESS FRAMEWORKS: COST IMPLICATION 190
8.5.3 SERVERLESS FRAMEWORKS: COMMUNICATION OVERHEAD . . 191
8.5.4 SECURITY AND FAULT TOLERANCE IN ML ARCHITECTURES . . 192
8.5.5 LESSONS LEARNED: SERVERLESS COMPUTING FOR TRAINING ML 192
8.6 CHAPTER SUMMARY 193
CHAPTER IX – CONCLUSION AND FUTURE WORK 195
9.1 SUB-HYPOTHESIS ONE (CHAPTERS 3 AND 4) 196
9.2 SUB-HYPOTHESIS TWO (CHAPTER 5) 197
9.3 SUB-HYPOTHESIS THREE (CHAPTER 6) 198
9.4 SUB-HYPOTHESIS FOUR (CHAPTERS 7 AND 8) 199
9.5 FUTURE WORK 200
REFERENCES 203

LIST OF TABLES

TABLE 3.1 : Summary of studies included in the systematic mapping 49

TABLE 3.2 : DISTRIBUTION OF PUBLISHED PAPERS BY VENUE TYPE 51

TABLE 3.3 : JOURNALS' ACRONYM-NAME MAPPING 52

TABLE 3.4 : CONFERENCES-VENUES MAPPING 53

TABLE 3.5 : DISTRIBUTION OF STUDIES BY MACHINE LEARNING PIPELINE STAGE 54

TABLE 3.6 : DISTRIBUTION OF STUDIES BY SERVERLESS PLATFORM USAGE 55

TABLE 3.7 : DISTRIBUTION OF STUDIES BY CHALLENGE & ISSUE 57

TABLE 3.8 : DISTRIBUTION OF STUDIES BY USED MACHINE LEARNING FRAMEWORK 58

TABLE 3.9 : DISTRIBUTION OF STUDIES BY USED MACHINE LEARNING MODEL TYPE 60

TABLE 3.10 : COMPARATIVE ANALYSIS OF LEADING FUNCTION-AS-A-SERVICE (FAAS) PROVIDERS 61

TABLE 4.1 : AWS LAMBDA MEMORY PRICING PER SECOND [1] 89

TABLE 4.2 : EVALUATING RESOURCE USAGE IN DISTRIBUTED PEER-TO-PEER TRAINING WITH FOUR WORKERS AND 30 BATCHES 92

TABLE 4.3 : TIME AND COST EVALUATION OF COMPUTE GRADIENTS IN PEER-TO-PEER TRAINING WITH SERVERLESS; MODEL TRAINED ON VGG11, MNIST DATASET, AND FOUR PEERS 96

TABLE 4.4 : TIME AND COST EVALUATION OF COMPUTE GRADIENTS IN PEER-TO-PEER TRAINING WITHOUT SERVERLESS; MODEL TRAINED ON VGG11, MNIST DATASET, AND FOUR PEERS 96

TABLE 6.1 : COMPUTATIONAL OVERHEAD OF AGGREGATION ALGORITHMS 125

TABLE 7.1 : TRAINING TIME PER EPOCH ACROSS DIFFERENT BATCH SIZES AND PEER COUNTS FOR MOBILENET V3 SMALL AND DENSENET121 MODELS 150

TABLE 8.1 : Comparative Analysis of Serverless Training Framework Architectures: An Overview of Key Training Computational Stages 160

TABLE 8.2 : COMPARATIVE ANALYSIS OF SERVERLESS MACHINE LEARNING FRAMEWORKS: GRADIENTS STORAGE 166

TABLE 8.3 : COMPARATIVE ANALYSIS OF SERVERLESS MACHINE LEARNING FRAMEWORKS: COMMUNICATION CHANNELS 167

TABLE 8.4 : COMPARATIVE ANALYSIS OF SERVERLESS MACHINE LEARNING FRAMEWORKS: COMMUNICATION OVERHEAD REDUCTION 167

TABLE 8.5 : COMPARATIVE ANALYSIS OF SERVERLESS MACHINE LEARNING FRAMEWORKS: SYNCHRONIZATION BARRIER 167

TABLE 8.6 : COMPARATIVE ANALYSIS OF SERVERLESS MACHINE LEARNING FRAMEWORKS: BATCH PROCESSING 168

TABLE 8.7 : COMPARATIVE ANALYSIS OF SERVERLESS MACHINE LEARNING FRAMEWORKS: FAULT TOLERANCE 168

TABLE 8.8 : COMPARATIVE ANALYSIS OF SERVERLESS MACHINE LEARNING FRAMEWORKS: AUTO-SCALING 169

TABLE 8.9 : COMPARATIVE ANALYSIS OF SERVERLESS MACHINE LEARNING FRAMEWORKS: SECURITY MEASURES 169

TABLE 8.10 : 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 179

TABLE 8.11 : 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 256 180

TABLE 8.12 : COMPARATIVE ANALYSIS OF TIME AND COST WITH SEQUENTIAL VS PARALLEL BATCH PROCESSING 183

LIST OF FIGURES

FIGURE 1.1 – OVERVIEW OF THE THESIS METHODOLOGY 9

FIGURE 3.1 – PEER-REVIEWED SELECTION PROCESS 45

FIGURE 3.2 – SYSTEMATIC LITERATURE MAPPING RESEARCH FOCUS 46

FIGURE 3.3 – PUBLISHED PAPERS PER YEAR 51

FIGURE 4.1 – Overview of the proposed Peer To Peer training based on Serverless computing 78

FIGURE 4.2 – SYNCHRONOUS(LEFT) AND ASYNCHRONOUS(RIGH) COMMUNICATION 84

FIGURE 4.3 – Parallel gradient computation orchestrated by AWS Step Function 91

FIGURE 4.4 – Comparison of Processing Training Time on Gradient Computation for Different Numbers of Peers and Batch Sizes in Peer-to-Peer Training with and Without Serverless Architecture 94

FIGURE 4.5 – SYNCHRONOUS VS ASYNCHRONOUS PEER TO PEER TRAINING OF MOBILENET V3 SMALL 98

FIGURE 5.1 – Comparison of Gradients Computation and Communication Time per Number of Peers for VGG11 and MobileNet V3 Small with a Batch Size of 1024. (a) Displays the Results for VGG11, while (b) Shows the Results for MobileNet V3 Small.. 104

FIGURE 5.2 – COMPRESSION ALGORITHM IMPACT ON TIME COMMUNICATION (SEND AND RECEIVE GRADIENTS) 106

FIGURE 5.3 – GRADIENT AVERAGING COMPARISON 107

FIGURE 5.4 – Time taken for calculating gradient averages within and outside the database 108

FIGURE 5.5 – MODEL UPDATE COMPARISON 109

FIGURE 5.6 – TIME TAKEN FOR MODEL UPDATE WITHIN AND OUTSIDE THE DATABASE. 110

FIGURE 6.1 – SEQUENCE DIAGRAM ILLUSTRATING THE INITIALIZATION PROCESS OF PEERS 115

FIGURE 6.2 – SEQUENCE DIAGRAM OF NEW PEER INTEGRATION INTO TRAINING NETWORK 117

FIGURE 6.3 – ILLUSTRATION OF THE APPLICATION OF COSINE SIMILARITY BETWEEN TWO VECTORS A AND B. 119

FIGURE 6.4 – COSINE SIMILARITY APPLICATION TO DETERMINE THE NUMBER OF MALICIOUS PEERS. 120

FIGURE 6.5 – SIMPLIFIED EXPLANATION OF VALIDATION DATA LOADING PROCESS FROM S3. 122

FIGURE 6.6 – Evaluation of network accuracy using Averaging, Zeno, and Meamed aggregation methods under various conditions 126

FIGURE 6.7 – SIMPLIFIED ILLUSTRATION OF DATA REDISTRIBUTION AFTER A PEER FAILURE. 128

FIGURE 6.8 – Heartbeat monitoring mechanism 129

FIGURE 6.9 – RECOVERY TIME WHEN A PEER FAILS 130

FIGURE 7.1 – Overview of the proposed Peer To Peer training based on Serverless computing 136

FIGURE 7.3 – COMPARISON OF AVERAGE AGGREGATION TIMES FOR MOBILENET V3 SMALL AND DENSNET121 MODELS AGAINST # OF PEER 149

FIGURE 7.2 – Comparison of Compute Gradient Time and Average Gradients Across Varying Batch Sizes for Different Number of Peers 154

FIGURE 8.1 – SPIRT ARCHITECTURE WORKFLOW. 161

FIGURE 8.2 – MLLESS ARCHITECTURE WORKFLOW 162

FIGURE 8.3 – SCATTERREDUCE-LAMBDAML ARCHITECTURE WORKFLOW .163

FIGURE 8.4 – ALLREDUCE-LAMBDAML ARCHITECTURE WORKFLOW 165

FIGURE 8.5 – TRAINING TIME FOR ONE EPOCH ACROSS SERVERLESS TRAINING FRAMEWORKS, DEPICTED ON A LOGARITHMIC SCALE. 173

FIGURE 8.6 – TRAINING TIME EVOLUTION WITH INCREASING WORKERS: THIS FIGURE ILLUSTRATES HOW THE TRAINING TIME FOR DIFFERENT STAGES EVOLVES AS THE NUMBER OF WORKERS INCREASES FROM 4 TO 16, ACROSS FOUR SERVERLESS ARCHITECTURES: SPIRT, SCATTERREDUCE, ALLREDUCE AND MLLESS. 175

FIGURE 8.7 – MEMORY ALLOCATION IN SERVERLESS COMPUTING FOR TRAINING WORKFLOWS: A COMPARISON BETWEEN SPIRT AND OTHER SERVERLESS TRAINING FRAMEWORKS. 177

FIGURE 8.8 – COMPARISON OF COMMUNICATION TIMES BETWEEN SCATTER REDUCES, AND ALL REDUCE PATTERNS AS A FUNCTION OF THE NUMBER OF WORKERS FOR 1 TRAINING STEP. 186

FIGURE 8.9 – SIGNIFICANT UPDATE EVALUATION ON MLLESS. 188

FIGURE 8.10 – COMPARATIVE ACCURACY EVALUATION OF SERVERLESS TRAINING FRAMEWORKS 189

LIST OF ABBREVIATIONS

ML	Machine Learning
AWS	Amazon Web Services
P2P	Peer-to-Peer
DB	Database
CI/CD	Continuous Integration/Continuous Deployment
IaC	Infrastructure as Code
S3	Simple Storage Service
RSA	Rivest-Shamir-Adleman
SQS	Simple Queue Service
FaaS	Function-as-a-Service
IPDPS	International Parallel & Distributed Processing Symposium
QRS	Quality, Reliability, and Security
IC2E	International Conference on Cloud Engineering
AI	Artificial Intelligence
TPDS	Transactions on Parallel and Distributed Systems
CPU	Central Processing Unit
GPU	Graphics Processing Unit
IoT	Internet of Things
NLP	Natural Language Processing
ECC	Elliptic Curve Cryptography
API	Application Programming Interface
OS	Operating System
VCS	Version Control System

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my supervisor, **Professor Fehmi Jaafar**, for his exceptional mentorship, unwavering support, and for fostering my confidence throughout my research journey. His guidance and professionalism have been invaluable in the completion of this work.

I am also immensely grateful to my co-supervisor, **Professor Fabio Petrillo**, for his continuous assistance, insightful feedback, and unwavering support. His encouragement and belief in my abilities have greatly contributed to the improvement and success of this thesis.

A heartfelt thank you goes to my parents, **Abderrazak Barrak** and **Chadlia Essid**, for their endless prayers and unwavering belief in my success. Your love and sacrifices have been the foundation of my achievements.

To my beloved wife, **Emna Ksontini**, thank you for your constant moral support and encouragement. Your love, faith in me, and persistent motivation have been a driving force pushing me towards success throughout this journey.

To my dear brother and sisters, thank you for your support and encouragement.

Lastly, I extend my thanks to all my friends and colleagues for the stimulating discussions, shared ideas, and joyous moments. Your companionship has enriched my academic experience.

May God bless you all.

PREFACE

The results of this work have been published or submitted for review to journals, conferences, and workshops. The list of publications is organized by type (journal articles, conference papers, posters, and shorter papers) and ordered by publication year. My contributions to each publication are also detailed.

JOURNAL ARTICLES

1. **Barrak, Amine**, Ranim Trabelsi, Fehmi Jaafar, and Fabio Petrillo. "Advancing Serverless ML Training Architectures via Comparative Approach" IEEE Transactions on Parallel and Distributed Systems. IEEE, TPDS 2024 **(Under Review)** [2].

2. **Barrak, Amine**, Fabio Petrillo, and Fehmi Jaafar. "Serverless on machine learning: A systematic mapping study." IEEE Access (2022) [3].

CONFERENCE PAPERS

1. **Barrak, Amine**, Mayssa Jaziri, Ranim Trabelsi, Fehmi Jaafar, and Fabio 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 [4].

2. **Barrak, Amine**, Ranim Trabelsi, Fehmi Jaafar, and Fabio Petrillo. "Exploring the impact of serverless computing on peer to peer training machine learning." In 2023 IEEE International Conference on Cloud Engineering (IC2E). IEEE, 2023 [5].

POSTERS AND SHORT PAPERS

1. **Barrak, Amine**. "Best Practices for Scalable and Efficient Distributed Machine Learning with Serverless Architectures." In 2024 IEEE 38th International Parallel & Distributed Processing Symposium (IPDPS-PhD Forum). IEEE, 2024.

2. **Barrak, Amine**. "The Promise of Serverless Computing within Peer-to-Peer Architectures for Distributed ML Training." In Proceedings of the AAAI Conference on Artificial Intelligence, Doctoral Consortium, 2024 [6].

3. **Barrak, Amine**. "Incorporating Serverless Computing into P2P Networks for ML Training: In-Database Tasks and Their Scalability Implications (Student Abstract)." In Proceedings of the AAAI Conference on Artificial Intelligence, 2024 [7].

4. **Barrak, Amine**, Fabio Petrillo, and Fehmi Jaafar. "Architecting Peer-to-Peer Serverless Distributed Machine Learning Training for Improved Fault Tolerance." arXiv preprint arXiv:2302.13995 (2022) [8].

The following publications are not directly related to the material presented in this thesis, but were produced in parallel with the research performed for this thesis.

OTHER PUBLICATIONS

1. **Amine Barrak**, Gildas FOFE, Leo MACKOWIAK, Emmanuel KOUAM, and Fehmi Jaafar. "Securing AWS Lambda: Advanced Strategies and Best Practices." In The 11th IEEE International Conference on Cyber Security and Cloud Computing. IEEE, (CSCLOUD 2024) [9].

2. Bourreau Hugo, Emeric Guichet, **Amine Barrak**, Benoit Simon, and Fehmi 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 [10].

3. Fehmi Jaafar, Darine Ameyed, **Amine Barrak**, and Mohamed Cheriet. "Identification of compromised IoT devices: Combined approach based on energy consumption and network traffic analysis." In 2021 IEEE 22nd International Conference on Software Quality, Reliability, and Security Companion. IEEE, (QRS 2021) [11].

CHAPTER I

INTRODUCTION

Serverless computing, a cloud-computing execution model, has rapidly gained popularity due to its ability to significantly reduce operational complexity and cost. In serverless architectures, the cloud provider manages the allocation of machine resources, not just server space but also the execution of code. This model allows developers to focus solely on the functionality of their applications without worrying about the underlying infrastructure [12]. Serverless functions are event-driven, automatically scaled, and executed in stateless compute containers that are ephemeral and fully managed by the cloud provider [13]. The pay-as-you-go pricing strategy of serverless computing optimizes costs by billing users only for the actual resources consumed during execution [14]. This model is ideal for event-driven scenarios, enhancing efficiency and responsiveness in real-time applications [15].

The primary advantages of serverless computing include scalability and cost efficiency. The scalability is managed by the cloud provider, who dynamically allocates resources to meet the demand of the application. This is especially beneficial for applications with variable workloads, as it ensures that resources are not wasted during idle times [16]. Moreover, the pricing model of serverless computing is based on the actual amount of resources consumed by an application, rather than on pre-purchased units of capacity, which can lead to significant cost reductions [17],[18].

As the adoption of serverless computing expands, its application to complex tasks such as machine learning is becoming increasingly feasible yet challenging [19, 20]. Machine learning models often require extensive computational resources for tasks like training, where serverless computing can offer significant scalability and cost benefits[21]. However, the

stateless and ephemeral nature of serverless functions poses unique challenges for persistent tasks like training machine learning models, which require managing large datasets and maintaining state across multiple training iterations [22]. Training large-scale ML models on serverless infrastructures can be limited by the performance and relative advantages over traditional serverful infrastructures, posing additional challenges [23]. Despite these challenges, serverless computing remains a promising paradigm for machine learning tasks due to its dynamic resource allocation and cost effectiveness in specific scenarios [24], [25]. However, the architecture chosen for training these models can greatly influence the effectiveness and efficiency of serverless applications [26], [21], [27], [28].

Machine learning training can be performed using various architectures, each with unique characteristics and implications for serverless environments [29, 23, 30, 31]. Centralized training typically involves a single, powerful machine or cluster handling the entire training process [32]. While this can simplify the management of state and resources, it does not naturally align with the stateless and distributed nature of serverless computing [25]. Centralized systems often face limitations in scalability and a single point of failure, which serverless computing aims to overcome by distributing workloads across multiple ephemeral instances [33, 34].

Distributed training systems, such as peer-to-peer networks, utilize multiple machines to handle different parts of the training process, often requiring significant coordination and data transfer between nodes [35, 36]. These distributed systems can benefit greatly from serverless computing due to its scalability, but they face challenges such as synchronizing state across ephemeral and stateless functions [37, 38]. This synchronization is crucial to ensure consistency and accuracy in the training process, yet it can introduce significant communication overhead and complexity [39].

The challenges of using serverless architectures for machine learning training vary depending on the chosen topology. Resource allocation and management is critical in serverless environments, especially in distributed systems where orchestration of numerous serverless functions must be managed dynamically [40]. Efficient resource management ensures that computational tasks are handled effectively, minimizing idle times and reducing costs [41]. However, dynamic resource allocation in a serverless context must be finely tuned to avoid over-provisioning or under-utilization [42].

State management poses a significant challenge due to the stateless nature of serverless functions. Maintaining state across training sessions is particularly difficult in distributed systems where state must be synchronized across multiple, potentially ephemeral, compute instances. To address this, proposed solutions often rely on external databases [23, 43, 22] or message queues, such as RabbitMQ [44], as communication channels to preserve state and facilitate coordination among serverless functions.

Communication overhead is a major concern in distributed learning topologies [45]. These systems require robust communication strategies to handle the exchange of model updates, which can be bandwidth-intensive and slow if not optimized properly. Some solutions have been proposed to address these issues, one approach involves removing bottlenecks on nodes performing aggregation tasks by proposing scattereduce computing [23], another strategy is to exchange only significant updates between nodes [46]. Additionally, gradient compression techniques can be employed to minimize data transfer latencies [47, 48].

Fault tolerance is essential in training architectures, especially in serverless settings where individual function instances may be transient. Ensuring that machine learning training can proceed in the face of such failures is crucial for the reliability of the system. This involves implementing robust mechanisms for detecting and handling worker failures, redistributing

workloads, and maintaining training progress without interruption [45]. Additionally, the system must be resilient against threats such as Byzantine gradients, where malicious or faulty nodes could send incorrect gradient updates.

The unique requirements of different machine learning training topologies highlight the necessity for tailored serverless solutions. These solutions must address the specific challenges of state management, communication overhead, resource allocation, and fault tolerance in order to harness the full potential of serverless computing. This research aims to bridge this gap by designing and evaluating architectures that efficiently integrate machine learning processes within serverless environments. By developing robust serverless frameworks tailored to the needs of machine learning, this work seeks to advance the field and provide practical solutions for real-world applications.

1.1 THESIS HYPOTHESIS

In the context of machine learning training, an optimal environment refers to a system that maximizes scalability, cost efficiency, and performance while minimizing resource waste, over-provisioning, and synchronization issues. It should efficiently handle resource-intensive tasks like gradient computation, provide automatic scaling, reduce latency in communication, and offer fault tolerance to recover from failures without compromising training progress. Serverless computing, with its ability to dynamically scale resources on-demand, eliminates the need for over-provisioning, where resources are allocated but underutilized, thus improving cost efficiency and performance.

A reliable machine learning environment ensures that the training process remains secure and trustworthy, even in the presence of faulty or malicious peers. In distributed peer-to-peer (P2P) architectures, reliability means detecting and mitigating untrustworthy

behavior, maintaining the integrity of the model, and continuing training without degradation. This involves secure communication, authentication, and fault-tolerant mechanisms that allow the system to recover and proceed correctly. Therefore, we present our research hypothesis composed of four sub-hypotheses:

> **Thesis Hypothesis:**
> Serverless computing provides an optimal environment for scalable and reliable machine learning training.

To answer this, four sub-hypotheses arise:

1. **Sub-Hypothesis 1:** Using serverless functions for parallel gradient computation accelerates the training process but introduces specific challenges.

   We systematically review the research literature, including over 150 studies, to understand the usage of serverless computing functions within machine learning pipelines, from data preprocessing to model deployment. We examine the primary technologies and frameworks utilized for integrating machine learning models into serverless environments, such as AWS Lambda, Google Cloud Functions, and Apache OpenWhisk. Focusing on the ML training phase, we compare the training process between traditional serverful architectures and serverless architectures across different batch sizes. Our analysis delves into the challenges of resource management and synchronization in distributed training, particularly within peer-to-peer (P2P) networks, and highlights the benefits of serverless functions for parallel and accumulative gradient computation. This is detailed in Chapters 3 and 4.

2. **Sub-Hypothesis 2:** Advanced techniques such as using RedisAI for machine learning operations within the database can reduce communication overhead in serverless distributed training environments.

   We explore techniques to reduce the communication overhead introduced by performing distributed training on serverless functions. The stateless nature of the architecture necessitates continuous reliance on databases to maintain state and facilitate communication between workers. To address this challenge, we introduce advanced techniques designed to minimize this overhead. One such technique is gradient compression, which reduces the volume of data exchanged between workers during training. Additionally, a major proposed optimization is performing ML operations directly within the database, such as averaging the computed gradients already saved within the database and updating the model within the database. This is detailed in Chapter 5.

3. **Sub-Hypothesis 3:** Implementing key mechanisms for fault tolerance and secure communication protocols improves the reliability and security of serverless machine learning training.

   We implement several key mechanisms to enhance fault tolerance and secure communication protocols in serverless machine learning training. Firstly, we ensure secure communication between workers, particularly when adding new workers to the training network, by using encryption protocols to protect data integrity and confidentiality. We employ a heartbeat mechanism to monitor the status of each worker, allowing us to promptly detect and respond to any failures. To address fault tolerance, we develop a mechanism to handle worker failures; if a worker goes down, its assigned dataset is redistributed among the remaining workers, ensuring continuous training. Additionally, to protect against Byzantine attacks, we incorporate a method to exclude gradient outliers.

These implementations collectively enhance the reliability and security of the serverless machine learning training process. This is detailed in Chapter 6.

4. **Sub-Hypothesis 4:** The proposed SPIRT architecture offers a fully serverless training workflow with superior training speed, performance, and fault tolerance compared to existing serverless ML training frameworks.

   Lastly, we implement a fully serverless ML training architecture inspired by the peer-to-peer mechanism to avoid a single point of failure. The training flow for each peer is orchestrated by a state machine for every training epoch. This architecture is scalable in two dimensions: by increasing the number of serverless functions to compute gradients within a peer and by adding more peers. We compare this architecture with state-of-the-art contributions by evaluating training time, architecture cost, accuracy, and mechanisms employed to reduce communication overhead. This is detailed in Chapters 7 and 8.

1.2 SUMMARY OF THESIS CONTRIBUTIONS

We present a general overview of the various contributions of this dissertation, organized by chapters. Figure 1.1 provides a general overview of the four parts of this thesis and the outputs of each part.

1.2.1 CONTRIBUTIONS OF CHAPTER 3 (SUB-HYPOTHESIS 1)

First major contribution : A systematic mapping study on serverless computing functions within the ML pipeline, from data preprocessing to model deployment. It examines the used technologies and used frameworks. It discusses challenges in literature such as cost and pricing, cold start, and resource scalability.

C1: Addressing Key Questions in Serverless Computing for ML Pipelines: We conducted a systematic mapping study of over 200 papers, refined with different exclusion steps to 53 papers published between 2018 and 2022, to understand the use of serverless computing functions within ML pipelines. This review includes an examination of primary technologies and frameworks such as AWS Lambda, Google Cloud Functions, and Apache OpenWhisk. The study addresses key questions: What are the publication trends of research studies about serverless on machine learning? What is the focus of research of applied machine learning on serverless computing? What are the potential challenges of adopting machine learning on serverless computing? The study discusses significant challenges such as cost and pricing, cold start issues, and resource scalability. It also explores the benefits of serverless architectures for simplifying deployment, improving scalability, and reducing operational overhead, while highlighting areas for future research and potential improvements in serverless ML pipeline implementations.

1.2.2 CONTRIBUTIONS OF CHAPTER 4 (SUB-HYPOTHESIS 1)

Second major contribution : Introducing a novel serverless-based architecture for efficient peer-to-peer machine learning training, enabling parallel gradient computation and enhanced resource utilization.

C2: Exploring the Impact of Serverless Computing on Peer-to-Peer Training Machine Learning: We proposed a novel architecture integrating serverless computing into peer-to-peer (P2P) networks for distributed machine learning training, aiming to answer how serverless computing can fater training and improve scalability, what benefits and challenges it presents? This architecture leverages serverless functions for parallel gradient computation within each peer, reducing computational resource demands. It incorporates AWS Lambda

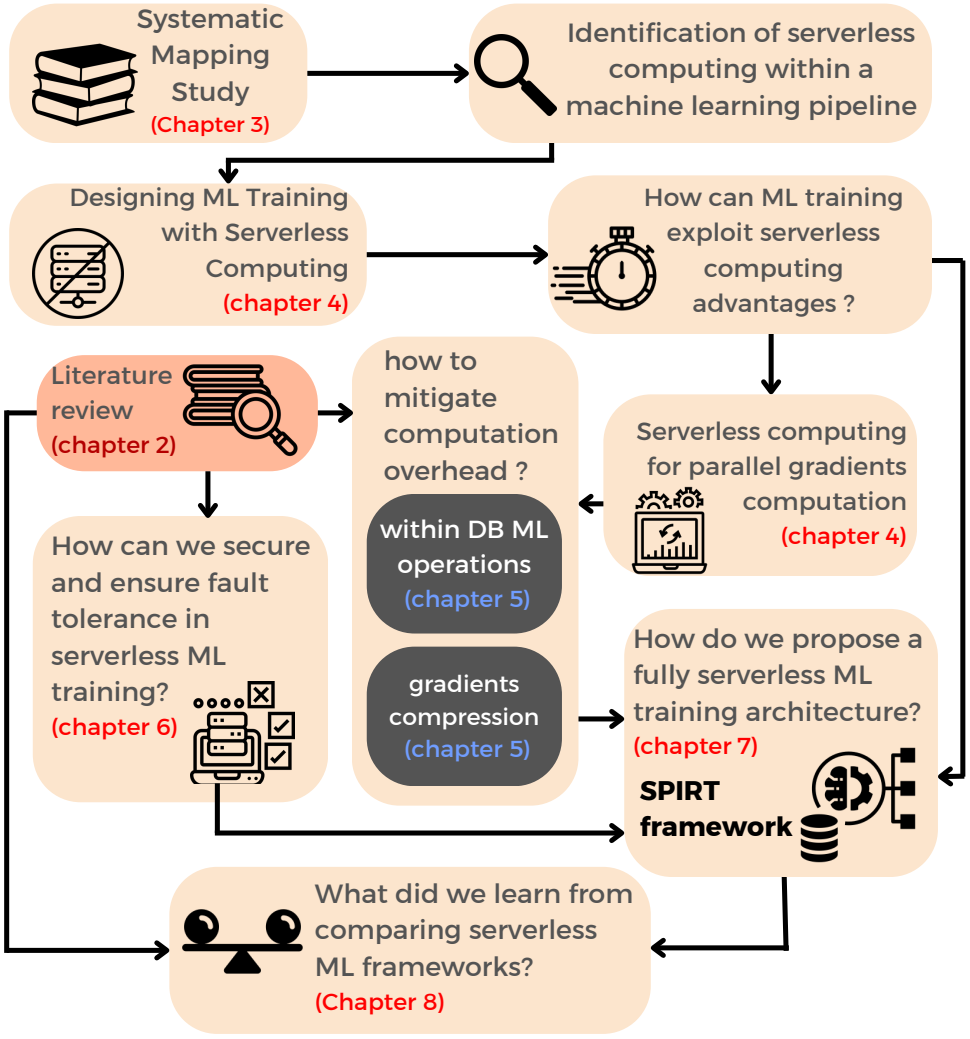


Figure 1.1 : Overview of the thesis methodology

for batch processing and gradient computations. Gradient exchange between peers is ensured using RabbitMQ, and synchronization mechanisms maintain training consistency. This study highlights the potential of serverless computing to optimize distributed ML training, providing a scalable and cost-effective solution for complex machine learning models in P2P networks.

1.2.3 CONTRIBUTIONS OF CHAPTER 5 (SUB-HYPOTHESIS 2)

Third major contribution : We developed a novel in-database gradient averaging and model updating approach using RedisAI, significantly improving training efficiency by reducing data transfer latencies.

C3: Reducing Communication Overhead in Serverless Distributed Training: In serverless distributed training, we addressed the challenge of communication overhead by optimizing gradient compression and in-database operations. How does the number of workers affect computation and communication overhead? Can gradient compression techniques effectively reduce communication overhead? How can in-database operations improve the efficiency of distributed training? We proposed compressing the communicated gradients and designed an architecture where critical operations *i.e.,*gradients averaging and model update, are conducted within the database using RedisAI. By centralizing these computations within the database environment, we aim to improve the training time and cost.

1.2.4 CONTRIBUTIONS OF CHAPTER 6 (SUB-HYPOTHESIS 3)

Fourth major contribution : We developed a secure communication mechanism between peers, implemented fault tolerance strategies, and enhanced the overall reliability of serverless distributed training systems.

C4: Enhancing Security and Fault Tolerance in Serverless Distributed Training: In this contribution, we tried to answer these research questions: How can secure communication be ensured in a serverless distributed training environment? What strategies can be implemented to maintain fault tolerance and manage peer failures effectively?

We implemented secure peer-to-peer communication using RSA cryptography and an authentication process based on signature verification, managed through SQS queues. For fault tolerance, we developed mechanisms to handle peer failures and integrate new peers, including dynamic data redistribution and a heartbeat monitoring system. Additionally, we incorporated robust aggregation algorithms to maintain model accuracy and reliability in the presence of Byzantine attacks. These measures created a resilient and secure serverless distributed training framework.

1.2.5 CONTRIBUTIONS OF CHAPTER 7 (SUB-HYPOTHESIS 4)

Fifth major contribution : We proposed a scalable, peer-to-peer distributed training framework with a fully serverless ML architecture to optimize ML training, enhance communication efficiency, and ensure fault tolerance.

C5: SPIRT: A Fault-Tolerant and Reliable Peer-to-Peer Serverless ML Training Architecture We aimed to answer the following research questions: How can we automate the ML training workflow within a serverless P2P architecture? How can we achieve scalability in serverless ML training with respect to acceptable communication overhead?

We proposed a scalable, peer-to-peer distributed training framework with a fully serverless ML architecture inspired by the peer-to-peer mechanism. The training flow for each peer is orchestrated by a state machine for every training epoch. This architecture is scalable in two dimensions: by increasing the number of serverless functions to compute gradients within a peer and by adding more peers.

Our framework includes robust fault tolerance and secure communication protocols, ensuring reliable distributed ML training in the presence of potential disruptions or adversarial

actors. By leveraging in-database model updates and orchestrated workflow coordination, SPIRT minimizes communication overhead and maximizes operational efficiency.

1.2.6 CONTRIBUTIONS OF CHAPTER 8 (SUB-HYPOTHESIS 4)

> **Sixth major contribution :** A comprehensive comparative study of serverless ML training frameworks, evaluating their training time efficiency, cost implications, and communication overhead management. The study provides insights for optimizing serverless ML training processes.

C6: Advancing Serverless ML Training Architectures via Comparative Approach: How do various serverless ML training architectures compare in terms of training time efficiency? What are the cost implications of different serverless ML training frameworks? How do different serverless architectures manage communication overhead during training? This contribution presents a comprehensive comparative study of serverless ML training frameworks, including SPIRT, ScatterReduce, AllReduce, and MLLess. By evaluating these architectures, we analyze their performance in terms of training time, cost-efficiency, and communication overhead management. This study also examines fault tolerance and security measures within each framework. The findings provide valuable insights for optimizing serverless ML training processes, helping practitioners select the most suitable frameworks for their specific requirements.

1.3 ORGANIZATION OF THE THESIS

The remainder of this dissertation is organized as follows. Chapter 2 provides background and a comprehensive literature review on the relevant topics, setting the foundation for the subsequent chapters. Chapter 3 presents the first major contribution, a systematic mapping

study on serverless computing functions within the ML pipeline, examining the technologies and frameworks used, as well as discussing challenges such as cost and pricing, cold start issues, and resource scalability. Chapter 4 introduces a novel serverless-based architecture for efficient peer-to-peer machine learning training, focusing on parallel gradient computation and enhanced resource utilization. Chapter 5 addresses the challenge of communication overhead in serverless distributed training by optimizing gradient compression and implementing in-database machine learning operations. Chapter 6 explores secure communication mechanisms and fault tolerance strategies in serverless distributed training systems, ensuring reliability and resilience. Chapter 7 proposes a scalable, peer-to-peer distributed training framework with a fully serverless ML architecture, aimed at optimizing ML training and enhancing communication efficiency. Finally, Chapter 8 conducts a comprehensive comparative study of serverless ML training frameworks, evaluating their training time efficiency, cost implications, and communication overhead management. Chapter 9 concludes the dissertation by summarizing key findings, discussing limitations, and suggesting avenues for future research.

CHAPTER II

LITERATURE REVIEW

2.1 CHAPTER OVERVIEW

The chapter of Literature Review in this thesis aims to provide an overview of the existing literature and research related to the topics of distributed machine learning. It provides an in-depth analysis of the most relevant and recent research on serverless computing, distributed machine learning, and various communication architectures. Additionally, this chapter delves into fault tolerance and failure recovery in distributed machine learning and serverless computing environments.

Machine learning technology has proliferated in more complicated applications in recent years. Training ML models is a very data-intensive activity that faces large-scale computing issues with sophisticated models and data lakes, and data input may become a severe performance bottleneck [49].

The advantages of scaling up systems by adding programmable GPUs have been shown [50, 51]. Nevertheless, it will hit physical hardware constraints, and the requirement for scaling out offers various advantages, including cheaper equipment costs [31], and robustness against faults [52].

There are two fundamentally different ways of partitioning the computing across machines: parallelizing either the data or the model. Data is partitioned as many times as there are worker nodes in the Data-Parallel technique, and all worker nodes apply the same algorithm to separate datasets. All worker nodes have the same model (centralised or replicated), therefore a coherent output emerges [53]. The worker nodes that operate on various regions of the

model process precise copies of the whole datasets in the Model-Parallel method. As a result, the model is the aggregation of all model pieces [54].

The degree of distribution that the system is planned to implement is a decisive element for topology. It can have a substantial impact on its performance, scalability, dependability, and security. Based on the taxonomy of distributed communication networks by Baran [55], Verbraeken *et al.* [31] identified the following four topologies which are Centralized systems, Trees, Parameter server paradigm, and Peer-to-peer. (1) Centralized systems employs a rigorously hierarchical method to aggregation, which occurs in a single central location. (2) Trees are decentralised systems in which each node communicates solely with its parent and child nodes. In the AllReduce [56] paradigm, for example, nodes in a tree add their local gradients to those of their offspring and transfer this sum to their parent node to produce a global gradient. (3) The Parameter Server paradigm (PS) [57] combines a decentralised collection of workers with a centralised set of masters responsible for maintaining the common state. Because the parameter servers handle all communication, the topology has the problem of forming a bottleneck. and (4) Peer-to-Peer. Is a fully distributed approach, each node has its own parameters and workers interact directly. This eliminates single points of failure and provides more scalability than a centralised solution [58]. This architecture presents a high level of communication between nodes, Sufficient Factor Broadcasting (SFB) [59] has been proposed to reduce the communication overhead.

There are several techniques that enable the interleaving of parallel computation and inter-worker communication. BSP is the simplest model for coordinating processing and communication phases [60]. For every synchronisation barrier, workers must wait until all other workers are finished, which increases overhead if certain workers proceed slowly [61]. Stale Synchronous Parallel (SSP) reduces synchronisation overhead by letting quicker workers advance for a set number of repetitions. All workers stop if this number is exceeded.

Barrierless Asynchronous Parallel [62]/Total Asynchronous Parallel [63] (BAP/TAP) allows worker machines communicate without waiting. The benefit is that it usually speeds up the most. The model may converge slowly or inaccurately because, unlike BSP and SSP, the error accumulates with delay [62].

During this thesis, we propose to use a peer to peer distributed training machine learning to benefit from the power of splitting large data into several workers in a cloud environment. Moreover, in such architecture communication is usually asynchronous where each worker compute the gradient and wait for others to share their computed gradient in order to learn from their parts of the model.

2.2 DISTRIBUTED MACHINE LEARNING: AN OVERVIEW

Machine learning technology has proliferated in more complicated applications in recent years. Training ML models is a very data-intensive activity that faces large-scale computing issues with sophisticated models and data lakes, and data input may become a severe performance bottleneck [49].

The advantages of scaling up systems by adding programmable GPUs have been shown [50, 51]. Nevertheless, it will hit physical hardware constraints, and the requirement for scaling out offers various advantages, including cheaper equipment costs [31], and robustness against faults [52].

There are two fundamentally different ways of partitioning the computing across machines: parallelizing either the data or the model. Data is partitioned as many times as there are worker nodes in the Data-Parallel technique, and all worker nodes apply the same algorithm to separate datasets. All worker nodes have the same model (centralised or replicated), therefore a coherent output emerges [53]. The worker nodes that operate on various regions of the

model process precise copies of the whole datasets in the Model-Parallel method. As a result, the model is the aggregation of all model pieces [54].

The degree of distribution that the system is planned to implement is a decisive element for topology. It can have a substantial impact on its performance, scalability, dependability, and security. Based on the taxonomy of distributed communication networks by Baran [55], Verbraeken *et al.* [31] identified the following four topologies: Centralized systems, Trees, Parameter server paradigm, and Peer-to-peer. (1) Centralized systems employ a rigorously hierarchical method to aggregation, which occurs in a single central location. (2) Trees are decentralized systems in which each node communicates solely with its parent and child nodes. In the AllReduce [56] paradigm, for example, nodes in a tree add their local gradients to those of their offspring and transfer this sum to their parent node to produce a global gradient. (3) The Parameter Server paradigm (PS) [57] combines a decentralized collection of workers with a centralized set of masters responsible for maintaining the common state. Because the parameter servers handle all communication, the topology has the problem of forming a bottleneck. and (4) Peer-to-Peer. In a fully distributed approach, each node has its own parameters and workers interact directly. This eliminates single points of failure and provides more scalability than a centralized solution [58]. This architecture presents a high level of communication between nodes. Sufficient Factor Broadcasting (SFB) [59] has been proposed to reduce the communication overhead.

There are several techniques that enable the interleaving of parallel computation and inter-worker communication. BSP is the simplest model for coordinating processing and communication phases [60]. For every synchronization barrier, workers must wait until all other workers are finished, which increases overhead if certain workers proceed slowly [61]. Stale Synchronous Parallel (SSP) reduces synchronization overhead by letting quicker workers advance for a set number of repetitions. All workers stop if this number is exceeded.

Barrierless Asynchronous Parallel [62]/Total Asynchronous Parallel [63] (BAP/TAP) allows worker machines to communicate without waiting. The benefit is that it usually speeds up the most. The model may converge slowly or inaccurately because, unlike BSP and SSP, the error accumulates with delay [62].

During this thesis, we propose to use a peer-to-peer distributed training machine learning to benefit from the power of splitting large data into several workers in a cloud environment. Moreover, in such architecture, communication is usually asynchronous where each worker computes the gradient and waits for others to share their computed gradient in order to learn from their parts of the model.

2.3 SERVERLESS COMPUTING IN THE CONTEXT OF DISTRIBUTED ML

Serverless computing has several computing capabilities, *i.e.,*, cost optimization [64], high scalability [16], that fit the need of the expensive computing operations of machine learning.

The usage of serverless runtimes has gained popularity as a means of handling machine learning workloads more effectively, particularly when training complex ML models. Many efforts have been made to promote the wider adoption of serverless runtimes and to effectively use FaaS platforms for properly handling ML workloads. Several serverless development frameworks, such as Pywren [65], which provides a specific map-reduce framework for serverless architecture, have been proposed to this goal. This framework enables users to execute their Python code function on a serverless platform with minimal concern for the underlying infrastructure, and provides efficient resource utilization and scalability for large-scale data processing tasks. The approach involves fragmenting the input data into smaller

units, utilizing the function-as-a-service (FaaS) concept, and generating a separate function for each data fragment to achieve massively parallel execution.

Additionally, external storage is employed in a functional manner for intermediate computation results to avoid the serverless stateless mode delicately, which can impact data locality and incurs extra network communications to fine-grained compute tasks, thereby causing inefficiencies that are prohibitive to system performance [66].

Crucial [38], another framework, is introduced as a framework for developing highly concurrent stateful applications on serverless architectures. It uses a simple programming model and allows you to easily transfer multithreaded algorithms to the FaaS environment. Crucial offers shared memory, fine-grained state management, fault tolerance, and low overhead, making it a suitable option for stateful applications such as machine learning. The core of Crucial is predominantly based on cloud thread. Essentially, it maps a thread to the invocation of a cloud function. The cloud threads manage the global shared state, which is stored in the shared or distributed DSO object layer and can be either ephemeral or persistent. Experimental outcomes demonstrate that Crucial delivers comparable or superior performance to Apache Spark [67], while executing substantially faster and at a similar cost.

Cirrus [68], on the other hand, has been meticulously designed to proficiently manage the entire machine learning (ML) workflow, with particular attention devoted to serverless, fine-grain, data-intensive serverless ML training, and hyperparameter optimization. Cirrus provides an interface that enables the execution of scalable ML training by leveraging the high scalability of serverless computing environments. This is achieved through the implementation of a parameter server on top of a virtual machine (VM) that serves as the storage access point for the global model shared by the executors, which are implemented using AWS Lambda functions. The centralised PS layer serves as the communication hub for all Function-as-a-

Service (FaaS) workers. The hybrid design of Cirrus is particularly noteworthy due to the ability of parameter servers to perform computation, which results in a substantial reduction of up to 200% in communication costs compared to indirect communication via external storage. As delineated by [69], Cirrus is shown to be between 3X-5X faster than VMs, although the platform can be up to 7X more costly.

Another proposed approach is SMLT [25], which is a serverless framework that features scalability, and user-centric deployment (*i.e.,*, training deadlines and budget limits) machine learning tasks. The framework employs an automatic and adaptive scheduling mechanism to dynamically optimize the deployment and scaling of resources during the training process. With the objective of fulfilling user-specified training timelines and budgetary restrictions. The architecture of SMLT is predicated on a parameter server-based model where workers are responsible for training and updating the global model, and uses external storage for intermediate computation results. SMLT adopts a Hybrid Storage Enabled Hierarchical Model Synchronization, which uses a fast storage medium (*e.g.,*, Redis) to satisfy the latency-sensitive demands of the synchronization scheme, and a cloud-based (*e.g.,*, AWS S3) for infrequent data to balance performance and cost. The hierarchical synchronization mechanism takes model gradients generated by each worker as input, and the shard generator in each worker divides the input gradients into equal-sized shards. These shards are uploaded to the parameter store, which acts as a communication intermediary between the stateless serverless workers. Each serverless worker also acts as a shard aggregator, downloading and aggregating its assigned shards generated by all workers. The resulting aggregated shard is then uploaded to the parameter store by each shard aggregator. Finally, the global aggregator residing in each worker downloads all aggregated shards, reconstructing the updated model for the next iteration. Furthermore, by providing an end-to-end design, SMLT mitigates the inherent challenges of serverless platforms and effectively realizes the transparency and scalability advantages of

serverless computing. Extensive experimental results demonstrate that SMLT outperforms state-of-the-art VM-based systems and existing serverless ML training frameworks in both training speed (up to 8x) and monetary cost (up to 3x).

Alternative works [43, 44] center their attention on the efficacious utilization of the definitive properties of the Function-as-a-Service (FaaS) model, such as the "pay-as-you-go" model, which facilitates financial savings through the allocation of serverless workers in a granular and adaptable manner. In this context, a research Paper [43] introduces the concept of λDNN, a framework that endorses the judicious allocation of functional resources, with the aim of minimizing the monetary expenses associated with Deep Neural Network (DNN) training workloads on serverless platforms, while simultaneously guaranteeing optimal performance. λDNN builds a high-level DNN training performance model, which operates effectively in serverless platforms by harnessing the Parameter Server (PS) network and function CPU utilization. The primary goal of DNN is to optimize functional resource provisioning plans in order to provide predictable performance in the context of serverless DNN training workloads from the user's perspective while simultaneously lowering resource costs from the service provider's perspective.

Finally, in order to securely train machine learning algorithms within a serverless environment, FedLess [70] proposed a serverless based federated learning framework that can surmount certain challenges associated with traditional federated learning methods, such as intricate infrastructure management and scalability, by leveraging the advantages of serverless computing. By providing crucial features like authentication, authorization, and differential privacy, FedLess facilitates federated learning across a diverse range of heterogeneous Function-as-a-Service (FaaS) providers. The FedLess framework enables data owners to collaborate in training machine learning while keeping the data local. The framework employs a serverless architecture that automatically adjusts resource allocation based on demand, guar-

anteeing cost-effectiveness and scalability. The authors validate the effectiveness of FedLess by presenting a case study of training a machine learning model utilizing federated learning on the MNIST dataset.

The evaluation results demonstrate that FedLess can achieve accuracy levels comparable to centralized training, while offering significant cost savings. The performance of FedLess in terms of latency and scalability was measured by varying the number of clients and model parameters. The experiments indicate that FedLess can achieve low latency and efficiently scale for large-scale federated learning tasks.

During this thesis, serverless computing will be used as outsourcing infrastructure management to a cloud provider, and provides benefits such as reduced operational costs, improved scalability, simplified deployment, and increased agility. By leveraging these benefits, the thesis aims to enable distributed machine learning training with improved fault tolerance. Overall, the use of serverless computing can bring significant advantages to the development and deployment of distributed machine learning systems.

Parallelization of the training process across multiple machines has the potential to significantly reduce training time while improving model accuracy. One popular way is data parallelism, which divides the data into divisions and distributes it over numerous processors that train the shared machine learning model locally.

The Parameter Server (PS) [71] is a distributed computing architecture for large-scale machine learning. It separates model parameter storage from gradient computation during training. The parameter server stores model parameters, while worker nodes calculate gradients in parallel. The server collects and updates the model parameters using the computed gradients. This approach enhances scalability and performance of machine learning systems, making it an essential tool for training large models on big data.

DistBelief [72] puts forth distributed model parallelism techniques for parameter server training using two pivotal algorithms, namely Downpour SGD and Sandblaster L-BFGS. In the former approach, the training data is split into small batches, and each worker node in the distributed system is assigned a subset of these batches. Each worker then autonomously calculates the gradients for the neural network weights using its local data batch and transmits them to the parameter server, also known as the "master" in DistBelief, for aggregation. The parameter server is responsible for integrating the gradients from all the workers, computing the updated weights, and subsequently disseminating them to the workers for the next iteration of training.

The parameter server approach has improved with key-value pairs, range-based updates, and vector clocks for better communication efficiency, scalability, and fault tolerance. In a recent study [71], this approach used two roles: server and worker nodes. The server nodes form a cohesive group that exclusively manages a specific subset of the model parameters. Worker nodes are grouped into cohorts and assigned specific data and tasks. Communication between server nodes ensures that each one is responsible for its own subset of parameters. The server group manages updates to the parameters. Conversely, direct communication between workers is not possible, and workers are only permitted to communicate with their respective servers through a task scheduler, which helps to promote fault tolerance and maintain system scalability.

The parameter server architecture represents the model as key-value pairs and uses key-value vectors as data structures. This approach promotes sparsity and optimizes linear algebra libraries. Push-and-pull operations transfer data between nodes, and range-based updates enhance network bandwidth efficiency. Asynchronous tasks boost system performance but may reduce convergence speed. To address this issue, the server offers flexible consistency, including sequential, eventual, and bounded delay. The servers store parameters

using consistent hashing, and entries are replicated using chain replication to ensure fault tolerance. The system optimizes range-based communication, compressing both data and vector clocks. Vector clocks track aggregation status, and modifications are synchronized between master and slave nodes.

The PS architecture is gaining popularity in serverless environments for effectively distributing learning workloads across numerous clusters. Cirrus [68] is a sophisticated machine learning (ML) system that leverages a parameter server (PS) architecture to distribute computation across functions-as-a-service (FaaS) workers that communicate with the central PS layer. Cirrus's PS implementation follows a master-worker model, where the master node is responsible for preserving the global model parameters, while the worker nodes execute the computational tasks and update the model parameters. Specifically, each worker node processes a subset of the training data and communicates the computed gradients to the master node. The master node subsequently aggregates the gradients and adjusts the global model parameters. To facilitate communication and synchronization among the nodes in the distributed system, Cirrus employs the Open MPI library, a widely-used message-passing interface (MPI) for parallel computing. The system's distributed data store comprises an interface that supports all use cases for intermediate data storage in the ML workflow, including a key-value store interface (set/get) and a parameter-server interface (send gradient / get model).

The objective of the data store is to provide rapid access to shared intermediate data by Cirrus's workers. Several optimizations are employed to achieve this goal, including the development of a multithreaded server that distributes work across multiple cores to update models with high throughput.

Additionally, data compression is implemented for the gradients and models transferred to/from the store to reduce pressure on the network links of the store. The optimization reduces the amount of data transferred by up to 2x. Furthermore, the data store employs sending and receiving sparse gradients and models data structures to minimize data transfer by up to 100x. The modular design of the data store makes it effortless for users to introduce new ML optimization algorithms, such as Adam.

Numerous empirical studies conducted on AWS Lambda have revealed that the unpredictable performance of serverless deep neural network (DDNN) training can primarily be attributed to the resource bottleneck of Parameter Servers (PS) and small local batch sizes. Consequently, λDNN [43], a sophisticated and cost-efficient function resource provisioning framework, was designed and implemented to facilitate predictable performance for serverless DDNN training workloads, while reducing the budget of provisioned functions. Leveraging the network bandwidth of PS and the CPU utilization of functions, a succinct analytical performance model for serverless deep distributed neural network (DDNN) training was devised. To adopt the PS architecture in λDNN, two primary factors were considered: its extensive usage in machine learning clusters for production purposes and the difficulties in implementing the Ring-AllReduce training architecture on serverless platforms. VM instances with sufficient resources are used as the PS to effectively aggregate gradients gathered from workers. The input training data is stored in a distributed storage and partitioned to the functions. During each iteration, the functions compute and push the gradients to the PS, which updates the model parameters. The workers communicate with the PS through TCP connections. This iterative training process continues until the training loss reaches a specified value or number of epochs.

FedLess [70] is a proposed approach that utilizes a serverless architecture for aggregating global model parameters. MongoDB was selected as the parameter server owing to its

robustness, reliability, replication support, and security features [73] . During the training process, each node computes updates to the model parameters based on its local data. Once training is complete, the clients upload their parameters to the parameter server. Subsequently, the FedLess controller initiates the model aggregation by invoking the aggregator function after ensuring that all clients have either finished, reached a configured timeout, or failed. The aggregator function loads the client results from the parameter server and aggregates the updates using a specified algorithm, such as Federated Averaging. The resulting aggregated model parameters are then returned to the nodes for further local training.

Over time, several extensions to PS have been proposed to mitigate the communication bottleneck, such as reducing the communication load [74, 75, 76, 77]. Other techniques include introducing asynchronous updates [78, 79], which enables worker nodes to update the parameters independently without waiting for the server's response. This reduces the communication overhead and enhances the training speed. Additionally, training with multiple servers [80, 81] is suggested as an alternative to relying on a single server to distribute the parameter updates. This approach can significantly enhance the system's scalability by accommodating more workers to the training process.

The thesis differs from a parameter server architecture by utilizing a peer-to-peer (P2P) architecture, which can improve fault tolerance by reducing reliance on a centralized server. The thesis aims to compare the fault tolerance of peer-to-peer (P2P) and parameter server architectures in distributed machine learning training. To do so, we will run tests using simulated failures and measure the system's ability to recover and continue training. Specifically, we will compare the time to recover and the impact on training accuracy in the event of a failure. The P2P architecture is expected to perform better due to its decentralized nature, which can reduce the impact of failures and enable more efficient recovery. Overall, the tests

will demonstrate the improved fault tolerance of P2P architectures in distributed machine learning training.

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 [82]. 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[23] and SMLT [25] 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 [23] and [25] 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 [83] 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 [84] 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 [85] for serverless ML training systems that rely on databases, introducing in-database model updates ton eliminate the traditional fetch-process-reupload cycle.

2.4 COMMUNICATION ARCHITECTURES IN DISTRIBUTED ML

Distributed machine learning relies heavily on efficient communication architectures to manage the exchange of data and synchronization of model parameters across multiple nodes. Effective communication architectures are essential for ensuring scalability, fault tolerance, and performance in distributed ML systems. This section explores three primary communication architectures: Peer-to-Peer (P2P) Systems, Parameter Server Systems, and Communication and Model Synchronization in Serverless Distributed ML.

2.4.1 PEER-TO-PEER (P2P) SYSTEMS

Peer-to-peer (P2P) architecture in ML systems involves independent nodes (peers) communicating directly without a centralized server. This fully distributed architecture trains a common ML model while keeping node data local, addressing privacy concerns (*e.g.,*GDPR [86]) and making it suitable for federated learning [87]. Wink and Nochta [88] proposed a P2P approach using n-of-n secret sharing and secure weight aggregation to prevent reverse engineering and membership inference attacks [89, 90].

P2P architectures are more resilient and fault-tolerant than parameter-server architectures, avoiding the Single Point Of Failure (SPOF) issue. Peers ensure the minimum required

number are functional before each training round, with mechanisms like pre-configured timeouts to handle missing participants [88].

Guerraoui *et al.* [45] introduced an open-source library comparing Byzantine fault resilience between server-based and fully decentralized P2P architectures. Their framework uses robust aggregation [91] with four aggregation rules (GARs): Median [92], Krum [91], MDA [93], and Bulyan [94]. They found that tolerating Byzantine servers incurs more overhead and cost than tolerating Byzantine workers.

P2P's scalability can handle hundreds of thousands of participants using gossip protocols, allowing scalable and efficient convergence [95]. TRIBLER [96], a decentralized search engine, uses gossip protocols for user recommendations.

However, P2P's main limitation is its high communication cost. The GARFIELD approach [45] showed that aggregation time in P2P is double that of single-server approaches, with higher CPU and GPU consumption. P2P architectures are fault-tolerant but come at a high computing and aggregation cost.

Another limitation is the honest-but-curious threat model, where participants follow protocols while trying to learn from messages, risking peers' private data. Malicious participants can compromise the ML model by sending fake values during aggregation. Detecting and mitigating Byzantine nodes is challenging [97, 98, 99].

This thesis will use P2P architecture in distributed ML training to improve fault tolerance by distributing computation and data across nodes, reducing reliance on a centralized server. Combining P2P architecture with distributed training aims to enhance fault tolerance in ML systems, leveraging serverless computing for efficient and reliable distributed ML.

2.4.2 PARAMETER SERVER SYSTEMS

Parallelization of the training process across multiple machines can significantly reduce training time and improve model accuracy. Data parallelism, which divides data and distributes it over processors, is a common method.

The Parameter Server (PS) [71] architecture separates model parameter storage from gradient computation. The parameter server stores model parameters, while worker nodes calculate gradients in parallel. The server collects and updates the model parameters using the computed gradients, enhancing scalability and performance for large-scale machine learning.

DistBelief [72] introduces distributed model parallelism techniques using Downpour SGD and Sandblaster L-BFGS. Training data is split into batches, with each worker node calculating gradients and transmitting them to the parameter server for aggregation.

Improvements to the PS approach include key-value pairs, range-based updates, and vector clocks for better communication efficiency, scalability, and fault tolerance. Server nodes manage specific subsets of model parameters, and worker nodes communicate with their respective servers through a task scheduler to maintain system scalability.

The PS architecture uses key-value vectors, push-and-pull operations, and asynchronous tasks to optimize performance. Consistent hashing and chain replication ensure fault tolerance, while vector clocks track aggregation status.

Cirrus [68] leverages a PS architecture to distribute computation across Function-as-a-Service (FaaS) workers. It employs the Open MPI library for communication and synchronization, and its distributed data store provides rapid access to shared intermediate data. Optimizations include multithreaded servers, data compression, and sparse gradients to minimize data transfer.

Empirical studies on AWS Lambda revealed that the resource bottleneck of PS and small local batch sizes impact performance. Consequently, λDNN [43] was designed to facilitate predictable performance and reduce costs. Virtual Machine (VM) instances act as the PS, aggregating gradients from workers who communicate via TCP connections.

FedLess [70] uses a serverless architecture for aggregating global model parameters with MongoDB as the parameter server. Clients upload their parameters to the server, and the FedLess controller aggregates updates using a specified algorithm.

Extensions to the PS architecture include reducing communication load [74, 75, 76, 77], asynchronous updates [78, 79], and training with multiple servers [80, 81] to enhance scalability and reduce communication overhead.

This thesis utilizes a peer-to-peer (P2P) architecture, which can improve fault tolerance by reducing reliance on a centralized server. We will compare the fault tolerance of P2P and PS architectures through simulated failures, measuring recovery time and impact on training accuracy. The decentralized nature of P2P is expected to enhance fault tolerance and enable efficient recovery.

2.4.3 COMMUNICATION AND MODEL SYNCHRONIZATION IN SERVERLESS 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 [82]. 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 [23] and SMLT [25] 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 for making the aggregation of gradients of the different workers. The use of the 'scatterReduce' method used in [23] and [25] 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 [83] tried to reduce bottleneck communication over the functions responsible for making the aggregation by utilizing both uplink and downlink bandwidth of serverless functions.

Contrastingly, MLLess [84] 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 sends 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 gradient accumulation through parallel gradients in order to further reduce the communication overhead. Additionally, we modify RedisAI [85] for serverless ML training systems that rely on databases, introducing in-database model updates to eliminate the traditional fetch-process-reupload cycle.

2.5 FAULT TOLERANCE IN DISTRIBUTED MACHINE LEARNING

Fault tolerance refers to the ability of a system to continue functioning according to its specifications despite the occurrence of faults.

2.5.1 GENERAL FAULT TOLERANCE TECHNIQUES IN DISTRIBUTED ML

Fault tolerance in ML systems can be broadly classified into two categories: typical fault tolerance applicable to any distributed system and Byzantine fault tolerance specific to distributed ML optimization.

- Retry Method: A common reactive approach where a failed request is repeatedly attempted with a configured wait delay between attempts. For example, FEDKEEPER [100], a federated learning system based on FaaS architecture, uses this technique by re-invoking worker nodes if they return an error during local training.

- Checkpointing: This involves keeping regular records of the system's state, allowing the system to resume from the latest saved state in case of failure. Popular ML frameworks like TensorFlow and PyTorch utilize this method [101]. SCAR, proposed by Qiao et al. [102], enhances this by combining partial recovery and prioritized checkpoints, reducing the cost of partial failures by 78-95

- Request Replication: Bouizem et al. [103] proposed this approach, where multiple replicas work in parallel on the same request, using the first response generated. This method improves performance and availability but consumes more resources.

2.5.2 FAULT TOLERANCE IN DISTRIBUTED TRAINING OF ML MODELS

Distributed training in ML systems typically involves multiple nodes, each calculating its local gradient based on the mini-batch it has. These local gradients are then aggregated through a parameter server or via peer-to-peer communication, depending on the architecture. The aggregated gradients form the new common model parameters, which are sent back to the nodes. This process repeats until the model reaches the desired accuracy.

The primary goal of training an ML system is to minimize a loss function, which indicates the model's inaccuracy [104]. Stochastic Gradient Descent (SGD) [105] is a commonly used optimizer in distributed learning for this purpose.

2.5.3 FAULT TOLERANCE IN CLOUD AND SERVERLESS COMPUTING

Cloud computing is essential for distributed ML due to the vast amounts of data involved. Fault tolerance in the cloud is crucial for the reliability of these systems.

- Heartbeat Mechanism: This detection technique periodically sends signals to determine the reachability and liveness of components. Cloud providers like AWS maintain fault tolerance through redundant hardware, load balancing, auto-scaling, replication, monitoring, alerting, and disaster recovery [106].
- Cross-Region Replication: This involves replicating data across geographically distant regions to ensure high availability and disaster recovery protection. For instance, Microsoft Azure's cross-region replication and AWS's disaster recovery solutions provide resilience and reliability [107].
- Serverless Computing: Platforms like AWS Lambda offer built-in resilience features such as automatic scaling, multi-availability zone deployment, retry mechanisms for

transient failures, and dead-letter queues for analyzing failed events [108]. Barrak et al. [9] expand on these by addressing security vulnerabilities, focusing on securing communication and robust authentication in serverless environments.

- Fission Framework: Fission, built on Kubernetes, has three components: Function Pods, Router, and Executor. Its native fault tolerance method is the Retry method. If a Pod returns an error, the router keeps forwarding the function call until a response is received or the maximum number of retries is reached [109].

- Additionally, AWS Lambda ensures fault tolerance by deploying functions across multiple Availability Zones (AZs), which allows the system to continue functioning even in the event of an outage in a single AZ. It also incorporates retry mechanisms for transient failures, where functions are automatically re-executed if they fail due to temporary issues, enhancing the overall reliability of serverless applications [1].

2.6 APPROACHES TO TOLERATING BYZANTINE FAULTS IN ML

Byzantine faults refer to any arbitrary or unpredictable behavior of a component in the system. Hence, software and hardware mistakes, network issues as well as having nodes sending false or fake updates to compromise the model can all cause Byzantine failures in distributed systems. A simple strategy to compromise the common model by Byzantine nodes can be done in the following manner: Let's say, for example, that the Byzantine node wants the server to run on the alternative gradient ***"gl"*** and that the sum of the gradients of healthy nodes is ***"g"***. The Byzantine node can achieve this by controlling the update step at the server by transmitting ***"n gl - g"*** as its gradient where ***"n"*** is the total number of nodes in the system.

[1] https://docs.aws.amazon.com/lambda/latest/dg/security-resilience.html

The most common way to deal with Byzantine faults is to apply robust aggregation [91] through gradient aggregation rules (GAR). They can be referred to as gradient filters or screening rules as well. Many studies have focused on this approach and proposed different methods. These methods differ from each other depending on the assumptions they make on the number of Byzantine nodes compared to the total number of nodes in the system or the network topology.

In this regard, the updated step of the KRUM [91] method uses the local gradient that has the minimum distance to its ***"M - b - 2"*** nearest gradients where ***"M"*** is the total number of nodes and ***b*** is the number of Byzantine nodes. There is another variant of KRUM called MULTI-KRUM [91]. Multi-Krum chooses ***m*** vectors and averages them instead of choosing only one, where ***m*** is a hyperparameter. KRUM's time complexity is $O(n^2 d)$ where ***d*** refers to the model dimension.

The BULYAN algorithm [94], on the other hand, is a two-stage method. It chooses ***"n - 2b"*** local gradients in the first stage using recursive vector median techniques like geometric median and KRUM. The ***"n - 2b"*** selected gradients are subjected to a coordinate-wise procedure in the second stage, where ***"n - 4b"*** values in each coordinate are kept (and subsequently averaged for aggregation), and the ***"2b"*** values that are the furthest from the coordinate-wise median (CM) are eliminated. The problem with BULYAN is its incapacity to treat all attacks.

SIGNSGD algorithm [110] uses the gradient's element-wise sign (instead of the gradient itself) exclusively during the update step. The global update is decided by majority vote. SIGNSGD achieves a fast convergence. Yet, the majority voting needs more optimization to avoid a single node becoming a communication bottleneck.

KRUM and its variant as well as BULYAN and SIGNSGD are all examples of aggregation rules that can be applied in a distributed architecture based on parameter server. Therefore, there are other filtering rules that are applied on a fully distributed ML systems based on P2P. For instance, Yang and Bajwa [111] proposed the decentralized learning method BRIDGE (Byzantine-resilient decentralized gradient descent), which is a specific application of the distributed gradient descent (DGD) algorithm. The BRIDGE algorithm, which differs from the DGD algorithm by its screening phase before each update, is Byzantine resilient. The coordinate-wise trimmed mean screening technique eliminates all $\boldsymbol{b}$ values (biggest and smallest) in each dimension, where $\boldsymbol{b}$ represents the maximum number of Byzantine nodes the algorithm can support.

Aside from robust aggregation rules, other techniques can be applied to deal with Byzantine faults: Byzantine fault tolerance based on coding schemes and Byzantine fault tolerance based on Blockchain. For the first category, each node evaluates a redundant gradient rather than a single gradient according to the fundamental principle of the redundant mechanism. Each computing node has a redundancy rate, which more specifically represents the average number of assigned gradients. There are several works that focused on this matter like DRACO [112] and DETOX [113]. As for the second category, Blockchain is used for this matter since it is a decentralized system that is characterized by its auditability, privacy, and persistence. [114, 115] are examples of works that deal with Byzantine faults while making use of Blockchain technology.

2.7 EVALUATION OF DISTRIBUTED ARCHITECTURES

The present state of scientific research regarding the comparative evaluation of the Parameter Server (PS) architecture and the Peer-to-Peer (P2P) architecture for distributed machine learning is characterized by a notable paucity of studies. Nevertheless, the corpus

of literature which seeks to undertake such comparisons includes two seminal investigations of exceptional significance [45, 116]. These investigations employ a comprehensive array of metrics to evaluate the relative merits of the two architectures under consideration. Specifically, these metrics include accuracy (or convergence), throughput, scalability, and latency.

Initially, we shall commence with an analysis of the significant divergences pertaining to the throughput metrics. A thorough experiment was undertaken in GARFIELD [45] to quantify the overhead associated with the use of GARFIELD-based applications compared to other baselines, by conducting experiments that measured the throughput while training multiple models. The study observes that CPU-based deployments display higher slowdowns than GPU-based ones. The outcomes indicate that the overhead of attaining Byzantine resilience is substantial and varies depending on the fault-tolerant system employed, as well as the size and complexity of the model. Moreover, the Decentralized Learning architecture results in the most significant slowdown in throughput, highlighting the importance of carefully considering the trade-offs when choosing a fault-tolerant system for distributed machine learning.

To delve deeper into the matter, the average latency per iteration is computed for each deployment. The results indicate that the computation time remains relatively constant for all applications, approximately 1.6 seconds. However, the communication cost plays a crucial role in inducing overhead. Remarkably, the decentralized architecture exhibits the most substantial latency per iteration. Notably, the study observes that the aggregation time in decentralized learning is twice as large as that of SSMW, due to an additional model aggregation step undertaken by the former application. An intriguing observation made is that ML training, particularly on GPUs, is network-bound. In other words, applications requiring more communication incur more substantial slowdowns. It is worth noting that communication constitutes more than 75% of the overhead, indicating its significant impact on the system's overall performance.

In contradistinction to prior findings, in the paper [116], it was observed that a series of experiments were conducted for the purpose of providing a quantitative evaluation of various system architectures, including 1PS, 2PS, 4PS, and P2P systems, with the aim of evaluating their performance with the same basic classification ML tasks. Worker machines were deployed from one to seven machines in order to evaluate and quantify the throughput and latency of each system. An improvement in latency and throughput was observed in P2P architecture in comparison to 1PS, 2PS, and 4PS. The reason for this improvement was that the portion of the model was located on the same machine as the server. Furthermore, in this training, there was no need to extract the model from remote machines as it was already updated on the same machine.

Subsequently, we shall proceed to compare the two aforementioned architectures in regards to their scalability. In order to assess the scalability of the systems in GARFIELD [45], the number of workers is augmented, thereby increasing the effective batch size. The results indicate that all systems demonstrate scalability with the addition of more workers, except for the decentralized learning application. The PS architecture exhibited impressive scalability performance. However, the communication overhead of a P2P system is proportional to $O(P^2)$, where P represents the number of workers. This hinders its ability to scale to large-scale clusters when P reaches several thousand. Notwithstanding, the paper [116] did not conduct a comparison between the two architectures in terms of scalability. Nonetheless, a noteworthy observation was made regarding the PS architecture. Specifically, the latency of PS decreased as more machines were added to the system. However, the latency increased after a certain point (the fifth machine in the experiment) due to all-to-one communication and data overload, resulting in a CPU bottleneck. The distributed nature of the system causes a degradation of scalability after five machines and increases the epoch time required to complete the training cycle.

Moving forward, our attention is directed towards the comparison of accuracy and convergence rate between the PS and P2P architectures. It is worth noting that the convergence rate is subject to the specific model and its dimensions. In the context of the GARFIELD framework [45], using the CifarNet model, the empirical outcomes reveal that both architectures achieve comparable ultimate accuracies. However, with ResNet-50, it is observed that combining network asynchrony with decentralization results in the most significant accuracy degradation. Asynchrony leads to the aggregation of outdated models and gradients, thereby decelerating convergence and diminishing the final accuracy.

Ultimately, GARFIELD [45] presents another noteworthy observation regarding the PS architecture. Specifically, experiments were conducted to evaluate the PS architecture's performance under adversarial conditions, where attacks were applied to the baseline PyTorch deployment, the crash-tolerant protocol, and the MSMW system. The empirical results reveal that both the vanilla deployment and the crash-tolerant deployment fail to learn under attack conditions. However, the MSMW system is able to train the model safely and achieve convergence to a normal, high accuracy. These empirical findings suggest that the MSMW system may be more resilient to adversarial attacks than other architectures, and could potentially be a promising solution for improving the security and robustness of machine learning systems in adversarial behavior.

2.8 CHAPTER SUMMARY

This chapter reviews the existing literature on distributed machine learning (DML), focusing on serverless computing, communication architectures, and fault tolerance. It discusses the benefits and challenges of scaling up machine learning systems using programmable GPUs and serverless runtimes. The chapter also evaluates different communication topologies and synchronization models, highlighting their impact on performance and scalability. Fault

tolerance techniques, including retry methods, checkpointing, and Byzantine fault tolerance, are examined to ensure system reliability. Lastly, a comparative analysis of Parameter Server (PS) and Peer-to-Peer (P2P) architectures is presented, considering metrics such as accuracy, throughput, scalability, and resilience to adversarial conditions.

CHAPTER III

SERVERLESS ON MACHINE LEARNING: A SYSTEMATIC MAPPING STUDY

3.1 CHAPTER OVERVIEW

Cloud computing is beneficial to businesses of all sizes in the marketing sector. It offers the abstraction of online services hosted on the cloud rather than complex local infrastructure. These services include everything from simple cloud storage to cloud infrastructure platforms. Cloud computing offers different benefits *i.e.,*high speed, efficiency and cost reduction, data security, scalability, back-up and data restore, control and level access, and unlimited storage capacity [117]. These services are offered in different proportions according to the provided service, such as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Software as a Service (SaaS), Function as a Service (FaaS) or Serverless.

Moreover, these platforms have grown significantly over the last decade and are widely adopted for the delivery of computing services. In particular, serverless computing provides a simplified architecture in which code execution is fully managed by the cloud provider , in such case, developers can focus only on code writing, increasing their productivity. Recently, serverless has been used as an infrastructure to build total ML pipelines or partially with faster deployment and elastic scalability.

On the one hand, serverless popularity is increasing, and it is receiving attention from developers, especially after Amazon launched AWS Lambda in November 2014 [2]. Recently, Wen *et al.*[19] found that questions about Serverless on StackOverflow have grown 380% from 2015 to 2020. The size of the serverless market is estimated to grow from 3.33 USD Billion in

[2]https://docs.aws.amazon.com/lambda/latest/dg/lambda-releases.html

2018 to USD 31.53 Billion in 2026 [118]. On the other hand, ML has been widely used in cloud computing, mainly when it is divided into a small pipeline stage (ML as a Service). The need becomes to use Serverless since the high cost of cloud resource management.

Thus, Serverless computing [119] is an interesting option regarding the resolution of small tasks, mainly when companies cannot estimate the traffic of their ML applications, scalability, and cost accurately [120]. Furthermore, several studies are exploiting serverless computing to accomplish tasks of the ML pipeline, such as training [121], hyperparameter tuning [122], and model deployment[20].

This chapter aims to map the current state-of-the-art to understand how Serverless was used in the machine learning pipeline and the challenges and opportunities for different stakeholders.

For achieving this **goal**, we perform a systematic mapping to answer three research questions by analyzing relevant studies. First, our study identified, classified, and evaluated the current state-of-the-art in machine learning on Serverless architecture. Next, we selected 50 primary studies from the Scopus database; then, we rigorously classified the studies to precisely categorize research results on ML and Serverless challenges.

The **audience** of this study is composed of both (i) researchers interested in contributing to this research area and (ii) practitioners interested in understanding existing research on machine learning applying Serverless architecture.

The main **contributions** of this study is to respond these research questions:

- What are the **publication trends** of research studies about serverless on machine learning?

- What is the **focus of research** of applied machine learning on Serverless computing ?
- What are the **potential challenges of adopting** machine learning on Serverless computing?

The rest of the chapter is organized as follows. The design of the study is presented in Section 3.2, whereas its results are elaborated in Sections 3.3. We have made a discussion in Section 4.5 where we broadened our perspective and the potential implications for both researchers and practitioners. Threats to validity is described in Sections 3.4. With Section 4.6, we close the chapter and discuss future work.

3.2 STUDY DESIGN

In this research, we follow the guidelines for systematic mapping studies [123]. We present the procedure to review the literature on machine learning usage on serverless architecture. In the following, we present the design of our study, including the search keywords, search technique, data sources, and inclusion and exclusion criteria are explained.

3.2.1 RESEARCH QUESTIONS

We set the following list of research questions as a guideline during the systematic mapping review:

*RQ1 - What are the **publication trends** of research studies about serverless on machine learning?*

By answering this research question, we aim to characterize the intensity of scientific interest in using machine learning on top of serverless architecture, the relevant venues where academics publish their results on the topic and their types of contribution over the years.

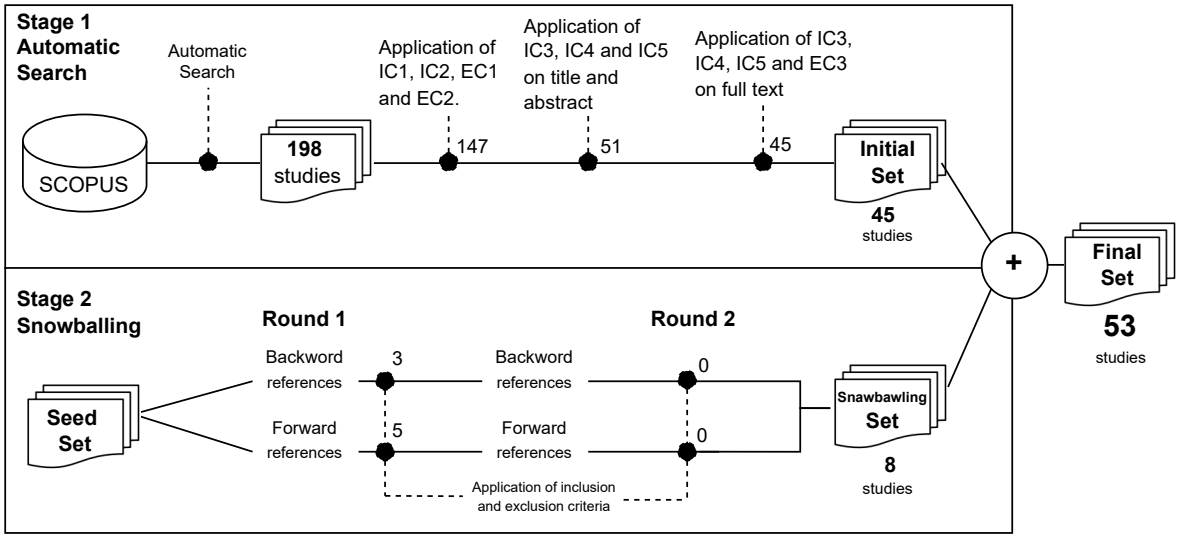


Figure 3.1 : Peer-reviewed selection process

*RQ2 - What is the **focus of research** of applied machine learning on Serverless computing ?*
By answering this research question, we aim to provide a solid foundation to classify existing research on machine learning in a serverless architecture.

*RQ3 - What are the **potential challenges of adopting** machine learning on Serverless computing?*
By answering this research question, our objective is to profile the state-of-the-art on challenges and opportunities to use machine learning on serverless architecture.

3.2.2 DOMAIN EXPLORATION

In the following, we describe the interesting domain covered by this research during the systematic mapping study.

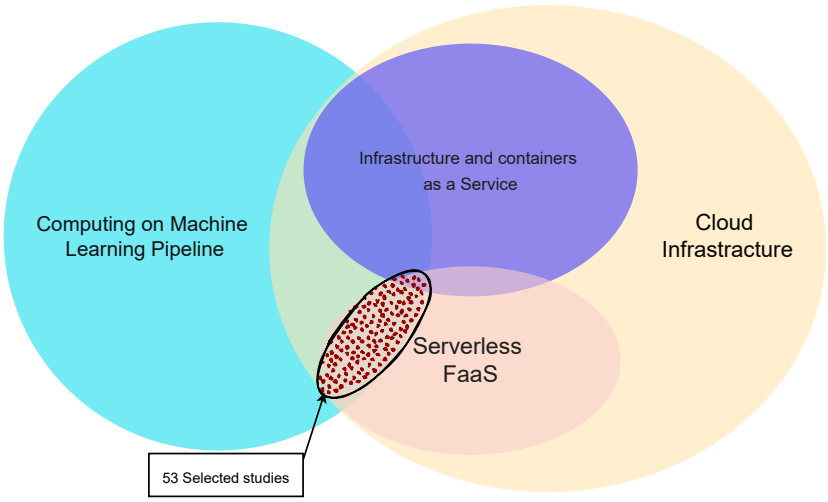


Figure 3.2 : Systematic Literature Mapping research focus

Cloud infrastructure: The types of cloud computing services vary, they provide access to IT infrastructure, hardware, and software resources. Cloud computing is all about delivering computing services like databases, software, analytics, servers, storage, networking, and intelligence. There are many benefits of cloud computing, including cost savings, scalability, and access to data centers around the world [124]. Cloud computing services fall into four main categories: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Software as a Service (SaaS), and Functions as a Service (FaaS) which is a relatively new Cloud service model.

Computing on machine learning pipeline: cloud services are a good option for anyone looking to train and deploy memory-intensive, complex Machine Learning/Deep Learning models. Cloud services are a cost-effective solution for both individual users and companies. The cloud allows employees to access files on any device [125].

Serverless on Machine Learning: Serverless architecture gives many opportunities and advantages to make the machine learning model more efficient and smoother [126]. As

shown in Figure 3.2, we focus on this systematic literature mapping on collecting research papers on applying serverless on machine learning.

3.2.3 SEARCH AND SELECTION PROCESS

As shown in Figure 3.1, we present our search and selection process. We designed a two-stage process, a systematic search similar to a previous study [127], to identify the current literature on serverless usage of machine learning. On Stage 1, we performed an automated search since it is the typical search strategy to identify relevant studies for a Systematic Mapping [128].

Defining the review goal, keywords were carefully selected to obtain relevant articles. In stage 1, several keywords were formulated and later narrowed down based on the research objectives. We designed our search query based on "machine learning" and "serverless." We executed the following search query on Scopus [3]:

```
("serverless" OR "lambda architecture" OR
"function as a service") AND
("machine learning" OR "deep learning")
```

Since we are looking for a particular subject, we applied the default automatic search, including the title, abstract, and keywords. We executed the query in June 2022, where we found **198 studies**. The papers were either included among the relevant articles or excluded as irrelevant for the review by studying their titles, abstracts, conclusions and complete content.

To extract only relevant articles for review, certain inclusion (IC) and exclusion (EC) criteria were set, specifically:

[3] https://www.scopus.com/

- IC1: The study must be an article, conference paper, or workshop;
- IC2: The study must be in the Computer Science area;
- IC3: The study must be a primary study;
- IC4: The study should address machine learning practices using serverless;
- EC1: The study is not written in English;
- EC2: The study is duplicate;
- EC3: The study is published as part of textbooks, abstracts, editorials, and keynote speeches.

After applying the IC1, IC2, EC1, EC2, we obtained **147 studies**. Then we analysed the title and abstract of each study and, after filtering by IC3, IC4 and EC3, we kept **51 studies**. We read all **51 papers**, filtering by IC3, IC4 and EC3 on the full text; we obtained the seed data set with **44 studies**. In Stage 2, we used our seed data set to perform two rounds of snowballing, backward and forward, detailed in 3.1. The snowballing research comes out with **8 additional studies**. Thus, we identify in the final dataset **52 studies** for this systematic mapping (Table 3.1). Our work is shredded on a public repository for study reproducibility [4].

3.3 RESULTS

The final set of publications presented in Table 3.1 was carefully read to answer the raised research questions. In the following, we are addressing carefully (1) the evolution trend of the set of papers and the different venues that were published; (2) the focus of the set of

[4]https://github.com/AmineBarrak/Serverless-on-ML

Table 3.1 : Summary of studies included in the systematic mapping

#	Reference	Title	Year
P01	[129]	A Case for Serverless Machine Learning	2018
P02	[21]	Exploring Serverless Computing for Neural Network Training	2018
P03	[130]	Implementation of unsupervised k-means clustering algorithm within amazon web services lambda	2018
P04	[131]	Pay-Per-Request Deployment of Neural Network Models Using Serverless Architectures	2018
P05	[132]	Serving deep learning models in a serverless platform	2018
P06	[133]	BARISTA: Efficient and scalable serverless serving system for deep learning prediction services	2019
P07	[134]	Behavior analysis using serverless machine learning	2019
P08	[68]	Cirrus: A Serverless Framework for End-To-end ML Workflows	2019
P09	[27]	Distributed Machine Learning with a Serverless Architecture	2019
P10	[135]	Function-as-a-Service Application Service Composition: Implications for a Natural Language Processing Application	2019
P11	[136]	Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving	2019
P12	[137]	On the FaaS track: Building stateful distributed applications with serverless architectures	2019
P13	[138]	Seneca: Fast and low cost hyperparameter search for machine learning models	2019
P14	[139]	Serving machine learning workloads in resource constrained environments: A serverless deployment example	2019
P15	[140]	Stratum: A serverless framework for the lifecycle management of machine learning-based data analytics tasks	2019
P16	[141]	Towards a Serverless Platform for Edge AI	2019
P17	[142]	TrIMS: Transparent and isolated model sharing for low latency deep learning inference in function-as-a-service	2019
P18	[143]	A cloud-based framework for machine learning workloads and applications	2020
P19	[144]	Automatic Tuning of Hyperparameters for Neural Networks in Serverless Cloud	2020
P20	[145]	Batch: Machine learning inference serving on serverless platforms with adaptive batching	2020
P21	[146]	Benchmarking Deep Neural Network Inference Performance on Serverless Environments With MLPerf	2020
P22	[147]	Enabling Cost-Effective, SLO-Aware Machine Learning Inference Serving on Public Cloud	2020
P23	[148]	FAASM: Lightweight isolation for efficient stateful serverless computing	2020
P24	[149]	Implications of Public Cloud Resource Heterogeneity for Inference Serving	2020
P25	[150]	Migrating Large Deep Learning Models to Serverless Architecture	2020
P26	[151]	Prognostics by classifying degradation stage on lambda architecture	2020
P27	[122]	Refactoring of Neural Network Models for Hyperparameter Optimization in Serverless Cloud	2020
P28	[152]	STOIC: Serverless Teleoperable Hybrid Cloud for Machine Learning Applications on Edge Device	2020
P29	[153]	Towards Federated Learning using FaaS Fabric	2020
P30	[121]	A Hybrid Framework for Effective Prediction of Online Streaming Data	2021
P31	[154]	A serverless gateway for event-driven machine learning inference in multiple clouds	2021
P32	[155]	AMPS-Inf: Automatic Model Partitioning for Serverless Inference with Cost Efficiency	2021
P33	[156]	Automatic Hyperparameter Optimization for Arbitrary Neural Networks in Serverless AWS Cloud	2021
P34	[157]	Cross-Platform Performance Evaluation of Stateful Serverless Workflows	2021
P35	[26]	Distributed double machine learning with a serverless architecture	2021
P36	[158]	Dorylus: Affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads	2021
P37	[159]	Edge-adaptable serverless acceleration for machine learning Internet of Things applications	2021
P38	[69]	Experience Paper: Towards enhancing cost efficiency in serverless machine learning training	2021
P39	[70]	FedLess: Secure and Scalable Federated Learning Using Serverless Computing	2021
P40	[160]	Gillis: Serving large neural networks in serverless functions with automatic model partitioning	2021
P41	[20]	High performance serverless architecture for deep learning workflows	2021
P42	[161]	Leveraging the serverless paradigm for realizing machine learning pipelines across the edge-cloud continuum	2021
P43	[162]	Performance and cost comparison of cloud services for deep learning workload	2021
P44	[163]	SLA-Aware Workload Scheduling Using Hybrid Cloud Services	2021
P45	[164]	Toward Sustainable Serverless Computing	2021
P46	[23]	Towards Demystifying Serverless Machine Learning Training	2021
P47	[165]	Towards situational awareness with multimodal streaming data fusion: Serverless computing approach	2021
P48	[166]	You Do Not Need a bigger boat: Recommendations at Reasonable Scale in a (Mostly) serverless and open stack	2021
P49	[43]	λDNN: Achieving Predictable Distributed DNN Training With Serverless Architectures	2021
P50	[167]	Serverless Computing Approach for Deploying Machine Learning Applications in Edge Layer	2022
P51	[38]	Stateful Serverless Computing with Crucial	2022
P52	[168]	INFless: A native serverless system for low-latency, high-Throughput inference	2022
P53	[44]	MLLess: Achieving Cost Efficiency in Serverless Machine Learning Training	2022

researchers on applying machine learning on Serverless architecture; (3) discussion of the challenges and opportunities to use Serverless on machine learning.

3.3.1 WHAT ARE THE PUBLICATION TRENDS OF RESEARCH STUDIES ABOUT SERVERLESS ON MACHINE LEARNING?

This research question aims at (1) characterizing the intensity of scientific interest and (2) the active publication venue on the usage of machine learning on serverless architecture.

3.3.1.1 PUBLICATION FREQUENCY

The selected papers of this study were analyzed to determine the trends in publication and the thematic evolution. Figure 3.3 shows the number of publications per year where researchers start exploring machine learning usage on serverless architecture. The results show that the average number of publications per year is approximately 12 from 2018 to 2021, starting with five papers in 2018 until 20 published papers in 2021.

Serverless computing has trended a significant engagement over the past years [19]. This boost has been caused by industry, academia, and developers for several reasons [169]. With the appearance of MLOps that include continuous and repetitive tasks (*i.e.,*code integration, training, deployment [170]), Serverless has started attracting ML developers.

3.3.1.2 PUBLICATION VENUE

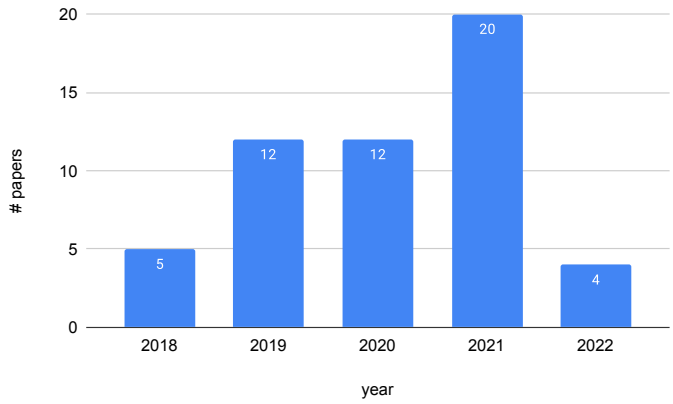


Figure 3.3 : Published papers per year

Researchers have been contributing on the usage of serverless on ML pipelines. Table 3.2 shows the various publication venues we find in the selected research papers.

Table 3.2 : Distribution of published papers by venue type

Venue type	#Studies	Studies
Conference papers	31	P02, P04, P05, P06, P07, P09, P11, P12, P13, P14, P15, P17, P19, P20, P23, P26, P27, P28, P32, P33, P35, P36, P38, P39, P40, P42, P43, P46, P48, P50 P52
Journal paper	10	P18, P21, P22, P30, P31, P37, P45, P49, P51, P53
Workshop paper	7	P01, P10, P16, P24, P29, P44, P47
Symposium paper	5	P03, P08, P25, P34, P41

The percentages of publications in conference papers, journal papers, workshop papers, and symposium papers are approximately 60% (31/53), 16% (10/53), 14% (7/53), and 10% (5/53), respectively. The topic Serverless for ML practices has started attracting more researchers, we found ten journal papers that were published which reveal the subject relevance where more studies can present additional contributions.

Table 3.3 : Journals' acronym-name mapping

Acronym	Journal Full Name
Concurr Comput	Concurrency and Computation: Practice and Experience
IEEE Access	IEEE Access
IEEE Software	IEEE Software
IEEE Trans. on Cloud Comput.	IEEE Transactions on Cloud Computing
IEEE Trans Comput	IEEE Transactions on Computers
J. Phys. Conf. Ser.	Journal of Physics: Conference Series
IEEE Internet Comput.	IEEE Internet Computing
Software Pract. Exper.	Journal of Software: Practice and Experience
ACM Trans. Softw. Eng. Methodol.	ACM Transactions on Software Engineering and Methodology
cs.DC	Distributed, Parallel, and Cluster Computing

Following the interpretation of publications, the most productive and primary journals, symposiums, conferences, and workshop venues related to serverless computing can be clarified. The list of journals we found is shown with their full names in Table 3.3. All eight journal venues were mentioned only once each.

The list of conferences is shown in Table 3.4. The "Cloud", "WOSC", "ICPE", "IC2E", "USENIX", "CCGRID", "Middleware", "ATC" and "Big Data" are considered the most active conferences held 19/42 (45%).

3.3.2 WHAT IS THE FOCUS OF RESEARCH OF APPLIED MACHINE LEARNING ON SERVERLESS COMPUTING ?

In this research question, our objective is to provide a solid classification of the existing research.

Table 3.4 : Conferences-venues mapping

Name	Occurrences
International Conference on Cloud Computing (CLOUD)	3
International Workshop on Serverless Computing (WoSC)	3
International Conference on Performance Engineering(ICPE)	3
International Conference on Cloud Engineering (IC2E)	2
International Middleware Conference (Middleware)	2
USENIX Annual Technical Conference (ATC)	2
International Conference on Big Data (Big Data)	2
International Symposium on Cluster, Cloud and Internet Computing (CCGRID)	2
Service-Oriented Computing and Applications (SOCA)	1
International Conference on Prognostics and Health Management (ICPHM)	1
International Conference on Information and Communication Systems (ICICS)	1
Innovation in Clouds, Internet and Networks and Workshops (ICIN)	1
International Conference on Parallel Processing (ICPP)	1
USENIX Operational Machine Learning (OpML)	1
USENIX Hot Topics in Edge Computing (HotEdge)	1
International Symposium on Software Reliability Engineering Workshops (ISSREW)	1
Pervasive Computing and Communications (PerCom)	1
International Conference for High Performance Computing, Networking, Storage and Analysis (SC)	1
North American Chapter of the Association for Computational Linguistics (NAACL)	1
International Conference on Distributed Computing Systems (ICDCS)	1
High Performance Serverless Computing (HiPS)	1
International Conference on Computing for Sustainable Global Development (INDIACom)	1
International Conference on Management of Data (SIGMOD)	1
Symposium on Cloud Computing (SoCC)	1
International Conference on Software Engineering Workshops (ICSEW)	1
Big Data in Emergent Distributed Environments (BiDEDE)	1
Conference on Recommender Systems (RecSys)	1
Neural Information Processing Systems (NeurIPS)	1
International Symposium on Workload Characterization (IISWC)	1
International Conference on Information Networking (ICOIN)	1
Operating Systems Design and Implementation (OSDI)	1
Architectural Support for Programming Languages and Operating Systems (ASPLOS)	1

Table 3.5 : Distribution of studies by Machine Learning pipeline stage

ML pipeline	#Studies	Studies
Data preprocessing	9	P01, P07, P08, P10, P25, P26, P34, P46, P48
Training / Learning	16	P01, P02, P03, P08, P09, P23, P29, P30, P36, P38, P39, P46, P49, P50, P51, P53
Hyperparameter Tuning	9	P01, P02, P08, P13, P19, P27, P33, P35, P46
Model Deployment (inference)	33	P04, P05, P06, P10, P11, P12, P14, P17, P18, P20, P21, P22, P23, P24, P25, P28, P30, P31, P32, P34, P35, P37, P40, P41, P41, P42, P43, P44, P45, P47, P50, P51, P52
End-to-end ML pipeline	6	P01, P08, P15, P16, P34, P42

3.3.2.1 RESEARCH STRATEGIES OF SERVERLESS USAGE ON ML PIPELINE

Machine Learning pipelines are composed of four main stages, (1) data processing, (2) model training, (3) Hyperparameter tuning, and (4) model deployment. We examine the papers to determine the main goal and the main solution each study proposes. As shown in Table 3.5, the most recurrent usage targeted by the primary studies are in deploying ML models on Serverless (33/53), followed by model training (16/53), hyperparameter tuning (9/53) and data preprocessing (9/53). There are (6/44) studies that tried to employ Serverless in the end-to-end ML pipeline. These results confirm that the use of serverless benefits in the different stages of ML is advantageous.

Table 3.6 : Distribution of studies by serverless platform usage

Serverless provider	#Studies	Studies
AWS Lambda	39	P01, P02, P03, P04, P05, P08, P09, P10, P11, P12, P13, P14, P19, P20, P21, P22, P24, P25, P26, P27, P31, P32, P33, P34, P35, P36, P37, P39, P40, P41, P43, P44, P45, P46, P47, P48, P49, P51, P52
Apache OpenWhisk	4	P18, P29, P39, P42
IBM Cloud Functions	4	P07, P38, P39, P53
Google Cloud Functions	3	P29, P39, P40
Azure Functions	3	P34, P39, P49
OpenFaaS	3	P29, P39, P52
Knative	2	P23, P50
KNIX	1	P40
Kubeless	1	P28

3.3.2.2 THE DIFFERENT SERVERLESS PROVIDERS

Table 3.6 presents the serverless platforms used in the considered research papers included in this study. It can be noticed that “AWS Lambda” has significant usage in 39 studies. We also found that “Apache OpenWhisk”, "IBM Cloud Function” and “Google Cloud Function" are used with four, four, and three published papers, respectively. Each platform has its own set of features and differs from others. We later compare the different providers in RQ3 4.5.

3.3.2.3 THE MAIN RESOLVED / DISCUSSED CHALLENGES AND ISSUES

The main solved / discussed challenges are cost / pricing (37/53) and resource scalability (30/53), as reported in Table 3.7. The high number of studies that discussed (1) cost/price and (2) scalability might indicate that Serverless provides a fair price architecture that provides a pay-per-use model that auto-scales in needs. Researchers seem to be interested in using Serverless for model deployment and make sure to keep a rational inference latency (22/53), in contracts (6/53), (2/53) and (2/53) focused on the storage, network, and training latencies, respectively. There were proposed solutions to reduce latency and improve performance by varying the batch size; this solution was present in (16/53). The cold start was discussed in (10/53) studies trying to mitigate it since Serverless containers have start-up latencies in the hundreds of milliseconds to several seconds, leading to the cold-start problem [171]. A significant number of studies (10/53) discussed the Service Level Objective (SLO). We mention that the SLO is an agreement set by a Serverless provider where there is the pre-defined service minimum response time [147]. Interestingly, few papers considered security and privacy with (4/53) and (4/53), respectively. However, only (2/53) paper mentioned the portability and reproducibility of the run-time environment.

3.3.2.4 MACHINE LEARNING FRAMEWORKS USED IN THE STUDIES

The ML frameworks helped the researchers to test their proposed solution easily, without understanding the underlying algorithms. Therefore, the choice of framework depends on the complexity of the targeted task. As reported in Table 3.8, the predominant ML frameworks are: Tensorflow (22/53), Keras (10/53) and MXNet (9/53). Indeed, other frameworks have been used in recent studies such as Pytorch (8/53) since it can be used for distributed training in parallel machines [23], Numpy (5/53) and OpenCV (3/53).

Table 3.7 : Distribution of studies by challenge & issue

ML & serverless Challenges and Issues	#Studies	Studies
Cost and pricing	37	P02, P03, P04, P05, P06, P08, P09, P10, P11, P12, P14, P15, P17, P20, P22, P23, P24, P25, P30, P31, P32, P33, P34, P35, P36, P38, P39, P40, P41, P43, P44, P46, P47, P49, P51, P52, P53
Resources scalability	30	P01, P04, P05, P08, P11, P12, P14, P17, P18, P19, P22, P23, P25, P27, P29, P31, P33, P34, P35, P36, P38, P39, P41, P43, P44, P46, P47, P51, P52, P53
Inference latency	22	P01, P04, P05, P06, P11, P15, P16, P17, P20, P21, P22, P23, P24, P25, P28, P30, P34, P40, P42, P47, P50, P52
Batching (varying batch size)	16	P03, P08, P09, P11, P17, P20, P25, P26, P28, P32, P37, P38, P41, P49, P52, P53
Cold Start	10	P05, P17, P21, P23, P25, P28, P34, P40, P41, P52
Service Level Objective (SLO)	10	P06, P11, P20, P22, P24, P32, P40, P44, P45, P52
Edge Computing	6	P15, P16, P28, P42, P45, P50
Storage Latency	6	P08, P12, P14, P26, P41, P51
Security	4	P23, P29, P33, P39
Privacy	4	P16, P29, P39, P50
Training Latency	2	P02, P34
End-to-end Latency	2	P37, P50
Network Latency	2	P17, P36
Portability	2	P18, P31

Table 3.8 : Distribution of studies by used machine learning framework

ML frameworks	#Studies	Studies
Tensorflow	22	P02, P08, P09, P11, P17, P19, P20, P21, P22, P23, P24, P25, P27, P28, P29, P32, P37, P39, P41, P49, P50, P52
Keras	10	P11, P19, P22, P27, P28, P32, P33, P37, P39, P42
MXNet	9	P05, P09, P11, P16, P17, P20, P22, P24, P40
Pytorch	8	P04, P14, P36, P38, P43, P44, P46, P53
Scikit-learn	6	P10, P28, P34, P35, P37, P42
Numpy	5	P10, P23, P35, P42, P43
PyWren	4	P01, P08, P12, P38
Spark ML	3	P08, P12, P26
OpenCV	3	P21, P25, P41
ONNX	2	P04, P14
Tesseract	2	P25, P41
Pandas	2	P35, P42
Bosen	2	P01, P08
Pillow [172]	1	P32
Caffe	1	P17
MNIST	1	P50

3.3.2.5 TYPE OF MACHINE LEARNING ALGORITHM USED TO TRAIN MODELS

The type of machine learning used to train the models depends on the research goals. Table 3.9 shows what type of machine learning was used. The results show that neural network models dominate the studies with (22/53). Neural network models are more challenging to be used, especially in distributed environment *i.e.,*distributed ML training [69]. Surprisingly, we found the use of supervised ML, such as logistic regression, random forest, and SVM, in (14/53) studies. These models are not resources costly in the training phase. Their usage is mostly for comparison purposes of the proposed architecture [137]. There are several

other machine learning algorithms. We mention ResNet (12/53) and Inception with different versions (8/53). These models are based on a conventional neural network used for intensive computing *i.e.,*image recognition [173].

3.3.3 WHAT ARE THE POTENTIAL CHALLENGES OF ADOPTING MACHINE LEARNING ON SERVERLESS COMPUTING?

In this research question, our objective is to profile state of the art on challenges of machine learning usage on serverless architecture.

3.3.3.1 SERVERLESS PROVIDERS

It is interesting to see Serverless providers evolving their services over the years. Carreira *et al.*[129] discussed about the Serverless capacity as they were not able to run Tensorflow [176] or Spark [52] functions on AWS lambda due to size limits (3GB RAM). Today the limit RAM size has increased to 10 GB for each serverless function [177]. We present in Table 3.10 the Serverless performance functionalities offered by the different providers we found in the primary studies. This table was filled in January 2022. We did not include Kubeless in the comparison table, since it is not an active project. In the previous RQ, we found that 77% of the studies (34/44) were using AWS Lambda. We can explain that result because this tool provides a high Random Access Memory allocation to reach 10Gb. Moreover, since the serverless function works only on demand, it has a timeout where the instance is shutdown after a timeout set by the provider. We can see that Amazon has the longest function timeout.

Table 3.9 : Distribution of studies by used Machine Learning model type

Types of ML models	#Studies	Studies
Neural Network (DNN, CNN, RNN, GNN)	22	P02, P04, P13, P14, P17, P18, P19, P21, P27, P28, P29, P32, P33, P36, P37, P40, P42, P43, P44, P45, P49, P50
Supervised ML (LR, RF, SVM)	14	P01, P06, P07, P08, P12, P26, P30, P31, P34, P35, P42, P46, P51, P53
ResNet	12	P05, P06, P11, P17, P20, P22, P24, P32, P40, P49, P46, P52
Inception	8	P06, P11, P17, P20, P22, P24, P32, P40
LSTM	7	P11, P14, P19, P22, P30, P39, P52
Stochastic Gradient Descent (SGD)	6	P01, P09, P23, P38, P46, P53
MobileNet	5	P20, P32, P49, P46, P52
squeezenet	3	P05, P17, P24
NLP	3	P04, P10, P21
K-means	4	P03, P12, P46, P51
VGG	3	P06, P33, P52
Reinforcement Learning (RL)	2	P09, P40
Federated Learning	2	P29, P39
NASNet	2	P11, P22
OpenNMT	2	P11, P22
Optical Character Recognition (OCR)	2	P25, P41
Yolo	2	P31, P47
Gym openAI	1	P09
Bert	1	P52

Table 3.10 : Comparative analysis of leading Function-as-a-Service (FaaS) providers

Serverless Provider	Function timeout (second)	Maximum Memory Allocation	Deployment package size
AWS Lambda [174]	900	10,240 MB	50MB zip 250 MB unzip
Apache OpenWhisk	600	2048 MB	48 MB
Google Cloud Functions	540	8560 MB	100MB zip 500MB unzip
IBM Cloud Functions	600	2048 MB	48 MB
OpenFaaS	300	42 MB	1MB
Azure Functions	600	1500 MB	n/a
KNIX[175]	30	2000 MB	100 MB
Knative	600	200 MB	256 MB unzip

We noticed that the deployment package size is small and differs from one provider to another. It is the total size allowed for the function source code *i.e.,*model. Providers offer the possibility of hosting the deployed model in extra storage if the model exceeds the limit for the serverless package size. For example, there is an option to use an external database or S3 bucket to store large payloads and pass the data identifier to the function calls. However, this option will cause additional latency to the system.

For better services, the serverless providers may ensure better user performance, especially the timeout function, to keep the instance warm and avoid the cold start latency.

3.3.3.2 SERVICE LEVEL AGREEMENT/OBJECTIVE

A Service Level Agreement/Objective is an agreement set by the serverless provider. Cloud providers claim different SLAs due to their unique technics. In such case, the perfor-

mance may vary for the same code from one cloud service provider to another [147]. We found that all the studies discussing the SLO agreement use the serverless for ML model deployment[155, 149, 133, 136, 147, 160, 145, 168]. They ensured that the inference model in their proposed solution was respected. For example, Amazon SLO regarding inference latency is that at least 98% of inference queries must be served in 200 ms. However, failing to acquiesce with the SLOs results will lead to compromised quality of service or even financial loss, e.g., end users will not be charged for queries not responded in time[178]. Regarding the machine learning models inference, the execution of small models (e.g., MNIST, Textcnn-69) can respond within 50ms under each memory configuration, but for the other large models, such as Bert-v1, ResNet-50 and VGGNet, a small memory configuration leads to quite a long execution time (exceeding hundreds of milliseconds). If configured with the maximum allowable memory size, the execution time for a single request exceeds 200ms, which makes it challenging to meet the latency SLO in the production environment [168]. Therefore, providers should share such agreements and statistics of service violations to help customers choose the best one, leading to a competitive environment for better services.

3.3.3.3 ENSURE RESOURCE SCALABILITY AND PREDICTIVE SCALING

In general, serverless architecture provides autoscaling features to handle workload spikes smoothly. Forecasting resource usage is no longer necessary to ensure that we always have the right amount of resources to host our applications. Compared with cluster computing, a serverless base model enables a rapid adjustment on-demand of the number of workers overtime [23]. Moreover, in multithread computation, a single-machine solution quickly

degrades when the number of threads exceeds the number of available cores, while in a serverless base solution, the scale-up is faster regarding the execution time [38].

However, serverless functions do not support customized scaling. Barista uses predictive scaling to achieve low-latency inference serving in the serverless cloud [133]. Moreover, a hybrid architecture between Serverless and VM is proposed, and even machine learning models to predict scaling to reduce over-provisioning to the best execution environment [136, 147].

The machine learning inference services are commonly latency critical, and the auto-scaling ability of serverless computing could deal with bursty workloads well[20]. Yang *et al.*[168] presented a solution called INFLess that reduces the allocation of resources for each serverless instance to reach optimal performance in inference services.

The resource scalability is a critical property of any ML training system since only the active workers at any given time will be billed. ML training is typically an iterative process in which a higher number of workers is desirable during the first training steps to diminish loss. When loss reduction stagnates and reaches convergence, the number of workers scales down once the learning curve starts to flatten out [69, 44].

3.3.3.4 SERVERLESS VS. IAAS FOR ML SERVING.

Several works in primary studies (37/53) developed the idea of reducing costs by using serverless in their ML solution (deployment, testing, etc.) instead of an infrastructure environment.

Serverless providers are proposing pay-as-you-go services, only paying for those resource usage - compared to other cloud resources such as the IaaS compute service AWS EC2,

where customers would be paying for the instance even when there is no traffic. Concretely, it is not all the time that Serverless is better than IaaS. When a company's traffic is known to deploy their model online, they must choose the service that performs their business model. If the traffic is unknown, it is better to use serverless since the payment is only for executions [179]. A hybrid architecture can be considered as an additional solution [136].

3.3.3.5 COLD START

Cold start in serverless is somehow expensive and causes significant performance degradation for serverless functions [180]. However, it is still better than other cloud resources *i.e.,*VM. There were several solutions for the cold start, for example, periodically warming the instance [160] or predicting the window timing where a request is expected, or scheduling the tasks [152]. To reuse against the cold start, in [135] presented a switchboard architecture composed of 6 serverless with the principle to warm the first function and trigger the rest of the functions. Another proposed to keep the instance warm for several minutes [181]. In [136], authors showed that keeping the instance warm has a low cost. They explained how $1 could spin up 7K inception-v3 Lambda instances, which can serve more than 20K requests per second. To avoid the cold start latency, Yang *et al.* [168] proposed a Long-Short Term Histogram (LSTH) to track application idle times and draw two histograms. The two histograms represent the request patterns in the last short (e.g., 1 hour) and long durations (e.g., one day). By tracking the application, they can select the pre-warming window to send inference requests to continuously keep the function instance alive. Their method helped to reduce resource waste while avoiding cold starts.

3.3.3.6 SECURITY AND PRIVACY

Privacy and security are always major concerns in serverless computing, especially for managing and analyzing sensitive data, such as healthcare data.

We observed that it is essential to set roles for every cloud function with specific security policies to provide only necessary access and prevent non-permitted operations. For example, Kaplunovich and Yesha [156] applied special protection to the hyperparameter metadata spreadsheet, where metadata is loaded directly during the startup and stored safely and securely in the protected Cloud location.

The Federated Learning-based architecture was proposed in the primary dataset [153, 70, 167]. This computing model supports edge computing, where the processing edges can learn from a shared machine learning model while keeping the model training on remote clients, followed by global aggregation of the updated model parameters. This keeps the training data local, which provides privacy and security benefits. Grafberger *et al.* [70] considers that the challenges of FL systems, such as scalability, complex infrastructure management, and wasted computing, can be solved with the Function-as-a-Service (FaaS) paradigm. However, it is necessary to be aware of the threats caused by malicious participants. For example, Tolpegin *et al.* [182] showed that a malicious subset of participants could decrease the accuracy of the model by injecting poisoned data when sending updates to the global model.

Several additional security measures can be applied, where only authorized and authenticated entities can invoke client functions. A practice of security between clients was applied in [70], where the FL server allows clients authenticated to read only from a shared global model and write back results without access to other clients. Another security measure was

applied in [70], where HTTP function requests exchanges can be encrypted using Transport Layer Security (TLS).

Rausch *et al.* [141] chose to transmit the base model to an edge device to refine the base model locally using a serverless function with the private data to ensure data privacy. The edge computing paradigm allows training distributed machine learning models between local edge data to secure data privacy and save resources in the cloud [167]. Bac *et al.* [167] applied a federated learning approach on serverless edge computing, where they saved bandwidth and ensured the data privacy of the edge nodes.

Moreover, Anthony S. Deese [130] handled the used access by applying AWS Cognito and Identity Access Management services. These services allow a user to access and monitor only the lambda function instances he created, which maintains the privacy of user training data and machine results.

3.3.3.7 BATCHING

Another essential factor that heavily impacts both the cost and the performance of ML serving inference is batching. For example, the batch size cannot be arbitrarily increased, as it leads to longer queuing latency and batch inference latency [183]. Tuning batch or resource configuration adaptively can improve the model performance. Clipper [183] introduces caching, batching, and adaptive model selection techniques to reduce the latency. INFaaS [184] automatically adapts the model variant batch size and hardware according to the required model performance.

For smaller batch sizes, the processing time increases linearly, but for larger batch sizes, it increases exponentially in an on-premise environment [20]. Deese [130] used the batch

mode (maximum size) read and write requests within AWS and found a significant speed increase from batch write operations, but a relatively small benefit from batch reads. Carreira *et al.* [68] finds that data fetching latency becomes low when applied mini-batches buffers. Wang *et al.* [27] considered that machine learning serverless functions should have a different size of data batch since many training samples need to be processed by different workers in parallel. Zhang *et al.* [136] showed that inference serving could benefit significantly from batching using costly hardware accelerators (e.g., GPU and TPU). The appropriate batch size with GPU instances can achieve a lower cost and shorter inference latency. However, serving inference queries using GPUs is not economically justified when there is not enough load. MLLess [44] kept the same mini-batch size in their distributed workers architecture to avoid changing the number of workers incurring costly data repartitioning transfers to adjust the mini-batch size.

Depending on the usage purpose of serverless computing in the machine learning pipeline phase, batching size can play an important role in reducing processing time, deployment latency, and data fetching.

3.3.3.8 SERVERLESS PORTABILITY

When using platform services from public cloud providers, there is a risk of dependence on the services and products they offer. This case is named the 'vendor lock-in' since switching technologies and vendors can be costly due to technical incompatibilities. Naranjo *et al.* [154] proposed to use open-source frameworks instead of public cloud providers. Most open-source serverless platforms rely on Kubernetes for orchestration and management of function pods, which makes the portability task more affordable [185]. Unfortunately, portability did not take

enough chances in the set of the studied papers. We found only two journal papers [143, 154] that took into account the level of portability of the run-time environment.

Portability is an important aspect; only when portability is ensured is a helpful simulation to test if the function shows better performance on another platform [169]. The Serverless Framework [186] offers plug-ins to simplify the deployment and execution of serverless functions across multiple clouds and FaaS environments. Junfeng Li *et al.* [185] compared the performance of four open-source serverless platforms using CloudLab testbed. Their work was provided to help developers to differentiate and select the appropriate serverless platform for different demands and scenarios.

3.3.3.9 EDGE COMPUTING

Edge computing is a distributed computing paradigm that brings computation and data storage closer to the data sources, especially popular with IoT device architecture. Edge computing has several benefits, such as reducing latency and bandwidth associated with public cloud [159], ensuring data privacy [141], and reducing computational resources relative to public and private clouds [152].

The serverless edge computing platform that provides the appropriate support to define AI workflow functions has been extended to work at the edge of the network to reduce response latency and bandwidth associated with the public cloud [141, 159, 167].

The ML module can be placed on the edge devices, or it can be placed on the Cloud or Fog layer for live or in-depth analysis of the data [140]. Zhang *et al.* [152] proposed a hybrid cloud system consisting of edge and cloud resources and integrating GPU acceleration.

The usage of edge computing depends on the user requirements and the analysis of available capacity.

3.3.3.10 COST REDUCTION

One of the primary purposes of using serverless with machine learning is cost reduction. High service costs are the major issue that papers try to reduce in different ways. Serverless usage is adopted to reduce unnecessary costs and improve manageability, like the allocation of virtual machines without full resource usage. For example, Wang *et al.*[27] demonstrated that a substantial amount of cost savings can be achieved by replacing dedicated IaaS cloud clusters with a serverless architecture. They proposed a solution called SIREN to reduce the training cost compared MXNet architecture. The AMPS-Inf achieves up to 98% cost savings without degrading response time performance [155]. Chahal *et al.*[163] presented an architecture based on load balancing the ML inference workload to reduce costs.

Cost reduction is a primary concern for developers and researchers. The cost is related to the design architecture, computing, inference deployment, and read/write queries. Depending on the machine learning project, a serverless-based architecture could be an effective option to reduce the cost.

3.3.3.11 INFERENCE LATENCY

Inference latency was well studied in the primary set of papers. Yu *et al.* [160] showed that the inference latency increases as the model grows. They proposed a serving model and generated a parallelization scheme deployed on serverless platforms to achieve optimal inference latency.

Zhang *et al.*[146] required a benchmark analysis to find an efficient inference workload. They found that the amount of memory allocated for each serverless function instance plays an important role in inference latency time reduction.

The latency can be influenced by the serverless cold start, or continued serverless warming [150]. Moreover, the hardware usage, such as GPU instances with the appropriate batch size, can have shorter inference latency compared to the CPU instances [136]. Gujarati *et al.* [187] proposed an autoscaling framework that aims to minimize resource waste for ML inference by using a predictive provision model. BATCH [145] designed a buffer layer on top of the serverless platform and bundles requests with batching for cost-saving serverless inference. Moreover, inference latency can be dominated by data fetching when there are queries involving cross-machine requests [131].

3.3.3.12 MLOPS AND SERVERLESS

The MLOps is modeled to make the intersection between machine learning, data engineering, and DevOps practices that associate software developers (the Devs) with IT operations teams (the Ops) to collaborate [188].

The machine learning pipeline contains several repetitive steps (data collection, data integration, data preparation and cleaning, model retraining, predictions) that need special operations to be automated in MLOps environments. The serverless architecture can be used to (1) automate the infrastructure; (2) build event-driven applications; (3) build APIs *i.e.,*API with Amazon gateway.

Serverless with data preprocessing: The serverless can be scheduled to pull data from the backend; trigger a serverless function when objects are written in data buckets *i.e.,*AWS S3; build APIs to transform and clean data.

Serverless with model retraining: Schedule or trigger new training when conditions are met.

Serverless with model inference: Schedule a serverless function for batch predictions; use step functions for ensemble predictions.

The serverless architecture can be feasible and optimal for projects adapting the MLOps approach. We plan as future work to explore how serverless can fit and optimise the MLOps-based projects.

3.4 THREATS TO VALIDITY

We applied Peterson guidelines to make our systematic mapping study [123]. However, threats to validity are unavoidable. This section presents the main threats to the validity of our study and how we mitigated them.

External validity. External validity relates to the generalizability of our results. The most severe external threat is that finding all the relevant studies on machine learning applied to serverless architecture from the designed query is absurd. As a solution, we applied a search strategy to the initial set of papers consisting of both automatic search and recursively backward-forward snowballing. Additionally, we applied a well-established peer-reviewed analysis to ensure that we have high-quality publications. We carefully defined the inclusion/exclusion rules that respect the requirements of our study with the agreement of all authors.

Internal validity. Internal validity relates to the experiment errors and biases. We mitigate the internal validity threats caused by author bias when selecting and interpreting data by applying well-assessed descriptive statistics of the collected data. Several re-verification steps between authors were performed to ensure a good classification dataset.

Construct validity. Construct validity is related to the degree to which an evaluation measures what it claims. We mitigated this potential bias by carefully defining the research query on the Scopus database. This database was preferred since it offers a more extensive list of modern sources [189]. In the keywording process, we included different taxonomies that can be mentioned to refer to the serverless, *i.e.,*lambda architecture, function as a service. Also, we are fairly confident about constructing the search string since the automatic search has been followed by snowballing. Also, we rigorously selected the potentially relevant studies according to well-documented inclusion and exclusion criteria. The first author performed this selection stage, and randomly a sample set was verified by the second author and agreement was ensured.

Conclusion validity. Conclusion validity is related to random variations and inappropriate use of statistics. To mitigate it, we rigorously defined and iteratively refined our classification framework, such as suggested by [190], so that we could reduce potential biases during the data extraction process. In addition, we ensured that we aligned with our research question and our main research objectives. We mitigated potential threats to conclusion validity by applying the verification agreement between authors in case of disambiguating cases. We provide a public repository for the reproducibility of the study to determine whether other researchers could obtain similar results from this study [5].

[5]https://github.com/AmineBarrak/Serverless-on-ML

3.5 CHAPTER SUMMARY

This study aims to provide a broader survey investigating the relationships among research contributions on Machine Learning usage on Serverless architecture. Specifically, we performed a systematic mapping on 50 primary studies and produced an overview of the state of the art on machine learning applications on serverless architecture. We found that (1) serverless usage on machine learning applications is a growing field starting from 5 on 2018 until 20 published papers on 2021, and more publication venues are interested to the subject; (2) serverless was adopted on the different ML pipeline, especially on ML model deployment with 33/53 papers. The most used serverless provider is usually AWS lambda, and the used ML model was the neural network. The main challenge of using serverless on ML was reducing cost and pricing (37/53), ensuring enough scalable resources (30/53), and reducing inference latency (22/53). There are several potential challenges of adopting ML on serverless, such as respecting the service level agreement, serverless provider, cold start problem, security and privacy, serverless portability, resource scalability, batch size, edge computing, cost reduction, and inference latency.

Depending on the targeted architecture and the solution, a trade-off between inference latency, serverless cold start, cost, scalability, batch size, and portability must be considered. For example, an open source provider would be a good solution if portability is essential. The results of this study will benefit both researchers willing to contribute further to the area and practitioners willing to understand existing research.

In future work, we plan to explore the effectiveness of serverless benefits with MLOps practices, especially in a distributed computing environment. Moreover, hybrid cloud architecture for machine learning pipeline phases can be a subject to study its validity depending on the user objectives and their data flow type.

CHAPTER IV

EXPLORING THE IMPACT OF SERVERLESS COMPUTING ON PEER TO PEER TRAINING MACHINE LEARNING

4.1 CHAPTER OVERVIEW

The exponential growth of data in the modern digital age [191] has transformed the landscape of artificial intelligence (AI) and machine learning (ML), propelling these fields into a new era of innovation and discovery. This vast deluge of data, has given rise to increasingly sophisticated and complex models that can extract valuable insights and make accurate predictions [31]. However, these sophisticated models pose a formidable challenge, due to the need for vast computational resources.

This escalating demand for computational power has led to the emergence of distributed training paradigm that seeks to address the limitations imposed by traditional, single-machine training approaches [192]. By harnessing the combined power of multiple devices, distributed training enables practitioners to tackle increasingly complex problems, expedite model development, and unlock the full potential of ML [193]. This training methodology encompasses the division of the dataset among a cohort of workers, each training their local model replicas in parallel and iteratively. To ensure convergence, the workers periodically synchronize their updated local models [45].

Various topologies have been proposed in the literature [31] to facilitate distributed training, including parameter server [59, 194] and peer-to-peer architectures [58, 195, 196, 197, 198] . In the parameter server architecture, the worker nodes perform computations on their respective data partitions and communicate with the parameter server to update the global model. In contrast, peer-to-peer (P2P) architectures distribute the model parameters

and computation across all nodes in the network, eliminating the need for a central coordinator. Each topology has its own unique set of advantages and disadvantages, making it suitable for diverse use cases and applications [31].

Regardless of the topology employed for distributed training, developers often struggle with managing resources and navigating the complexities of ML training. This can result in over-provisioning and diminished productivity, posing challenges for ML users striving to achieve optimal outcomes [25].

To address these challenges, building machine learning (ML) on top of serverless computing platforms has emerged as an attractive solution that offers efficient resource management and scaling [27, 68, 23, 140]. By automatically scheduling stateless functions, serverless computing eliminates the need for developers to focus on infrastructure management, allowing them to concentrate on ML model design and implementation [199, 3]. However, ML systems are not inherently compatible with the Function-as-a-Service (FaaS) model due to limitations such as statelessness, lack of function-to-function communication, and restricted execution duration [25, 44].

To overcome this and promote the adoption of serverless computing, numerous efforts have been made to optimize the utilization of FaaS platforms for managing ML pipelines[68, 38, 25, 70, 84, 69, 200]. A primary focus has been on the implementation of parameter server architectures. This architecture has demonstrated significant benefits when deployed in a serverless environment, including reduced costs [84], scalability [25, 70], and improved performance efficiency [200].

Notwithstanding these encouraging findings, there remains a dearth of research elucidating the ramifications of serverless computing on peer-to-peer architecture. **To the best of our**

knowledge, no research has been conducted to study the impact of serverless computing in a peer-to-peer environment.

Distributed training in peer-to-peer (P2P) networks offers benefits such as improved scalability and fault tolerance [30], but also presents challenges. As the network grows, communication, synchronization, and model update overheads increase, leading to latency and reduced training efficiency [45]. The diverse nature of devices in P2P networks can also cause imbalanced workloads and resource constraints, complicating the training process [201].

Another challenge faced during distributed training in P2P is the implementation of parallel batch processing inside each or the workers using popular machine learning frameworks like PyTorch. These frameworks often rely on the available and limited resources of individual workers to perform parallel computing on batches, which can lead to inefficiencies when resources are scarce [202, 203]. Consequently, these frameworks may resort to processing batches sequentially, which can result in longer training times and diminished performance.

In this chapter, we present a novel approach to address all these challenges associated with distributed training in P2P networks by integrating serverless computing for parallel gradient computation. Our approach consists of the following components: (a) Incorporating serverless computing into the P2P training process, which eliminates the need to expand the number of workers in the network, effectively reducing communication and synchronization overhead and consequently enhancing training efficiency. (b) Introducing an advanced technique that leverages serverless functions and workflows for parallel gradient computation within each worker, ensuring efficient and accelerated gradient computation for each peer in the network, even in the presence of resource constraints. Our work takes advantage of the dynamic resource allocation capability of serverless computing, which can adapt to real-time requirements and address the challenges posed by the increasing number of peers

and their varying capabilities. Additionally, the pay-as-you-go cost model of serverless computing enhances resource utilization by charging based on actual usage, resulting in a more cost-effective solution for distributed machine learning training.

Through a series of experiments and analyses [6], we demonstrate the effectiveness of our proposed approach in improving training, and optimizing resource utilization.

Our main contributions in this chapter include:

- *Propose a novel architecture that integrates serverless computing into P2P networks for distributed training.*
- *Introducing an advanced technique for efficient, parallel gradient computation within each peer, leveraging accumulative gradients even under resource constraints.*
- *Demonstrate the effectiveness of the proposed approach in improving training and optimizing resource utilization.*

4.2 METHODOLOGY AND SYSTEM DESIGN

In this section, we present a novel P2P training ML system based on Serverless computing, focusing on the design architecture, algorithm, and techniques to reduce peer overload. Our approach aims to improve efficiency, scalability, and alleviate resource constraints in ML training.

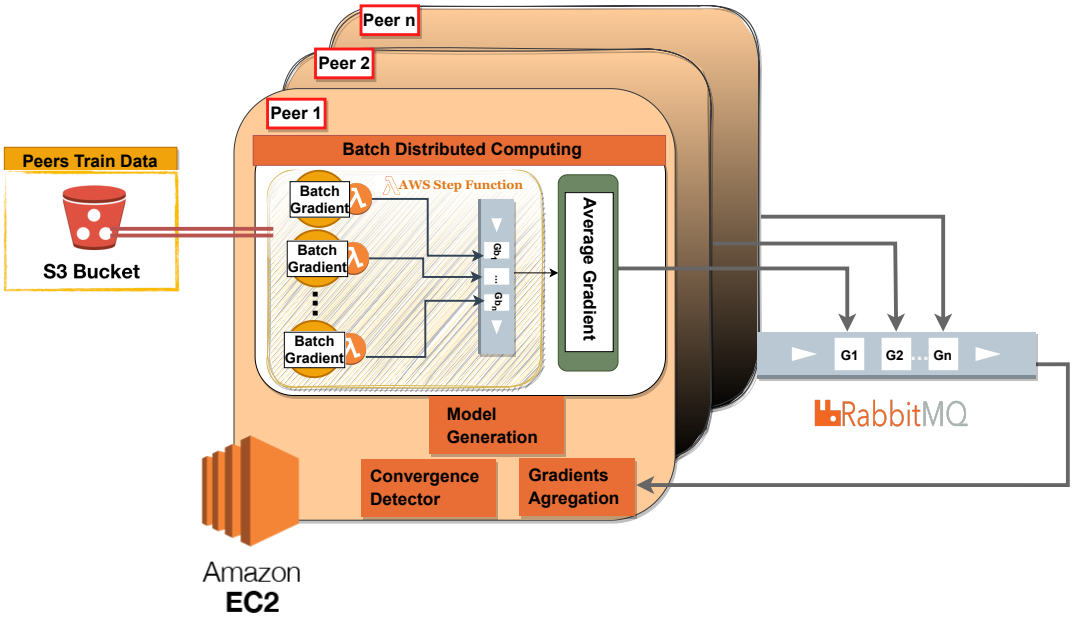


Figure 4.1 : Overview of the proposed Peer To Peer training based on Serverless computing

4.2.1 DESIGN ARCHITECTURE OF PEER TO PEER TRAINING MACHINE LEARNING BASED ON SERVERLESS COMPUTING

Figure 7.1 describes the overall proposed architecture. During the training of deep learning models, PyTorch strives to maximize resource utilization efficiently. However, ML frameworks *i.e.,*PyTorch, do not inherently possess a mechanism to seamlessly transition between parallel and sequential processing under resource constraints. In real-world scenarios, ML frameworks leverage a GPU for computations when available and default to the CPU when GPU resources are not accessible.

By harnessing the power of serverless computing, our system architecture enables parallel gradient computations across multiple Lambda functions, leading to a substantial reduction in overall computation time. We thoroughly examine the intricacies of our peer-to-

[6]https://github.com/AmineBarrak/PeerToPeerServerless

peer architecture, which consists of four integral system components. An overview of the peer to peer ML system based on Serverless computing architecture is depicted in Figure 7.1.

AWS S3 Buckets : In a peer-to-peer network, data is systematically partitioned into discrete segments, with each peer's assigned portion subsequently uploaded to a dedicated S3 bucket. This approach guarantees seamless access to their own data for each peer, while simultaneously leveraging the high-performance, cloud-based architecture of S3.

AWS Lambda Function : We strategically chose to implement parallel batch processing, a complex task made feasible by employing AWS Lambda serverless functions. By harnessing AWS Lambda's capabilities, we link each data batch to a specific Lambda function responsible for executing the necessary gradients computations. This approach significantly reduces total computation time through the wise distribution of workloads across multiple Lambda function instances, accelerating data processing and cutting down the time needed to complete processing the training set. Additionally, We integrate AWS Step Functions to manage, orchestrate and invoke the Lambda serverless parallel computing process, adapting to the availability of data batches and ensuring efficient handling of the workload.

EC2 Instance: Each EC2 instance in our system architecture, assigned to individual peers, carries multiple responsibilities. First, it acts as a trigger for invoking Lambda functions responsible for essential gradient computation. Additionally, it includes a crucial set of features that enable gradient exchange between peers. Ultimately, the EC2 instance is equipped with a specialized feature to detect model convergence, further boosting the overall efficiency of the system.

RabbitMQ : The proposed architecture relies on the utilization of RabbitMQ, that enable seamless communication between peers. After computing gradient averages over batches, a peer publishes the resultant data to its dedicated queue. Other peers in the network

can access the gradients published in the queue, enabling efficient and seamless information sharing. This is a critical aspect of our methodology, as it allows each peer to access the required information quickly and accurately to perform computations. RabbitMQ's reliability and security ensure smooth and secure data transmission and communication, promoting an efficient processing of complex data sets.

4.2.2 PEER TO PEER TRAINING MACHINE LEARNING

We specify a peer-to-peer architecture that leverages distributed computation for the purpose of training machine learning models. Algorithm 4.1 present the logic we followed. Initially, a workload is provided that includes the Deep Neural Network (DNN) model and the training dataset, along with parameters specifying the number of peers (P), batch size (B), and training epochs (E).

Additionally, each peer has an array of key-value pairs, where the key is the peer's rank (ID) and the value is the computed gradient.

We explain in the following the different sections of the algorithm.

4.2.2.1 DATASET PREPROCESSING

Within our system architecture, we have integrated a preprocessing stage to transform the training dataset using methods like min-max scaling, standardization, and normalization. After preprocessing, the dataset is divided into partitions for each peer in the training process.

Input: Deep Neural Network training workload with its **input train dataset** D **with size** n, **validation dataset** V, the **number of peers** P, **model size** d_m, the **learning rate** η, the **batch size** B, and the **number of epochs** E.

Output: The trained model with updated weights θ^*.

Peer of rank $r = 0, \ldots, P$:

▷ Each peer simultaneously implements:

Initialize: *Gradients_Peers*={ }
Initialize communication channels in RabbitMQ
Initialize a dedicated peer queue q_r
Initialize model parameters randomly as $\theta_0 \in \mathbb{R}^d$
for *epoch e = 1* ***to*** E **do**
    Load a unique partition of data D_r
    Randomly partition the subset D_r into m batches of size B
    for *each batch b* **do**
        $g_{t,b} \leftarrow$ **ComputeBatchGradients**(θ_{t-1})
    end
    AverageBatchesGradients as $g_{t,r} \leftarrow \frac{1}{m}\sum_{b=1}^{m} g_{t,b}$
    $Gradients\_Peers[r] \leftarrow g_{t,r}$
    SendGradientsToMyQueue$(g_{t,r}, q_r)$
    for *i from 0 to P* **do**
        if *i is not equal to r* **then**
            $g_{t,i} \leftarrow$**ConsumeGradientsFromQueue**(q_i)
            WaitUntilReceptionDone()
            $Gradients\_Peers[i] \leftarrow g_{t,i}$
        end
    end
    $g_t \leftarrow$ **AverageGradients**(*Gradients_Peers*)
     if *is_synchronous* **then**
        SynchronisationBarrier()
    end
    Update the model as $\theta_t \leftarrow \theta_{t-1} + \eta \cdot g_t$
    if ***DetectConvergence***(θ_t, V) **then**
        Return updated model θ^*.
    end
end

Algorithm 4.1: P2P ML Distributed Training

A dataloader is implemented to further split the partitions into batches, which are then stored in designated Amazon S3 cloud storage buckets.

4.2.2.2 COMPUTE BATCH GRADIENTS

The peer-to-peer training paradigm entails a multi-stage process wherein each worker subdivides its designated data subset into smaller batches, which are intended to expedite the training and convergence process by allowing each worker to compute gradients for smaller subsets of the data. During the training phase, each worker calculates the gradients for the batches of data it has processed and subsequently averages these gradients across all batches. This crucial step enables each worker to obtain an accurate representation of the gradients for its designated subset of data.

4.2.2.3 COMMUNICATION PROTOCOL

To communicate between peers, we used Amazon MQ's RabbitMQ for exchanging gradients between multiple peers during the model synchronization process. Each peer is assigned a dedicated queue that contains a single, persistent gradient message. When a new gradient is generated, it replaces the previous one in the queue, ensuring that the latest gradient is always available for consumption by other peers.

Peers can access and consume gradient messages from all other queues without deleting them, which promotes efficient gradient exchange and prevents data loss in case of temporary disruptions. The persistence of gradient messages guarantees the availability of the necessary information for model synchronization, even under challenging network conditions.

When peers are ready to synchronize their models, they read the gradient messages from all other queues, excluding their own. This process allows them to effectively update their models based on the gradients received from other peers, streamlining the distributed training process across the entire system.

To store received gradients from peers, a dictionary is created, where the peer's rank serves as the key to map to its corresponding received gradient. Each peer retains the received gradients in the local dictionary, and if the dictionary's size exceeds a threshold predefined in advance, the peer retrieves the gradients and calculates their average. The worker then updates its model parameters in accordance with the result. This iterative process continues for a predetermined number of epochs, as established by the input hyperparameters.

To overcome Amazon MQ's message size limitations (100MB per message), large files are stored in Amazon S3 and referenced using UUIDs. Sending UUIDs through Amazon MQ enables efficient, scalable data transfer without compromising performance or reliability, providing a flexible solution for seamless data exchange.

4.2.2.4 AVERAGE GRADIENTS

After receiving gradients from other peers, each peer aggregates the gradients by computing their average and uses this averaged gradient to update their local model parameters. The advantage of this approach is that it allows each peer to learn from the gradients computed by other peers, resulting in a more accurate representation of the global gradients.

4.2.2.5 SYNCHRONOUS & ASYNCHRONOUS GRADIENT COMPUTATION

In the following stage, the worker simultaneously distributes the averaged gradients to all other workers in the network and receives from them their averaged gradients as well. This process can be executed using either synchronous or asynchronous approaches. Figure 4.2 show an example of synchronous and asynchronous communication using four workers.

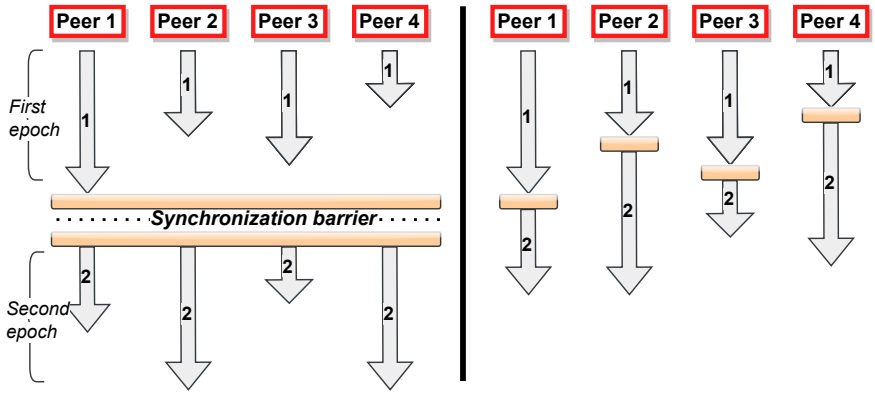


Figure 4.2 : Synchronous(left) and asynchronous(righ) Communication

In the asynchronous communication, Amazon MQ's RabbitMQ service provide a separate dedicated queues for each peer. These queues store the latest gradients generated by each peer, and they can be accessed and consumed by other peers without having to wait for every peer to finish their gradient computation. This means that a peer can start updating its model with the latest available gradients from other peers, without waiting for gradients from slower peers or those experiencing temporary disruptions.

In the synchronous communication, a synchronization barrier is added to ensure that all peers progress through the distributed training process together.

Synchronizing autonomous peers in a distributed system is challenging, especially when using RabbitMQ queues for gradient communication. Factors like varying resource availability can cause some peers to progress through epochs at different speeds. To address this issue, we have implemented a RabbitMQ-based synchronization mechanism. Each peer sends a message to a designated synchronization queue, signifying the completion of gradient computation, sending, and receiving for all connected peers. Once the size of this synchronization queue matches the total number of peers, it indicates that all peers have completed the current epochs, and they can then proceed to the next one in a coordinated manner.

4.2.2.6 CONVERGENCE DETECTION

To detect model convergence, two key techniques are used: *ReduceLROnPlateau* and *Early Stopping*. ReduceLROnPlateau adjusts the learning rate during training, improving generalization by preventing overshooting the loss function's minimum. It monitors model performance on a validation dataset, reducing the learning rate if improvement stalls.

Early stopping detects convergence by tracking performance during training and stopping when performance degrades, preventing overfitting. If convergence isn't reached through these techniques, the epoch limit determines the maximum training iterations.

Achieving convergence ensures the model's accuracy and effectiveness in making predictions.

4.2.2.7 MEMORY, CPU AND TIME METRICS COLLECTION

To assess and diagnose the efficiency of the system architecture, several Python libraries are used for recording performance metrics. Tracemalloc is utilized for measuring RAM

utilization, psutil for monitoring CPU usage in real-time, and the perf_counter function for evaluating time-based performance. These tools enable a deep understanding of system performance and identification of areas requiring optimization or improvement.

4.2.3 SERVERLESS COMPUTING TO REDUCE A PEER OVERLOAD COMPUTING

We leverage AWS Lambda for serverless parallel batch processing, enabling efficient workload distribution and reducing computation time. By assigning specific Lambda functions to data batches, we effectively manage gradients computations. AWS Step Functions orchestrate the Lambda functions, adapting to data batch availability for optimal workload handling. This serverless approach minimizes peer overload and accelerates training set processing, enhancing overall performance.

4.3 EXPERIMENTAL SETUP

4.3.1 DATASETS

MNIST: The MNIST Handwritten Digit Collection [204] consists of 60,000 samples of handwritten numerals, each categorized into one of ten classes.

CIFAR: The CIFAR Image Dataset [205] 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.

To ensure the model's accuracy was measured on unseen data, we split each dataset into training and test sets during the training process.

4.3.2 MODEL ARCHITECTURES AND HYPERPARAMETERS

SqueezeNet 1.1: SqueezeNet 1.1 is an efficient CNN architecture [206], with fewer parameters (1.2 million) and a small model size (<5MB).

MobileNet V3 Small: MobileNet V3 Small [207] is a lightweight CNN tailored for mobile and edge devices, featuring inverted residual blocks, linear bottlenecks, and squeeze-and-excitation modules. With approximately 2.5 million trainable parameters and a compact model size,

VGG-11: VGG-11 is a deep convolutional neural network (CNN) architecture developed for image classification tasks [208]. It is a variation of the VGG family, with 11 weight layers, including convolutional and fully connected layers. With an input resolution of 224x224 and approximately 132.9 million trainable parameters.

4.3.3 EC2 INSTANCES CONFIGURATION FOR PEERS

We aim to determine the ideal machine instance for three different neural network models: Vgg11, MobileNet V3 Small, and SqueezeNet 1.1. We started with the smallest available machine instance and trained the models on it. If the machine crashed due to resource limitations during training, we moved up to the next larger machine instance until we found one that was able to train the model without issue. Additionally, we incrementally increased the number of peers during the experimentation, starting with 4 and adding 4 peers at a time until we reached 12 peers, to determine the computation and communication resources usage. Ultimately, we determined that the Vgg11 model require t2.large instance, while the MobileNet V3 Small and SqueezeNet 1.1 models could be trained model could be trained on t2.medium instance. This approach allowed us to optimize the use of resources and achieve optimal performance for each model, taking into account both computation and cost.

4.3.4 SERVERLESS CLIENT FUNCTIONS CONFIGURATION

In the following, we discuss our approach to implementing a serverless training workflow by leveraging AWS Step Functions and Lambda functions for parallel gradient computation and batch processing.

4.3.4.1 SERVERLESS AWS LAMBDA CONFIGURATION FOR GRADIENT COMPUTATION

We prepared an AWS lambda serverless function for machine learning batch training. The function is designed to be invoked with essential parameters such as the specific model, batch identifier, optimizer, learning rate, and loss function. To obtain the necessary data batch for training, the function accesses an S3 bucket, where we have pre-processed and stored batches.

To streamline these Lambda functions, we implemented them using ARM architecture. This choice offers several advantages, notably their remarkable efficiency and lightweight characteristics. ARM processors are known for their energy-efficient design, allowing Lambda functions to run with minimal resource requirements. However, Lambda functions require libraries and dependencies to function, which posed challenges due to AWS Lambda's deployment package size limitations—50MB when compressed and 250MB when uncompressed. ARM-based layers inherently offer a reduced size, making them advantageous. To ensure seamless deployment on our custom ARM architecture, we packaged ML dependencies, including the PyTorch library, into a zip file. Additional dependencies can be integrated as separate layers within the AWS Lambda service, allowing for a modular structure while adhering to the service's constraints. This approach facilitates the development of efficient and scalable training processes within our ARM-based environment.

Additionally, we configured Lambda functions to access Amazon S3, granting them the necessary permissions to retrieve data from S3 buckets.

4.3.4.2 SERVERLESS AWS LAMBDA PRICING

One of the key factors in pricing for AWS Lambda is the amount of memory allocated to the function. The prices of AWS Lambda are calculated based on the amount of memory allocated to the function and the duration of the execution.

The objective is to compare the costs of running the same workload using EC2 peer to peer instances without serverless and using EC2 small instances by invoking serverless lambda for parallel compute gradients. This comparison will give us insights into the cost-effectiveness of using serverless computing in contrast to traditional computing methods.

Table 4.1 the prices of AWS Lambda memory charges per hour.

Table 4.1 : AWS Lambda memory pricing per second [1]

Memory (MB)	Price / 1 sec
128	$0.0000017
512	$0.0000067
1024	$0.0000133
1536	$0.0000200
1700	$0.0000220
1800	$0.0000233
2000	$0.0000259
2048	$0.0000267
2800	$0.0000362
3072	$0.0000400
3600	$0.0000466

4.3.4.3 DYNAMIC AWS STEP FUNCTION STATE MACHINE FOR PARALLEL BATCH PROCESSING

We have developed a Dynamic State Machine using AWS Step Functions, designed to compute parallel batch gradients on serverless Lambda functions. This state machine is generated dynamically according to the given batch number, allowing it to accommodate varying batch sizes. By leveraging the parallel computing capabilities of AWS Step Functions, each Lambda invocation processes an assigned batch saved in an S3 bucket. Once the state machine is deployed, it is invoked with the necessary input, which includes the total number of batches and the data required for the Lambda function to compute gradients corresponding to each data batch. This data encompasses the model, batch, optimizer, learning rate, and loss function. Our approach effectively enables parallel processing of gradient computations within a serverless environment using AWS Step Functions and Lambda functions.

Diverging from conventional training methodologies, where peers laboriously compute gradients batch by batch, our approach pioneers a paradigm shift, as depicted in Figure 4.3, empowering each peer to concurrently perform gradient calculations across multiple batches. By leveraging the concurrent abilities of AWS lambda, we link each data batch to a specific Lambda function, which takes on the responsibility of executing the essential gradient computations.

4.4 EXPERIMENTAL RESULTS

In this section, we present the results of our experiments, which is divided into four main subsections. We evaluate and analyze the performance of our proposed approach in various

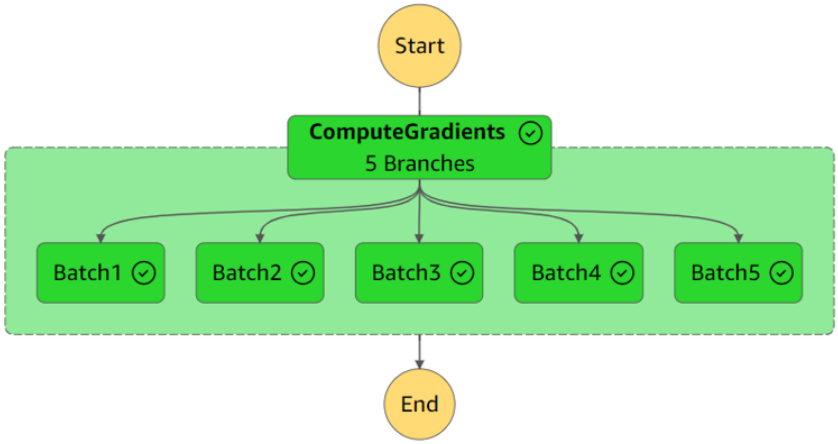


Figure 4.3 : Parallel gradient computation orchestrated by AWS Step Function

aspects related to distributed deep learning, such as computational resources requirements, serverless infrastructure efficiency, and synchronization barriers in peer-to-peer training.

4.4.1 IDENTIFY TASKS NEEDING EXPENSIVE COMPUTATIONAL LEVEL

To determine the resource usage and identify computationally expensive tasks in a distributed peer-to-peer training setup. In this setup, four worker nodes collaborate to train a machine learning model. The experiment focuses on measuring the resource usage at different stages of the distributed training process, including computing gradients, sending gradients, receiving gradients, updating the model, and convergence detection, is monitored and captured.

Metrics such as CPU usage, memory consumption, and processing time are recorded for each stage. The experiment continues to four epochs and the average per epoch is computed. Afterward, we compare resource consumption across different stages and identify the most computationally demanding tasks.

According to the results of our experimental investigation on three different models, namely VGG11, MobileNetV3 Small, and SqueezeNet, using two distinct datasets, MNIST

Table 4.2 : Evaluating resource usage in distributed Peer-to-Peer training with four workers and 30 batches

Model (instance type)	Dataset	Training Stage	Compute Gradients (per batch)	Send Gradients	Receive Gradients	Model Update	Convergence detection
squeezenet 1.1 (t2.medium)	MNIST / CIFAR	CPU Usage (%)	194,82 / 195,45	39,37 / 43,95	71,85 / 73,72	122,4 / 141,5	198,17 / 196,17
		Memory (MB)	600 / 570	568,29 / 566,64	555,32 / 569,91	566 / 530	574 / 540
		Processing Time (s)	14,93 / 14,01	0,084 / 0,08	0,25 / 0,27	0,18 / 0,0052	0,19 / 0,16
MobileNet V3 Small (t2.medium)	MNIST / CIFAR	CPU Usage (%)	197,87 / 198	40,45 / 40,92	75,3 / 81,42	147,6 / 134,8	198,05 / 198
		Memory (MB)	840 / 640	835,70 / 630,86	843,81 / 624,33	786 / 780	800 / 800
		Processing Time (s)	29,72 / 24,01	0,11 / 0,11	0,38 / 0,43	0,015 / 0,016	1,12 / 0,92
VGG 11 (t2.large)	MNIST / CIFAR	CPU Usage (%)	198,4 / 198,2	65,22 / 74	53,72 / 56	166 / 154	198,4 / 198,8
		Memory (GB)	4,10 / 4,24	3,19 / 3,075	4,8 / 4,33	2,4 / 2,46	2,41 / 2,4
		Processing Time (s)	104,37 / 104,20	7,38 / 6,72	15,55 / 19,54	4,8 / 4,2	9,20/7,6

and CIFAR, we have identified the most resource-intensive step during the training process. As demonstrated in the tabulated data of Table 4.2, the computation of gradients consumes a substantial amount of computational resources and memory, particularly for VGG11, which requires approximately 4 GB of memory per batch, and given that we executed 30 batches during our experiment. In comparison to other stages, such as sending and receiving data, updating models, and detecting convergence, the computation of gradients resulted in the highest CPU usage. As a result, it is reasonable to recommend the migration of the computation of gradients to a serverless infrastructure, which can reduce the overheads associated with managing and provisioning resources.

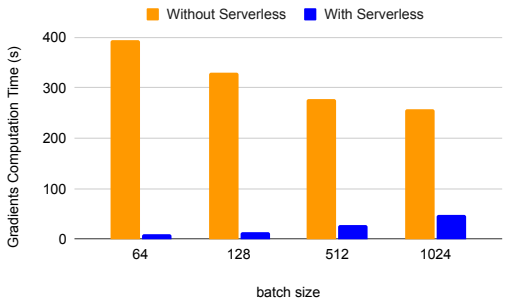
4.4.2 EVALUATION OF SERVERLESS INFRASTRUCTURE FOR GRADIENT COMPUTING

Throughout this section, we conducted a series of experiments to evaluate the impact of serverless infrastructure on the performance and cost of gradients computing. We evaluate

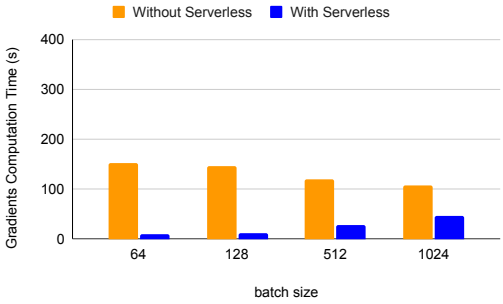
two distinct architectures to assess the impact of serverless integration on resource utilization and cost. In the first architecture, we train a VGG11 model and MNIST dataset with *t2.large* instances. In the second architecture, we train the same model with *t2.small* instances, while offloading high-computational tasks to a distributed lambda serverless infrastructure.

4.4.2.1 COMPUTATION TIME COMPARISON: SERVERLESS VS. INSTANCE-BASED ARCHITECTURES FOR GRADIENT COMPUTING

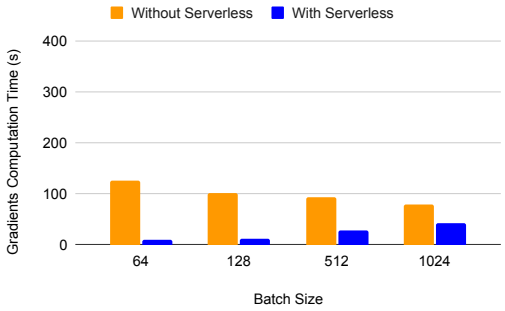
We examined different architectures, batch sizes, and numbers of workers to gain a comprehensive understanding of the potential benefits and challenges associated with serverless integration in terms of execution time of the gradients computation. The findings from our experiments are illustrated in a bar plot Figure 4.4 , where we have two bars for each batch size - one representing the time taken with serverless infrastructure (blue bar) and the other without serverless infrastructure (orange bar). This visual representation clearly highlights the significant improvements in the time taken to compute batches when employing serverless infrastructure across various batch sizes (64, 128, 512, and 1024) and numbers of workers (4, 8, and 12). For instance, in a configuration with 4 workers and a batch size of 64, the blue bar (serverless) is considerably shorter than the orange bar (non-serverless), demonstrating a remarkable 97.34% reduction in the time taken to compute batches. Similarly, with 8 workers and a batch size of 128, the improvement reaches 92.04%. However, it is worth noting that the improvement tends to decrease as the number of workers increases, especially for larger batch sizes.



(a) Training Time with Four Peers



(b) Training Time with Eight Peers



(c) Training Time with Twelve Peers

Figure 4.4 : Comparison of Processing Training Time on Gradient Computation for Different Numbers of Peers and Batch Sizes in Peer-to-Peer Training with and Without Serverless Architecture

4.4.2.2 COST COMPARISON: SERVERLESS VS. INSTANCE-BASED ARCHITECTURES FOR GRADIENT COMPUTING

In the previous experiment, we evaluated the impact of serverless infrastructure on computation time for gradient computing in peer-to-peer training, and our findings showed significant improvements across various batch sizes and numbers of workers. Notably, the case with four workers exhibited the highest improvement in terms of computation time, which prompted us to focus our attention on this specific scenario. To gain a deeper understanding of

the implications of these performance enhancements, we now turn our attention to investigating how these improvements may affect the cost of employing serverless architectures.

In the current analysis, we focus on the case study involving four workers. We compare the costs of employing serverless and instance-based architectures for gradient computing in peer-to-peer training, using VGG11 model and MNIST dataset with four peers. Tables 4.3 and 4.4 present the time and cost evaluation of compute gradients for different batch sizes with serverless and without serverless architectures, respectively. We determined the lambda memory size by adjusting it to meet minimal functional requirements for the gradients computation.

From Table 4.3, we observe that the time taken to compute gradients decreases with the decrease in batch size for the serverless architecture. This results in varying costs per batch size, with the lowest cost observed at a batch size of 64. In comparison, Table 4.4 presents the costs associated with the instance-based architecture, showing a clear increase in the costs as the batch size decreases. The cost differences between the two architectures can be attributed to the use of different instance types (t2-small, t2.large) and the varying memory size requirements for the lambda functions in the serverless architecture.

To analyze the trade-off between computation time improvement and cost, we compare the estimated compute gradients cost (USD) for both architectures across all batch sizes. For batch size 1024, we notice a cost decrease of approximately 75% when using the serverless architecture. However, as the batch size decreases, the cost difference between the two architectures also decreases. At a batch size of 64, the serverless architecture costs approximately 4 times more than the instance-based architecture.

The results highlight the importance of measuring the cost of employing serverless architectures for gradient computing, as there is a trade-off between the significant improvements

in computation time and the cost associated with using serverless infrastructure. It is essential for researchers and practitioners to consider their specific requirements, such as training time constraints and budget limitations, when selecting an architecture for gradient computing.

Table 4.3 : Time and cost evaluation of compute gradients in Peer-to-Peer training with Serverless; model trained on VGG11, MNIST dataset, and four peers

Batch Size	1024	512	128	64
Instance Type	t2-small	t2-small	t2-small	t2-small
Lambda Memory size	3600 MB	2800 MB	1800MB	1700MB
Time to Compute Gradients (seconds)	47,8	28,1	12,9	10,5
Estimated EC2 instance Cost (USD / seconds)	0.00000639$	0.00000639$	0.00000639$	0.00000639$
Estimated EC2 instance Cost (USD / seconds)	0.00002578$	0.00002578$	0.00002578$	0.00002578$
Estimated Compute Gradients Cost for 4 Peers (USD)	0.02659536$	0.028725888$	0.034085248$	0.040663776$

Table 4.4 : Time and cost evaluation of compute gradients in Peer-to-Peer training without Serverless; model trained on VGG11, MNIST dataset, and four peers

batch size	1024	512	128	64
Instance Type	t2-large	t2-large	t2-large	t2-large
Time to Compute Gradients (seconds)	258	278,4	330,4	394,8
Estimated EC2 instance Cost (USD / seconds)	0.00002578$	0.00002578$	0.00002578$	0.00002578$
Estimated Compute Gradients Cost for 4 Peers (USD)	0.00664884$	0.007181472$	0.008521312$	0.010165944$

4.4.3 PEER TO PEER TRAINING AND COMMUNICATION BARRIER SYNCHRONISATION

In our experiments, we aimed to compare the performance of two different peer-to-peer (P2P) approaches: synchronous P2P and asynchronous P2P. We conducted experiments on Mobilenet v3 small with a batch size of 64, a learning rate of 0.001, and the optimizer SGD. Our findings revealed that the synchronous P2P approach outperformed the asynchronous P2P approach in terms of convergence rate and achieved a higher accuracy level. Specifically, the synchronous P2P approach achieved an accuracy of 84.3% after approximately 128 epochs, while the asynchronous P2P approach required a greater number of epochs to converge and exhibited instability during the convergence. This was due to the asynchronous approach's tendency to consider outdated gradients, resulting in more epochs number to converge [23]. These results indicate that in P2P communication, synchronicity plays a crucial role in achieving a faster and more accurate convergence rate.

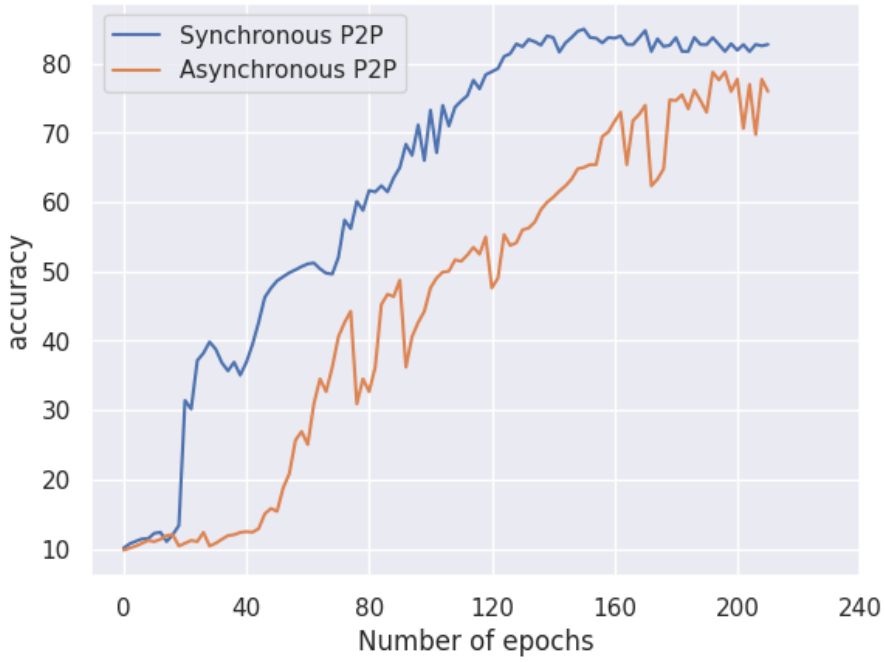


Figure 4.5 : Synchronous Vs Asynchronous Peer to Peer Training of MobileNet V3 Small

4.5 DISCUSSION

In this section, we reflect on the key findings and implications of our research on distributed deep learning in peer-to-peer training setups. We discuss the potential benefits and challenges of employing serverless infrastructure, the importance of reducing communication overhead, addressing synchronization barriers, and the impact of the choice of model architecture and dataset on the overall training performance.

4.5.1 BENEFITS AND CHALLENGES OF SERVERLESS INFRASTRUCTURE

Our experimental results have demonstrated that the computation of gradients is the most resource-intensive task in distributed peer-to-peer training setups. By migrating this task to serverless infrastructure, we observed significant improvements in computation time

across various batch sizes and numbers of workers. This highlights the potential of serverless infrastructure in providing an efficient and scalable solution for computationally expensive tasks in distributed training.

However, the use of serverless infrastructure also introduces cost implications. We analyzed the trade-off between the computation time improvements and the cost of employing serverless architectures for gradient computing. Our results indicated that, depending on the batch size and the specific requirements of the training process, the cost of using serverless infrastructure may not always be lower than traditional instance-based architectures. In some cases, such as smaller batch sizes, the serverless architecture could be more expensive. Therefore, researchers and practitioners need to carefully consider the specific requirements of their training processes, including training time constraints and budget limitations, when deciding whether to adopt serverless infrastructure.

4.5.2 IMPACT OF MODEL ARCHITECTURE AND DATASET CHOICES

Our experiments have shown that the choice of model architecture and dataset can have a significant impact on various aspects of distributed peer-to-peer training, including computational resource requirements, communication overhead, and synchronization barriers. Larger model architectures and more complex datasets generally require more computational resources, result in higher communication overhead, and demand longer synchronization times.

This highlights the importance of selecting appropriate model architectures and datasets for distributed training processes, considering the available resources and the desired trade-offs between training time, cost, and performance.

4.6 CHAPTER SUMMARY

In this chapter, we present a novel serverless peer-to-peer (P2P) architecture for distributed training, introducing an efficient parallel gradient computation technique using accumulative gradients to address resource constraints. We evaluated the performance of our approach by comparing it against a traditional EC2 VM setup, where execution was sequential. By parallelizing the computation and leveraging accumulative gradients, our serverless solution significantly improved efficiency. The evaluation focused on computational resource requirements, serverless infrastructure efficiency, and synchronization barriers. Our experimental results demonstrated that the computation of gradients is the most computationally expensive task, with the serverless infrastructure offering up to a 97.34% improvement in computation time compared to the sequential EC2 setup. We also explored the trade-off between these time improvements and the associated costs, revealing up to 75% cost savings with larger batch sizes in the serverless environment, while smaller batch sizes, such as 64, led to faster execution times but costs up to 5 times higher than the EC2 alternative.

The insights gleaned from this research can be utilized by other researchers and practitioners to build upon our work, further optimize distributed training processes, and potentially revolutionize the way machine learning models are trained across various applications.

CHAPTER V

REDUCING COMMUNICATION OVERHEAD IN SERVERLESS DISTRIBUTED TRAINING

5.1 CHAPTER OVERVIEW

A common trait amongst existing serverless architectures for distributed training is their heavy reliance on databases as a communication channel. This necessity arises from the stateless nature of serverless architectures and limitations in directly transmitting large data sizes. As a result, latency can arise, especially during the iterative model update and gradient aggregation process. In these architectures, model parameters and computed gradients need to be fetched from the database, processed, and re-uploaded in each iteration. This recurring cycle imposes a significant burden, resulting in additional computational and communication costs that can significantly impact overall efficiency.

Efforts such as LambdaML [23], SMLT [25], and MLLess [84] leverage databases for communication, collecting intermediate local statistics from workers through a defined pattern. However, the repetitive cycles of fetching and uploading data introduce significant overheads and latencies, negatively impacting training efficiency. In distributed serverless environments, the recurring database retrieval loads during training phases pose challenges. While these works utilize databases to store and retrieve model parameters, the resulting overhead remains largely unexplored. This often leads to increased communication overhead, undermining overall training performance.

Data compression techniques play a crucial role in mitigating communication overhead. By compressing model parameters and gradients before transmission, the amount of data that needs to be communicated is reduced, thereby decreasing latency and improving overall

efficiency. Compression algorithms such as quantization and sparsification have shown promise in reducing the size of transmitted data without significantly impacting model accuracy. Techniques like gradient compression [209], model sparsification [210], Delta compression [211], and communication optimization [212] have been proposed to minimize the data transfer during the training process, leading to a more efficient and cost-effective setup.

Other works propose in-database distributed machine learning solutions to reduce communication overhead [213]. Additionally, libraries such as RedisAI [214] enable seamless inference on trained models within Redis, and other solutions like OML4R [215] and MADlib [216] offer SQL-based functions for prediction and inference within database environments. E. Schüle et al. used automatic differentiation for a dedicated gradient descent operator, which generates LLVM code to train a user-specified model on GPUs. They fine-tune GPU kernels at the hardware level to allow higher throughput and propose non-blocking synchronization of multiple units [217].

However, within our unique architectural framework, we incorporate a customized Redis. This component allows us to perform average and update operations directly within Redis. Additionally, we explore the integration of compression techniques to further reduce communication overhead. In our experiment, we first explore the efficiency 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 and employing compression techniques. The insights derived from these experiments will then be compared with the traditional method of iterative fetch-update-store operations, typically used with standard Redis.

By examining and addressing these issues, the contributions of this chapter are as follows:

1. We propose a novel architectural framework that incorporates customized Redis to perform average and update operations directly within the database, integrating compression techniques to reduce communication overhead and improve training efficiency.

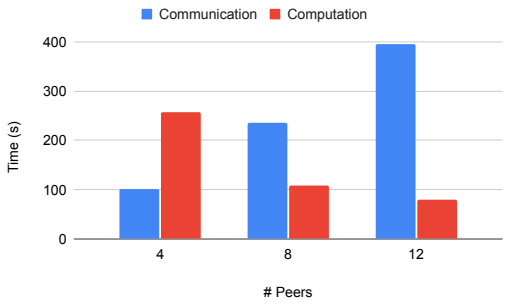
2. We present an empirical evaluation using two distinct models, MobileNetV3 Small and ResNet18, trained on the MNIST dataset, to assess the impact of model size on communication overhead and demonstrate the benefits of our approach compared to traditional iterative fetch-update-store operations.

5.2 COMPRESSION AND COMMUNICATION OVERHEAD

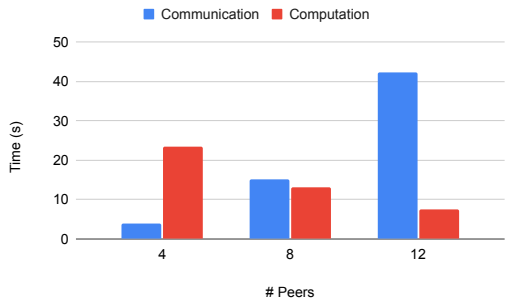
In this section, we will explore Compression and Communication Overhead in distributed deep learning systems. We will first analyze the impact of varying the number of workers on computation and communication overhead, followed by an investigation into Gradient Compression techniques for enhancing communication efficiency during the training process.

5.2.1 COMPUTATION AND COMMUNICATION OVER WORKERS

To elucidate the impact of communication overhead on system performance in a peer-to-peer architecture, we conducted rigorous experiments involving both VGG11 and MobileNet V3 Small models, varying the number of workers. In each experiment, we meticulously recorded both the compute time and the communication time. The results presented in the Figures 5.1 show the relationship between the number of workers (peers), communication time, and computation time for VGG11 and MobileNet V3 Small models when using a batch size of 1024. In both cases, the figures reveal that as the number of workers increases, computation time decreases while communication time increases. This can be attributed to the fact that with more workers, the dataset is divided among more devices, allowing for faster computation. We



(a) VGG11: Gradients Computation and Communication Time



(b) MobileNet V3 Small: Gradients Computation and Communication Time

Figure 5.1 : Comparison of Gradients Computation and Communication Time per Number of Peers for VGG11 and MobileNet V3 Small with a Batch Size of 1024. (a) Displays the Results for VGG11, while (b) Shows the Results for MobileNet V3 Small.

notice that the magnitude of the increase is much higher in the VGG11 model compared to the MobileNet V3 Small model. This could be due to the VGG11 model having a larger number of parameters, which results in more gradient information being communicated between workers.

5.2.2 COMPRESSION / DECOMPRESSION

Our system architecture has been formulated to effectively address the primary challenges faced in peer-to-peer architecture, which is the elevated communication overhead arising from the transmission of gradients among disparate peers. In this context, we have incorporated the QSGD algorithm [209], which capitalizes on a compression technique that quantizes gradients before transmission to reduce the size of the transmitted gradients. Consequently, the algorithm can considerably reduce the amount of information that necessitates transmission over the network, leading to a reduction in communication overhead and a consequent improvement in the efficiency of the training process. Upon computation of gradients by a peer, they are subjected to compression before being published in the queue.

5.2.3 GRADIENT COMPRESSION FOR COMMUNICATION IMPROVEMENT

As mentioned in the previous section, communication overhead increases as the number of workers increases. Gradient compression can be a solution to mitigate this. In order to assess the impact of gradient compression on communication overhead, an experimental investigation was executed using the VGG11 model, the MNIST dataset, and a network composed of four peers. Our paramount focus was on precisely measuring the send and receive times from a single peer, in order to comprehensively elucidate communication efficiency. As illustrated in Figure 5.2, we demonstrate that the utilization of gradient compression techniques yields a significant reduction in communication time when compared to the utilization of non-compressed gradients. For instance, with the batch size of 64, the send time reduces from 6.2 seconds to 4.57 seconds with compression, while the receive time decreases from 18.8 seconds to 12.06 seconds. This shows that, although reception typically takes longer than sending, both processes benefit from gradient compression, resulting in overall improved communication efficiency.

5.3 MACHINE LEARNING OPERATIONS WITHIN DATABASE

5.3.1 MOTIVATION

The integration of Redis and RedisAI into our machine learning training architecture aims to improve performance and efficiency. Redis, with its in-memory caching capabilities, accelerates the retrieval of frequently accessed data, boosting overall ML application performance. It offers low latency, high throughput, and versatility in storing various ML-related data, including serialized models and intermediate results.

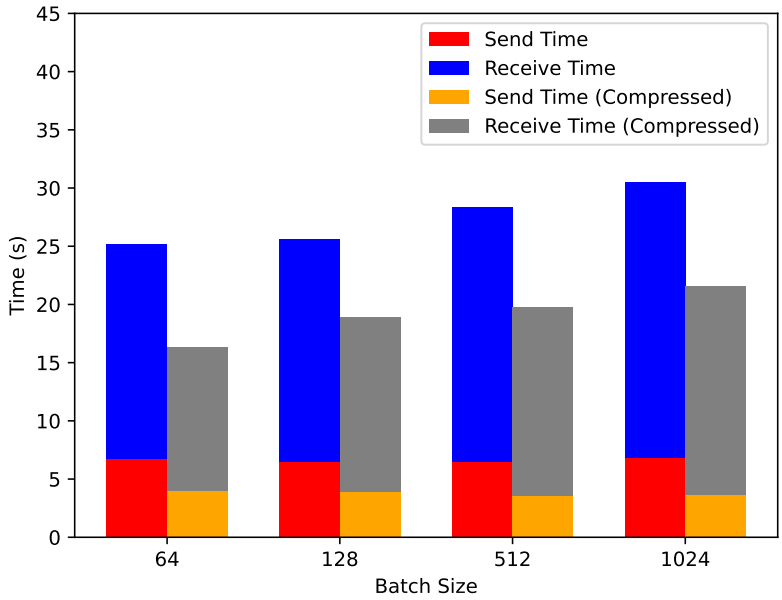
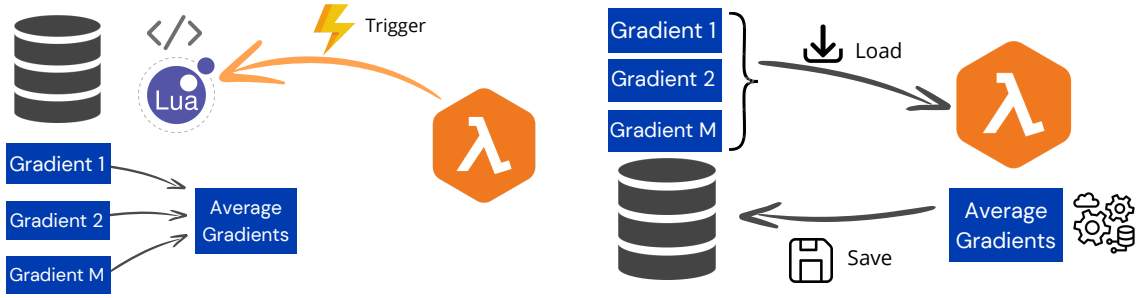


Figure 5.2 : Compression Algorithm impact on Time Communication (Send and Receive Gradients)

5.3.2 GRADIENTS AVERAGE WITHIN REDIS

Our approach focuses on improving the efficiency of machine learning workflows by calculating gradient averages directly within Redis. We achieve this by building a Lua script that is triggered within the database to perform the averaging. By centralizing this computation, we eliminate the repetitive cycles of fetching, updating, and storing data, significantly reducing data transfer latencies. In our experiments, we utilize two distinct models—MobileNetV3 Small and ResNet18—trained on the MNIST dataset to evaluate the impact of model size on overhead reduction. By comparing the results with traditional methods that perform gradient averaging outside the database, we demonstrate the advantages of in-database gradient averaging. Figure 5.3 showcases the gradient averaging within the database (before) and the gradient averaging outside of the database. This method provides efficiency gains and reduces

overhead, offering valuable insights into the performance improvements achievable through our approach.



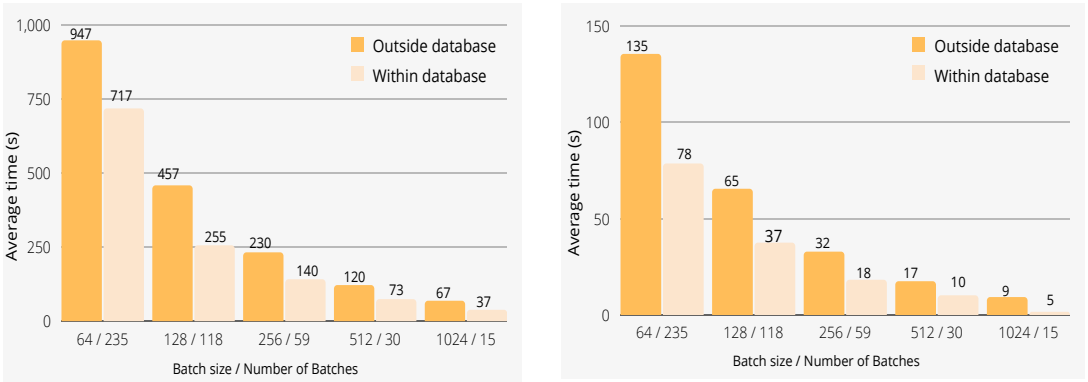
(a) Gradient averaging with DB (b) Gradient averaging outside DB

Figure 5.3 : Gradient averaging comparison

An in-depth analysis of the experimental results provides persuasive evidence of the significant efficiency improvements facilitated by in-database operations, as vividly depicted in Figure 5.4. It concisely illustrates the substantial reduction in time for gradient averaging calculations within the database, as opposed to outside, for both the MobileNetV3 Small and the ResNet-18 models. As the plot underscores, the in-database approach yields consistently lower computation times across the spectrum of batch sizes. For instance, the MobileNetV3 Small model experiences an impressive decrease in computation time from 135.29 seconds outside the database to 78.52 seconds within it for a batch size of 64. Using the largest batch size of 1024, a remarkable 82% improvement is achieved, requiring only 5.4 seconds for in-database computations.

A similar efficiency advantage is observed with the larger ResNet-18 model, where gradient 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 69% improvement in

processing efficiency, proving that our approach's benefits are applicable even for larger, more demanding models.



(a) MobileNet V3 Small Model (b) Resnet-18 Model

Figure 5.4 : Time taken for calculating gradient averages within and outside the database

5.3.3 MODEL UPDATES WITH REDISAI

The RedisAI module extends Redis' ML compatibility by enabling the execution of deep learning model inferencing and tensor operations within the Redis environment. This integration empowers Redis to directly serve ML models and perform computations, expanding its role in ML workflows. Figure 5.5 showcases the model update within the database (before) and the model update outside of the database (after) implementing the approach.

Our implementation strategically leverages the PyTorch backend, a fitting choice for our approach. The update of model parameters is orchestrated through the use of the stochastic gradient descent (SGD) optimizer.

We can use the new command in the Redis CLI as follows:

(a) Model update within DB (b) Model update outside of DB

Figure 5.5 : Model update comparison

```
AI.ModelUpdate <Params key> <Lr> <Grads key>
```

Arguments:

- Params key: The name of the key containing the model parameters.
- Lr: The learning rate value.
- Grads key: The name of the key containing the gradients.

We acquired the open-source code for RedisAI and integrated a new feature that facilitates in-database updates of models utilizing the PyTorch backend. After implementing this capability, we compiled the RedisAI module with the following steps:

1. Clear any previous builds: `make -C opt clean ALL=1`
2. Build the module: `make -C opt`

To integrate the RedisAI module when initiating the Redis server, we utilized the –loadmodule command line option and provided the accurate path to the module's library in the following manner:

```
redis-server --loadmodule ./bin/linux-x64-release/install-cpu/redisai.so
```

Our approach involved deploying the Redis instances on Amazon EC2 R6a.large machines, which are highly advantageous for handling memory-intensive tasks.

The results of this approach are shown in Figure 5.6 , which delves into the time taken for model updates within and outside the database. It highlights the significant reduction in model update times when processed within the database. MobileNetV3 Small model updates led to a remarkable 82% decrease in time, shrinking from 3.49 seconds outside the database to a mere 0.64 seconds within it. Similarly, ResNet-18 updates demonstrated an approximate 83% improvement, with update times dropping from 27.5 seconds outside the database to just 4.8 seconds within it.

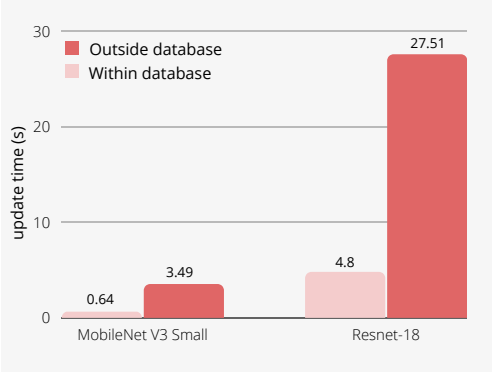


Figure 5.6 : Time taken for Model Update within and outside the database

5.4 CHAPTER SUMMARY

This chapter provides tools to reduce communication overhead in serverless distributed training systems. By understanding the trade-offs between computation and communication as the number of workers increases and using gradient compression techniques and in-database ML operations, we can improve the scalability and efficiency of ML workflows. Our findings highlight the potential of these strategies to create more scalable, efficient, and robust distributed deep learning systems.

Our findings suggest that employing gradient compression algorithms, such as QSGD, can reduce the P2P communication overhead by 27.42% for sending (from 6.2 seconds to 4.57 seconds) and 36.17% for receiving (from 18.8 seconds to 12.06 seconds) with a batch size of 64. This demonstrates the effectiveness of gradient compression in optimizing communication efficiency in distributed training scenarios.

Additionally, by leveraging RedisAI for in-database operations, we achieve a significant reduction in time for model updates and gradient averaging. The integration of in-database operations in RedisAI led to a substantial improvement of up to 82% in processing time, proving the efficacy of our architecture across different models and batch sizes. This highlights the potential of RedisAI to streamline and accelerate machine learning workflows within distributed training systems.

CHAPTER VI

SECURITY & FAULT TOLERANCE IN SERVERLESS DISTRIBUTED TRAINING

6.1 CHAPTER OVERVIEW

The integration of serverless computing into distributed machine learning (ML) optimization is gaining significant attention due to the cost-effective solutions offered by cloud platforms like Amazon Lambda [218], Google Cloud Functions [219], and Azure Functions [220]. While the parameter server (PS) topology has been extensively studied in distributed learning, the peer-to-peer (P2P) topology remains underexplored, particularly in the context of serverless computing. This oversight neglects P2P's inherent benefits, such as improved data privacy and fault tolerance over PS.

Despite the advantages of the P2P topology, it introduces critical security challenges, particularly concerning the integration and authentication of new peers [221]. Without stringent authentication protocols, unauthenticated peers could disrupt the training process. Moreover, even with robust authentication, the risk remains that compromised trusted peers might inject malicious updates, hindering model convergence and compromising the integrity of the training process.

Traditional aggregation of local updates in distributed learning typically involves straightforward averaging. However, this approach assumes all nodes are healthy and compliant, making it vulnerable to Byzantine behavior [222]. To counter this, our fault tolerance mechanisms ensure that the distributed ML system can continue training and converge to an optimal state despite the presence of malicious peers.

To address these challenges, we employ several strategies. Fault tolerance is bolstered through techniques such as Availability Zones, retries, architectural decisions, checkpointing mechanisms [223, 224, 102, 44, 23], and heartbeat techniques for failure detection [225]. To prevent data leakage, we secure communications within the distributed system [70, 197, 226, 10]. Additionally, robust aggregation techniques like KRUM [91], MULTI-KRUM [91], GeoMed [92], MarMed [92], and ZENO [227] are employed to mitigate Byzantine faults [222]. Recent adaptations of these rules to P2P architectures include the BRIDGE framework [111] and blockchain-based solutions [226].

Building on this existing work, our contributions include:

1. **Communication Security**: We implement secure peer initialization, authentication, and secure novel peer integration.
2. **Robustness Against Byzantine Attacks**: We evaluate our system's ability to maintain model convergence in the presence of malicious peers.
3. **Failure Mitigation**: We incorporate a heartbeat mechanism to address peer failures effectively and redistribute the data to the remaining peers.

In this chapter, we present detailed implementations and evaluations of these contributions, demonstrating their effectiveness in enhancing the security and fault tolerance of serverless distributed ML training.

6.2 SECURING COMMUNICATION AND INTEGRATION

Our system first addresses authentication, ensuring that only authorized and trusted peers gain access to the distributed training network. Secure communication within the network is achieved by employing RSA cryptography with a key size of 2048 bits, facilitating an

authentication process based on signature verification. All exchanges between peers are managed through Amazon Simple Queue Service (SQS) queues, where each peer has two distinct queues: one for join requests and the other for receiving passwords for other peers' databases. This mechanism encompasses various scenarios, from the initial setup of peers within the network to the integration of new peers, and it also details the management of private keys within our system.

6.2.1 PEERS INITIALIZATION AND AUTHENTICATION

This section explains the preliminary setup of the initial peers in our system. Since generating a signature requires an input message, we provide each peer with a shared secret. This secret is securely transmitted to the Lambda functions responsible for the authentication process, specifically the "Init Peers" and "Authenticate Peers" functions.

To explain the initial authentication procedure in a simple and clear manner, we provide Figure 6.1, which offers an illustrative diagram meticulously explaining the initialization of a two-peer system.

To explain the process even further, the following steps are performed to establish a state of mutual trust between the two peers:

1. In the initial phase, the administrator triggers the initialization of peers by supplying each peer with the URLs of **join request** SQS queues that are linked to the neighboring peers. Subsequently, each peer generates its own pair of keys, consisting of its private key and a corresponding public key.

2. In this step, each peer generates a digital signature based on its own private key and the shared secret that was provided. It then broadcasts it via other peer's respective **join**

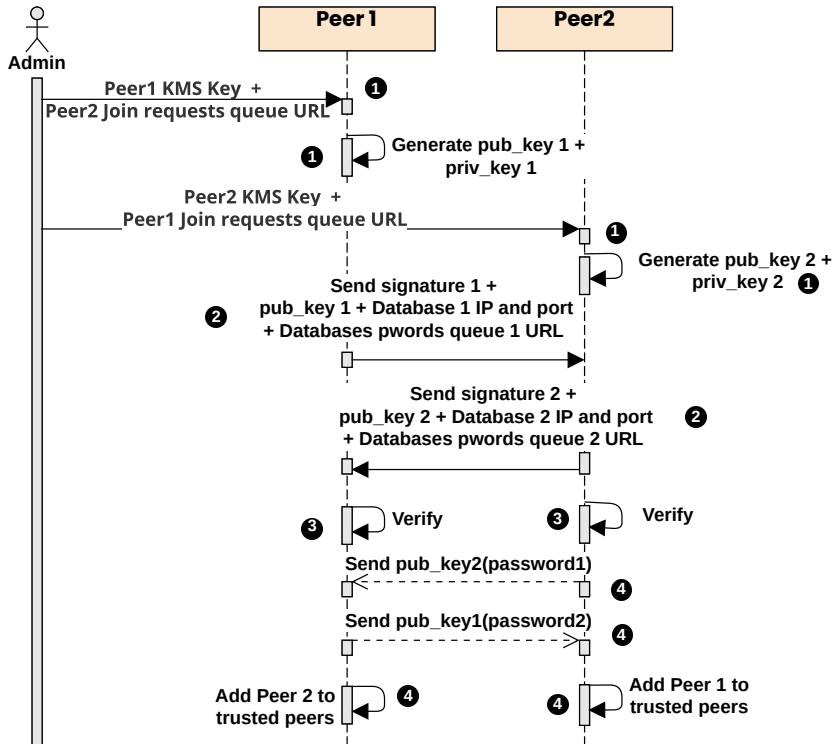


Figure 6.1 : Sequence Diagram Illustrating the Initialization Process of Peers

requests queues, along with its own public key, its database IP address and port as well and the URL of its queue responsible for receiving other databases passwords.

3. This step is dedicated for the verification where each peer verifies the signature of other peers: For instance, Peer 1 validates the signature of Peer 2, and conversely, Peer 2 also verifies the signature of Peer 1, as illustrated in Figure 6.1.

4. Once the verification was performed successfully, peers mutually exchange their encrypted database passwords to ensure a confidential exchange: Peer 1 encrypts its database password using Peer 2's public key to ensure only Peer 2 can read the original password from the encrypted password and vice versa. After this exchange, they proceed to saving each other details inside their respective databases into the list of trusted peers.

It is important to note that the encrypted passwords are saved as they are so only the corresponding peer can read it.

6.2.2 ADDING AND AUTHENTICATING NEW PEERS

This section clarifies the process by which a new peer integrates into the training network alongside pre-existing peers to prove its trustworthiness.

To explain this process, Figure 6.2 illustrates the case when a new peer (Peer 3) is introduced to the network. The following steps are performed:

1. At the start, the administrator initiates the new peer (Peer 3) by supplying it with the URLs of **join requests** SQS queues that are linked to existing peers inside the network along with their respective public keys. Peer 3 generates its own pair of key afterwards.

2. Peer 3 proceeds to the generation of its digital signature using its public key and the shared secret that was supplied then broadcasts it along with its public key, database IP address and port as well, the URL of its SQS queue responsible for receiving the passwords and its own database password encrypted with the recipient's public key. All of this is sent via **join requests** queues of other peers.

3. In this phase, Peer 3 awaits validation from the existing peers. These peers are responsible for validating the authenticity of the new peer by comparing its signature with the provided public key.

4. Once the verification phase was done successfully, Peer 1 and Peer 2 send their respective databases passwords encrypted using Peer 3's public key along with their respective signatures to Peer 3. Then, they proceed to saving the details of Peer 3 to their databases as a new trusted peer.

5. The final step is to have Peer 3 receive and validate the signatures. After the validation, Peer 3 proceeds to saving the details of Peer 1 and Peer 2 to its database as new trusted peers.

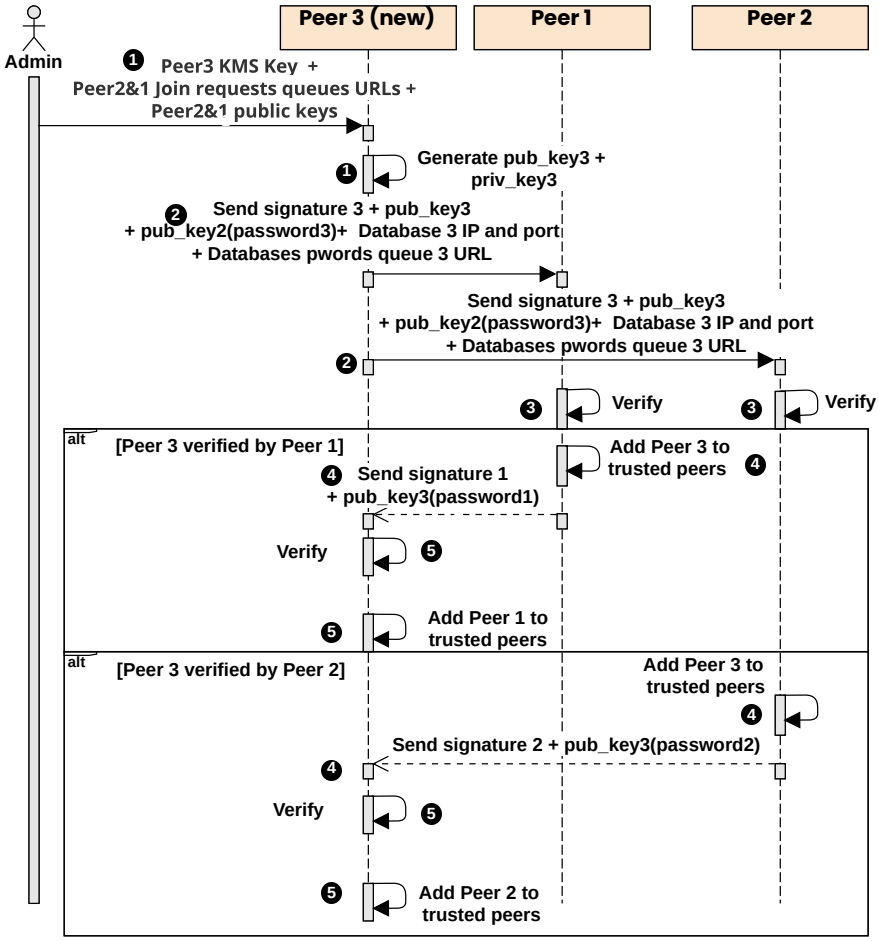


Figure 6.2 : Sequence Diagram of New Peer Integration into Training Network

6.2.3 PRIVATE KEYS MANAGEMENT ON CLOUD

Private keys constitute sensitive information that must not be disclosed or inadvertently exposed within the network. Therefore, storing them directly within a database is not considered a secure practice.

As each peer is required to retain its private key privately throughout the entire training process to be able to decrypt received passwords, concurrently addressing the statelessness of serverless computing, we supply each peer with its own unique encryption key from AWS Key Management Service (KMS) [228]. This key is a symmetric encryption key, generated and managed by KMS, which each peer employs to encrypt its respective private key before storing it within its dedicated database during the execution of **Init Peers** Lambda Function. From this point onward, everytime a peer needs to use its private key to decrypt the received passwords to access other peers' databases, it needs to decrypt its private key first using the same KMS encryption key that was provided to it.

In pursuit of heightened security measures, we implemented a stringent approach whereby access to the encryption key was strictly limited to the specific Lambda Functions that are both associated with the peer possessing the key and actively utilize it. In this manner, even in instances where an unauthorized entity gains access to a peer's database, it remains unable to decipher the associated private key.

6.3 SECURE MACHINE LEARNING TRAINING FROM BYZANTINE ATTACKS

In our context, Byzantine attack tolerance refers to the ability of a distributed ML system to maintain its fundamental functionality—specifically, the training of an ML model to ensure its convergence toward an optimal state—even in the presence of compromised malicious peers within the network. To ensure the distributed system is resilient to Byzantine attacks, we implement robust aggregation algorithms such as MeaMed and Zeno. These algorithms use the number of Byzantine peers within the network as an input parameter to filter out their local gradients during the aggregation process. As a result, the system can prevent attacks such as gradient poisoning, backdoor attacks, and evasion attacks, which target the integrity of the gradients and model performance. Consequently, the **Aggregate** Lambda function

follows a two-step procedural sequence: first, determining the number of Byzantine peers in the network, and second, applying the selected robust aggregation algorithms.

6.3.1 DETERMINING THE NUMBER OF BYZANTINE PEERS

Since gradients are represented as vectors, cosine similarity is a suitable metric for quantifying their similarity. Cosine similarity between two vectors A and B is computed as follow:

$$cosine\_similarity(A,B) = \frac{A \cdot B}{\|A\| \times \|B\|} \tag{6.1}$$

Here, $A \cdot B$ represents the dot product of vectors A and B, and ||A|| and ||B|| are their respective norms. Cosine similarity yields a value between -1 and 1. A cosine similarity of 1 indicates that the vectors are in the same direction (maximum similarity), 0 implies they are orthogonal, and -1 signifies they are in opposite directions (maximum dissimilarity), as illustrated in Figure 6.3.

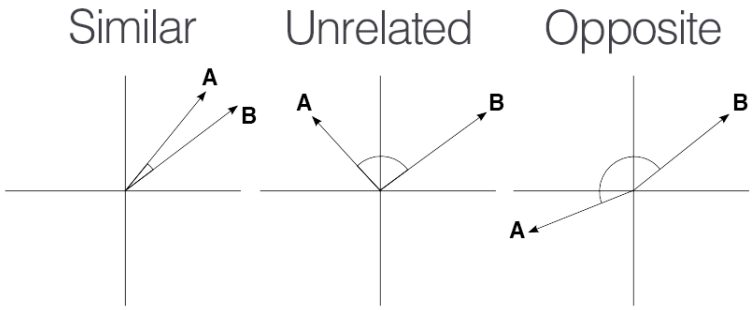


Figure 6.3 : Illustration of the application of cosine similarity between two vectors A and B.

Upon reaching the **Aggregate** Lambda Function, each peer in the network has successfully computed its local gradient and stored it in its respective Redis database. This is ensured by the **Synchronize** Lambda Function, which uses a shared synchronization queue where each peer sends a notification message upon task completion. After completing the authentication phase and reaching this stage, each authenticated peer can access the databases

of other peers to retrieve their stored local gradients, facilitating seamless gradient exchange within the network. The communication flow will be detailed in the next chapter.

Before detailing the process of determining the number of Byzantine peers, it is important to note that in P2P distributed training, each peer concurrently acts as both a worker and a server. This allows each peer to use its local gradient as a reference point for comparison. By employing the cosine similarity metric, each peer compares its local gradient with those of other peers to identify outliers, deemed malicious. Specifically, if the cosine similarity between the reference gradient and another gradient exceeds a threshold (0.6 in our case), the gradient is considered healthy; otherwise, it is deemed an outlier and thus malicious, as illustrated in Figure 6.4. This methodology enables the identification of the number of malicious peers within the network.

The use of a similarity metric to compare gradients for determining the number of Byzantine peers is based on the observation that healthy gradients exhibit similarity among themselves, whereas malicious gradients stand out as outliers.

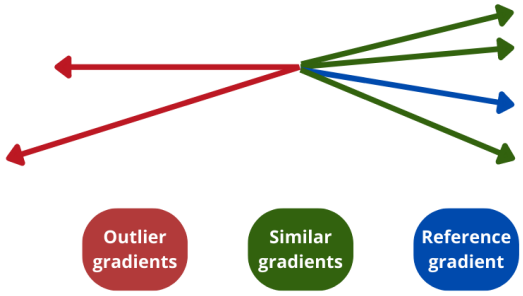


Figure 6.4 : Cosine similarity application to determine the number of malicious peers.

6.3.2 APPLYING ROBUST AGGREGATION ALGORITHMS

With the determination of the number of Byzantine peers achieved in the preceding step, we implemented two robust aggregation algorithms Zeno [193] and Meamed [91].

In each epoch, the AWS Step Function begins by accepting the aggregation algorithm to be applied during the training, encapsulated within a JSON format, along with other essential parameters required for the training process. This input is then conveyed to the **Aggregate** Lambda Function, which undertakes the responsibility of applying the chosen algorithm. It is important to emphasize that only a single robust aggregation algorithm is used throughout the entire training epoch.

6.3.2.1 ZENO ROBUST AGGREGATION

The initial algorithm we selected, known as Zeno [193], is founded upon the comparison of all local gradients one by one against a validation batch. This approach necessitates the presence of a Validation Set. Considering that each peer operates analogously to an independent Parameter Server and conducts local aggregation within the confines of its **Aggregation** Lambda Function, we have accordingly supplied a dedicated Validation Set for each peer. Given the consideration of the statelessness inherent in serverless functions, as well as the AWS Step Function, the preservation of Validation data within these entities becomes unfeasible. Hence, the Validation Set is recurrently loaded during each epoch via the Lambda Function tasked with the aggregation process. This is achieved by having each peer extract its stored validation data from its designated Validation Set stored in advance within S3 buckets. Figure 6.5 illustrates the process of validation data loading executed by peers. For the sake of clarity and simplicity, only the **Aggregate** Lambda Function has been referenced in the illustration.

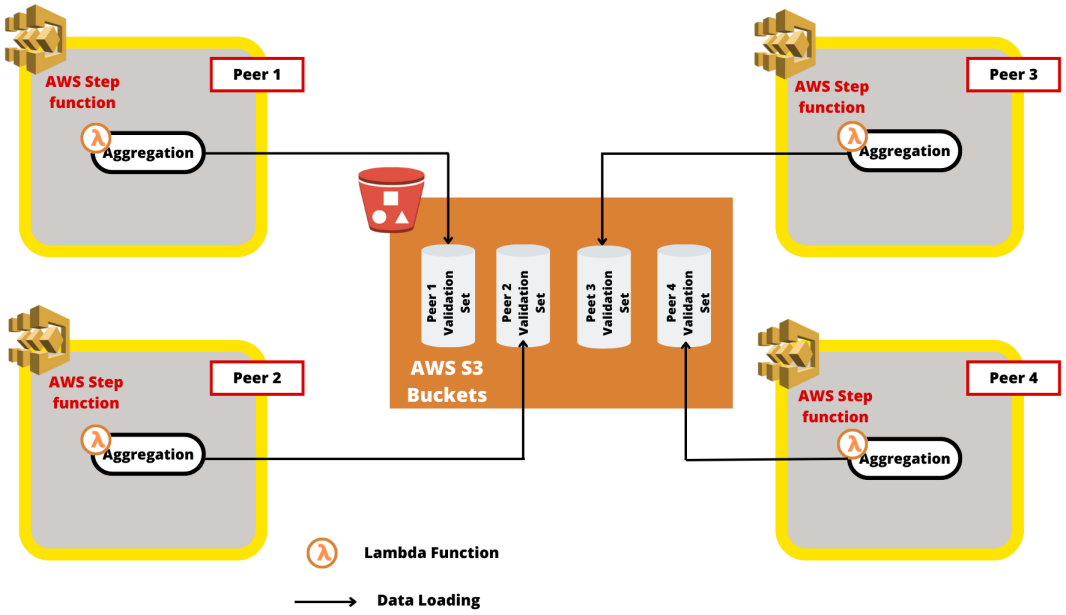


Figure 6.5 : Simplified explanation of validation data loading process from S3.

In the Zeno algorithm, each gradient receives a Zeno score based on updating the ML model with the corresponding gradient to filter outliers. This process is resource-intensive due to the loading of validation data and iterative model updates, potentially leading to memory exhaustion or timeout issues in serverless environments.

To mitigate this, we use the Redis database for model updates. The **Aggregate** Lambda Function calls a RedisAI [229] method to update the model and store the result in the database associated with the invoking peer. RedisAI is a module designed for executing and managing Deep Learning and Machine Learning models within the Redis data store. Detailed information about RedisAI is provided in chapter 5.

6.3.2.2 MEAMED ROBUST AGGREGATION

As for the second algorithm Median Absolute Deviation-based, it does not go through complex operations like in the case of Zeno.

The MEAMED algorithm is designed to achieve robust aggregation in distributed machine learning systems, particularly in the presence of outliers or adversarial data. It leverages the properties of medians and the Median Absolute Deviation (MAD) to ensure the aggregation process remains resilient to faulty or malicious updates. The MEAMED algorithm follows these steps:

1. **Collect Updates**: Gather updates (e.g., gradients or model parameters) from all workers.
2. **Calculate the Median**: For each dimension of the update vectors, compute the median of the values from all workers.
3. **Compute Deviations**: Calculate the absolute deviation of each worker's update from the median.
4. **Median Absolute Deviation (MAD)**: Compute the MAD for each dimension.
5. **Thresholding**: Determine a threshold to identify and exclude outliers.
6. **Exclude Outliers**: Exclude values from workers whose deviation from the median exceeds the threshold.
7. **Compute Robust Mean**: Calculate the mean of the remaining values for each dimension.

By employing the Meamed algorithm, the aggregation process becomes robust against outliers, ensuring that the learning process remains reliable even in the presence of adversarial or faulty updates.

6.3.2.3 EVALUATION AND RESULTS OF SECURING ML FROM BYZANTINE ATTACKS

Each time, we make use of one and only aggregation algorithm during the whole training process. To assess the performance of the selected aggregation algorithms, different scenarios are tested, and their respective testing accuracies per epoch are utilized as the evaluation metric:

- **No attack:** In this particular case, we conduct the training process in the absence of any adversarial attacks, ensuring the network remains healthy and devoid of any intentional disruptions. The trim parameter for Zeno is set to 1 and its regularization weight is configured to be equal to 0.0005. As for its validation batch its size is set to 10.
- **Under attack:** We tested the training with the presence of one Byzantine malicious peer out of four. The results were compared with those from averaging-based aggregation with no attack. For these experiments, the trim parameter for Zeno was set to the number of malicious peers, the validation batch size to 10, and the regularization weight to 0.0005. Each attack was tested individually, with the number of malicious peers constant throughout the training process. The attacks conducted in our experiments are as follows:
    1. **Noise attack:** Each malicious peer adds a random Gaussian noise with mean equal to 0 and a standard deviation equal to 200 to its local gradient.

2. **Sign-flipping attack:** Each malicious peer multiplies its local gradient by -100 before sharing it.

3. **Label-flipping attack:** Given that the MNIST dataset encompasses 10 labels ranging from 0 to 9, for each malicious peer, the process for determining the flipped label is as follows: denoted as **"9-label"**.

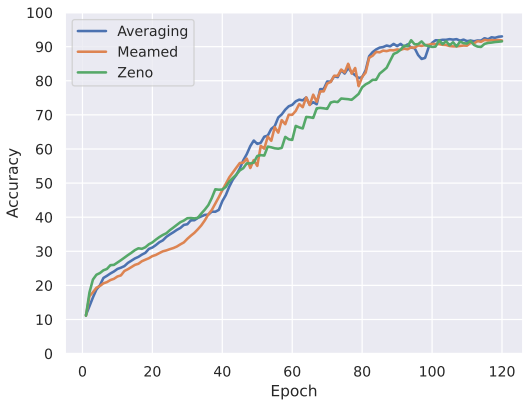
Figure 6.6 shows the evaluation of network accuracy using Averaging, Zeno, and Meamed aggregation methods under various conditions : (a) normal training with no attack, (b) training under sign flip attack, where gradients are manipulated, and (c) training amidst a noise attack, where random Gaussian noise is added to local updates.

By simulating adversarial scenarios of one malicious peer, such as a sign flipping attack [230] where the malicious peer inverts and amplifies its local gradient, and a noise attack [231] 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.

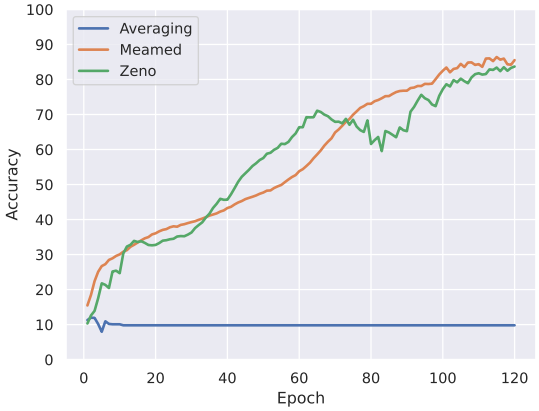
The adoption of the Meamed and Zeno aggregation algorithms significantly increases computational overhead when compared to a baseline aggregation method. Specifically, as shown in table 6.1, the computational time per epoch for the Meamed and Zeno algorithms increased by approximately 8.2 and 5.9 times, respectively.

Table 6.1 : Computational Overhead of Aggregation Algorithms

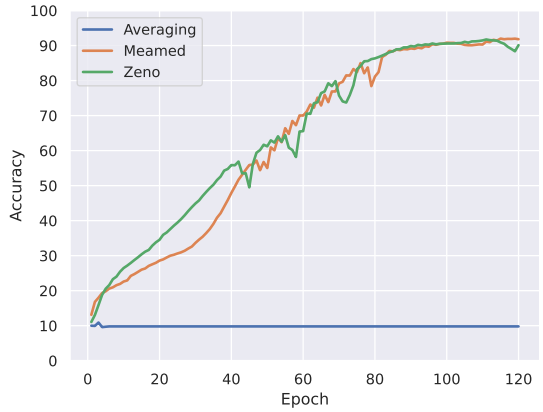
Algorithm	Time (s) / epoch	Overhead
Zeno	310.34	5.9 times
Meamed	431.32	8.2 times



(a) Normal training without attack



(b) Training under sign flip attack



(c) Training with noise attack

Figure 6.6 : Evaluation of network accuracy using Averaging, Zeno, and Meamed aggregation methods under various conditions

In a scenario without adversarial attacks, all three aggregation methods – Averaging, Zeno, and Meamed – achieved an accuracy above 90% within approximately 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. During a noise attack scenario, both Zeno and Meamed reached convergence above 90% after nearly 90 epochs, but the normal averaging method remained divergent.

6.4 FAULT-TOLERANCE AGAINST PEERS FAILURE

Ensuring fault-tolerance against peers failure holds a significant importance, as the system could potentially encounter a deadlock scenario in the event of a peer failure. This situation might arise when other peers are in a state of anticipation, expecting the failed peer to produce its local gradient, persist it within its respective database, and subsequently dispatch a notification to the synchronization queue to announce the fulfillment of its task.

In order to mitigate this concern, a heartbeating mechanism has been incorporated within the system. Before progressing to the synchronization phase encapsulated within the **Synchronize** Lambda Function, each peer undertakes an assessment of the well-being of other peers by invoking the **Check Heartbeats** Lambda Function. This assessment is conducted through sending a signal to the respective Databases of these peers, followed by an awaiting period for their corresponding responses. The absence of a response to this signal signifies that a peer is inactive. Consequently, the identified peer is subsequently excluded from the list of trusted peers and enlisted within the record of inactive peers within the databases maintained by the functional peers. The reason behind our dependence on the database to attain fault-tolerance against peers failure stems from the fact that within the architecture, the database stands as the sole stateful entity that we can monitor.

Upon the successful identification of inactive peers within the network through the Lambda Function tasked with executing the heartbeat mechanism, a consensus among the network's peers must be reached concerning inactive peers. To uphold the integrity of this procedure, the catalog of inactive peers associated with each individual peer is subjected to a meticulous validation against the inactive peers from all other nodes within the network to ensure that a peer is classified as inactive only when it attains consensus agreement from the majority of peers.

Once the agreement is done, inactive peers data should not be lost given their potential significance within the training process. To address this concern, a systematic approach is adopted wherein the data owned by an inactive peer is partitioned and subsequently distributed among the active peers. This distribution is orchestrated in accordance with a ranking system, wherein each active peer inherits a portion of the data from the inactive peer, aligned with their respective rank. Figure 6.7 demonstrates how this segmentation is performed.

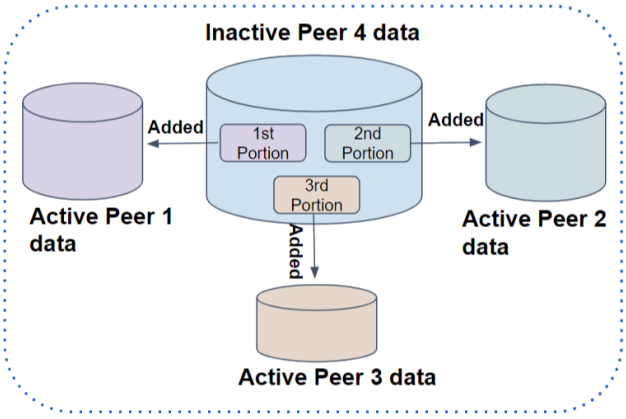


Figure 6.7 : Simplified illustration of data redistribution after a peer failure

An increase in data volume necessitates to create a new AWS Step Function to be able to support the new portion of data. Hence, the **Update & Trigger Next epoch** Lambda Function takes care of this matter by supplying the new AWS Step Function with the appropriate number of data batches required for the next training epoch along with other necessary inputs. The new ARN of the newly created Step Function, through which it can be invoked, is saved to the Redis database and reemployed in the next epochs if no new requirements arise.

6.4.1 FAILURE SIMULATION APPROACH

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.

Heartbeat monitoring for failure detection

Our system includes a crucial 'heartbeat' mechanism activated every epoch, where each peer assesses the status of other peers' databases. Peers send signals and wait for responses to ensure operational health. Each peer pings the databases of others, as these are the only stateful components.

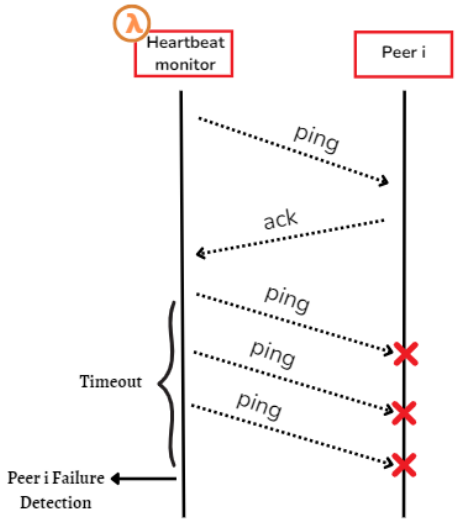


Figure 6.8 : Heartbeat monitoring mechanism

If a peer, as illustrated in Figure 6.8, fails to respond within a set time, it is marked 'inactive' and moved from the trusted peers list to the 'inactive peers' list. This continuous monitoring ensures the network's reliability and maintains seamless peer-to-peer communication.

6.4.2 EXPERIMENTAL RESULTS

6.4.2.1 *PEER FAILURE SCENARIO*

The flow of the experiment are depicted in Figure 6.9 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.

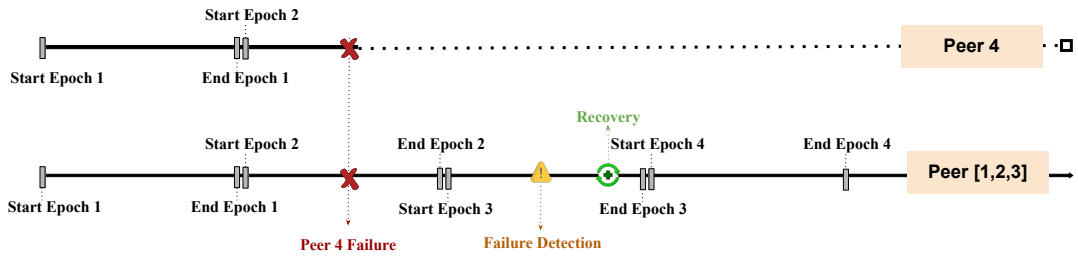


Figure 6.9 : Recovery Time when a Peer Fails

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 represented a worst-case scenario, extending the detection period. Nonetheless, the remaining peers identified the failed peer within the ongoing epoch, taking a total of 50.9 seconds to align with the next epoch's heartbeat step.

Following the single-peer detection, the remaining peers reached a consensus on the failed peer within 9.66 seconds, aggregating the total detection time to 61.56 seconds. Despite

the deferred detection of the simulated failure in the third epoch, the training process was not interrupted, maintaining progression towards model convergence.

Upon consensus on the peer failure, the recovery process was triggered, creating a new AWS Step Function to redistribute the failed peer's computational workload. This increased the remaining peers' load from 15 to 20 batches per epoch. The complete recovery process, including the creation, deployment, and database entry of the new AWS Step Function, required an additional 1.94 seconds. After recovery, the peers continued training in the fourth 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 transitioned from four to three peers. This increase is attributed to the additional gradients being averaged due to the redistribution of the failed peer's workload.

6.4.2.2 *INTEGRATION OF 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.

6.5 CHAPTER SUMMARY

This chapter investigates integrating serverless computing with distributed machine learning (ML) optimization, emphasizing the peer-to-peer (P2P) topology. While P2P offers benefits like improved data privacy and fault tolerance, it also presents significant security

challenges. We address these by implementing robust authentication protocols and secure communication channels using RSA cryptography.

Key contributions include:

1. **Communication Security:** Implementing secure peer initialization and authentication processes using RSA cryptography.
2. **Byzantine Fault Tolerance:** Utilizing robust aggregation techniques such as Meamed and Zeno to ensure model convergence despite the presence of malicious peers.
3. **Failure Mitigation:** Incorporating a heartbeat mechanism to detect peer failures and redistributing data among remaining peers to maintain the training process.

We provide detailed implementations and evaluations of these contributions, demonstrating their effectiveness in enhancing the security and fault tolerance of serverless distributed ML training. Experimental results show that our system can maintain high accuracy and resilience against adversarial attacks and peer failures.

CHAPTER VII

SPIRT: A FAULT-TOLERANT AND RELIABLE PEER-TO-PEER SERVERLESS ML TRAINING ARCHITECTURE

7.1 CHAPTER OVERVIEW

Distributed machine learning (ML) has become increasingly important in tackling the rising complexity and large amounts of data associated with ML models, which traditional single-machine learning frameworks struggle to manage. This approach harnesses multiple computational nodes for simultaneous processing, greatly reducing the time necessary to train larger, more intricate models[192].

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) architectures [31]. 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 [30, 232].

In the parameter server architecture, for instance, the worker nodes perform computations on their respective data partitions and communicate with the parameter server (PS) to update the global model [59]. In contrast, peer-to-peer (P2P) architectures distribute the model parameters and computation across all nodes in the network, eliminating the need for a central coordinator [233].

In the pursuit of optimizing distributed ML architectures, serverless computing has emerged as a revolutionary development. Today, the growing trend of serverless computing, including platforms such as Amazon Lambda[218], Google Cloud Functions[219],

and Azure Functions[220], offers flexibility and cost-effective solutions for distributed ML systems[3, 168, 140]. A common trait among existing serverless architectures—whether based on Parameter Server or P2P setups—is a heavy reliance on databases as a communication channel. This necessity arises from the stateless nature of serverless architectures and limitations in directly transmitting large data sizes[44, 23]. As a result, communication latency can arise, especially during the iterative process of model update and gradient aggregation[23, 234]. In fact, in these architectures, the model parameters and computed gradients must be fetched from the database, processed, and re-uploaded in each iteration. This recurring cycle imposes a significant burden, leading to additional computational and communication costs that can significantly impact overall efficiency.

Notably, while the usage of serverless computing within parameter server architecture has been extensively studied and proven to yield benefits like cost reduction [84], scalability [25, 70], and performance efficiency [200]. Surprisingly, however, the potential of serverless computing within peer-to-peer (P2P) architectures remains less explored, despite the immense advantages P2P offers in terms of data privacy, load balancing, and fault tolerance, as a single node's failure does not disrupt the system [235].

In relation to this, our preceding work [5] explored the integration of serverless computing within VM-based P2P architecture, aiming to offload excessive gradient computation when faced with resource constraints. Despite the benefits of this strategy, it represents a piecemeal approach that does not fully harness the potential of a completely serverless system.

Adding to these challenges, security is a significant concern in distributed Machine Learning (ML) environments, particularly in Peer-to-Peer (P2P) based architectures [221]. One issue is the absence of stringent mechanisms to authenticate new peers joining the network

[226, 236]. Unauthenticated or malicious peers could join the network, potentially causing significant disruption to the training process.

Beyond the threat of unauthenticated new peers, even existing, trusted peers can become compromised, introducing and contributing malicious gradients intended to sway the model training in harmful directions, severely compromising the model's integrity and performance [45, 226, 111]. This risk emphasizes the importance of robust aggregation methods [91, 92, 227], which can help mitigate the effect of such outliers on the overall model's performance.

Addressing the outlined challenges, we introduce the Serverless Peer Integrated for Robust Training (SPIRT) Architecture. In this setup, each peer is identified by its associated database, e.g., Redis, and executes shard-based gradient computation, aggregation, and model convergence checks. With orchestrated workflow coordination, e.g., AWS Step Functions, SPIRT effectively overcomes the stateless limitations inherent in serverless computing. Consequently, it takes full advantage of the serverless architecture within a peer-to-peer setup for machine learning training.

In this endeavor, we propose a reliable peer to peer, serverless ML training architecture with the following contributions:

1. **Automated ML training Workflow:** We propose an automated ML training serverless workflow within a P2P architecture, maximizing efficiency and scalability while reducing operational complexities.

2. **Serverless-adapted Redis Enhancement:** We modify RedisAI for serverless ML training systems that rely on databases, introducing in-database model updates to eliminate the traditional fetch-process-reupload cycle. This enhancement streamlines communication and reduces overhead, facilitating more efficient serverless ML operations.

3. **Enhanced Fault Tolerance:** We integrate secure, scalable peer-to-peer communication, new participant integration, and robust aggregation mechanisms to establish a fault-tolerant environment. This approach ensures reliable distributed ML training, even in the presence of potential disruptions or adversarial actors.

To ensure transparency and foster further research, we have made a replication package for this study accessible [7].

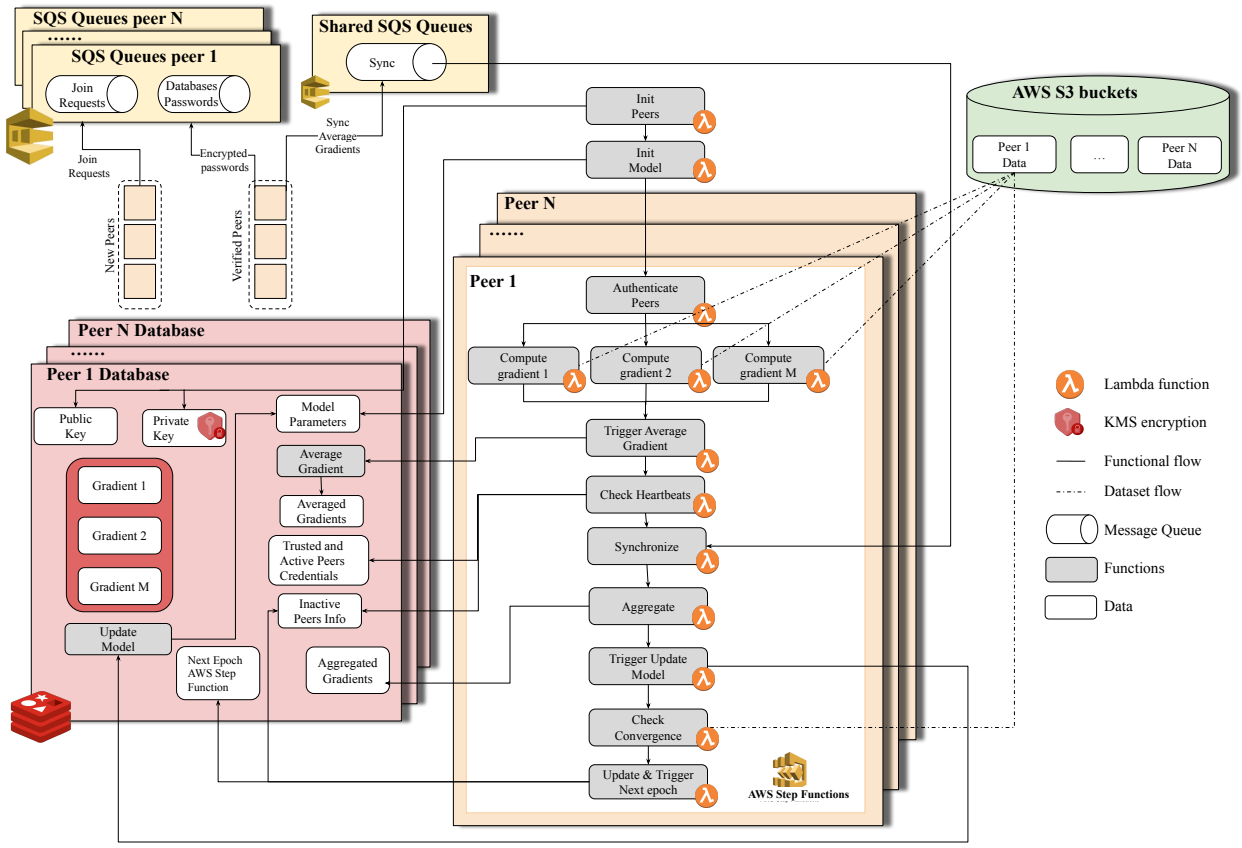


Figure 7.1 : Overview of the proposed Peer To Peer training based on Serverless computing

[7] https://github.com/AmineBarrak/TrainingML_P2P_Serverless

7.2 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.

7.2.1 COMPREHENSIVE OVERVIEW OF THE PROPOSED ARCHITECTURE

Our proposed serverless P2P distributed training architecture, as illustrated in Figure 7.1, 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 [237]. Notably, comparable services on platforms like Google Cloud, Azure, and IBM Cloud enable possible architecture replication.

7.2.2 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.

7.2.2.1 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.

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

7.2.2.3 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.

7.2.2.4 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.

7.2.2.5 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.

7.2.2.6 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.

7.2.3 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.

7.2.3.1 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. More details can be found in chapter 6.

7.2.3.2 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.

7.2.3.3 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.

7.2.3.4 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.

7.2.3.5 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.

7.2.3.6 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.

7.2.3.7 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.

7.2.3.8 MODEL UPDATE

Utilizing the in-database programming feature of RedisAI, we directly update each peer's model parameters stored in the Redis database using the aggregate gradient, thereby bypassing the conventional read-process-write cycle.

7.2.3.9 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.

7.2.3.10 UPDATE AND TRIGGER NEW EPOCH

In the context of our AWS Step Function based workflow, each step function orchestrates the process for a single epoch. The stateful nature of this service necessitates the instantiation of a new Step Function at the end of each epoch to maintain the continuity of training inside each peer. To facilitate this, we employ a dedicated Lambda function that is responsible for spawning the new Step Function. This Lambda function initializes the new Step Function with the correct inputs, including the subsequent epoch number, the degree of parallelism for gradient computation tasks, and a flag indicating whether a convergence check is required (based on the current epoch number). This process ensures an updated step function based on current needs. The ARN (Amazon Resource Name) of the new function is stored in the Redis database, which is reused in subsequent epochs if no new requirements arise.

The same Lambda function also undertakes the responsibility of updating the list of inactive peers through a consensus-driven approach. To ensure the integrity of this process, the list of inactive peers for each node is cross-validated against corresponding lists from all other nodes within the network. An inclusive agreement is applied, meaning a peer is only marked as inactive if it is listed as such in every peer's record.

7.2.3.11 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, failure, or straggler peers. When one or many peers become inactive or fail, our dedicated Lambda function first identifies these instances using the consensus-based approach described previously. Similarly, when a peer becomes a straggler—i.e., when its processing time significantly lags behind the rest of the peers—our system detects this delay. Once inactive, failed, or straggler peers are identified, the data originally assigned to them is segmented and distributed among the active peers based on a predefined ranking system. Each peer, according to their rank, inherits a corresponding portion of the data from the inactive or underperforming peer, ensuring seamless progress despite delays or failures.

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. More details can be found in chapter 6.

7.3 EXPERIMENTAL SETUP

In order to evaluate the efficacy of the Serverless Peer Integrated for Robust Training (SPIRT) architecture, we design a series of experiments to evaluate the performance of various CNN models across different datasets on the proposed architectures.

7.3.1 DATASETS

We utilized a public datasets for our experiments:

- **MNIST:** The MNIST Handwritten Digit Collection [204] consists of 60,000 handwritten digit samples, each belonging to one of ten classes.

To ensure the model's accuracy was measured on unseen data, we split the dataset into training and test sets during the training process.

7.3.2 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 [207].
- **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 [238].
- **DenseNet-121:** A uniquely structured CNN with about 8 million parameters and 121 layers. It leverages dense connections, where each layer is connected to all other layers, enhancing learning efficiency [239].

7.3.3 REDIS AND REDISAI CONFIGURATION

We utilized Redis and RedisAI for in-database model updates. Each peer's Redis was deployed within an Amazon EC2 R6a.large instance. To access the database hosted on EC2,

we utilized SSH tunnels to forward the port and enable access from public services such as AWS Lambda.

7.3.4 AWS LAMBDA CONFIGURATION

Several AWS Lambda functions, discussed in Section 7.2, 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 zipped package must be less than 50MB, and the unzipped files cannot exceed 250MB.

We utilized a Virtual Private Cloud (VPC) to minimize the need for multiple SSH connections between compute gradients Lambda functions and the database EC2 instance.

7.4 EFFICIENCY AND PERFORMANCE IN SERVERLESS P2P ARCHITECTURE

7.4.1 MOTIVATION:

Our innovative serverless peer-to-peer architecture marks the advent of a decentralized era in model training. Peers are enabled to concurrently compute gradients in parallel based on their assigned datasets, exchange gradients for robust aggregation, and update their model parameters. Yet, a theoretically sound architecture is merely the initial phase; the critical next step is to assess its performance under various operational conditions using multiple deep learning models of different complexity and computational demands.

Moreover, it's crucial to explore scalability in two significant aspects: *(1) Intra-peer scalability:* The ability of our architecture to manage parallel gradient computations within a

single peer, and *(2) Inter-peer scalability:* The adaptability of our architecture to changing numbers of peers in the system.

Our motivation lies in thoroughly understanding our architecture's scalability under these critical dimensions. By altering the number of peers and harnessing each peer's parallelization capabilities, we can observe their impacts on performance metrics like gradient computation time, aggregation time, and overall training time per epoch. This approach gives us a holistic view of our architecture's performance under varied operational circumstances.

7.4.2 APPROACH:

For our experiments, we utilized two distinct deep learning models, Densenet121 and MobileNet V3 Small, to represent a range of model complexities, using the standard MNIST dataset for training.

Intra-peer scalability was explored by adjusting the batch size, thereby determining the number of concurrent gradient computations within a peer. With batch sizes of 64, 128, 256, 512, and 1024, we traced the impact on gradient computation and averaging times.

Inter-peer scalability was assessed by varying the number of peers, starting from 4 and incrementing by 2, up to 12. This helped us gauge the system's adaptability and its effect on parameters like aggregation time and training time per epoch.

7.4.3 RESULTS:

As we observed in Figure 7.2, an increase in batch size yields an increased time to compute gradients. This is expected, as a larger batch size implies more data to process at once per peer, which intuitively would increase the time needed for computation. We observed

this increase in computation time consistently across both Densenet121 and MobileNet V3 Small models, signifying that the effect is model-agnostic. Additionally, this trend persists across different numbers of peers. This indicates that the increase in computational burden due to a larger batch size cannot be offset simply by adding more peers, as the computation time per gradient computation still increases.

On the other hand, as we decrease the batch size, we enable more parallel gradient computations due to the availability of more smaller-sized batches. Although this parallelization could potentially lead to faster computation times, we interestingly notice an increase in the average computation time within each peer's database due to the overhead associated with averaging multiple gradients.

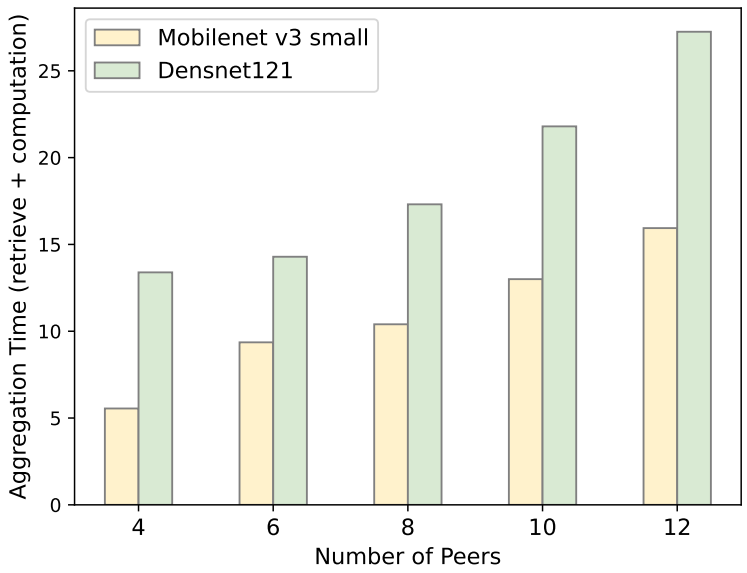


Figure 7.3 : Comparison of Average Aggregation Times for Mobilenet v3 small and Densnet121 Models Against # of Peer

Further dissection of the number of peers' impact on performance, independent of batch size, reveals an interesting trend. As illustrated in Figure 7.3, the aggregation time, which

includes both the computation of the aggregated gradient and the time to retrieve locally averaged gradients from other peers, decreases when the number of peers is reduced (as shown in the bar plot). This suggests an inherent computational overhead linked with gradient aggregation over larger networks of peers, a factor that requires careful consideration when designing and scaling such decentralized learning systems.

Table 7.1 : Training time per epoch across different batch sizes and peer counts for MobileNet V3 Small and Densenet121 models

	Mobilenet v3 small (Batch Size)					Densenet121 (Batch Size)				
# Peers	64	128	256	512	1024	64	128	256	512	1024
4	96.27	57.46	**39.61**	**41.41**	**42.85**	375.82	202.62	**125.7**	**90.67**	**81.17**
6	68.55	43.69	**33.05**	**36.11**	**44.39**	268.13	150.16	**98.41**	**73.39**	**73.77**
8	54.42	38.05	**32.5**	**35.64**	**39.48**	209.79	121.41	**83.65**	**67.44**	**65.25**
10	48.97	36.97	**31.65**	**37.56**	**41.17**	182.32	110.95	**79**	**67.02**	**63.89**
12	48.43	37.3	**32.57**	**37.21**	**39.53**	169.12	104.39	**74.41**	**65.19**	**62.03**

The data presented in Table 7.1 illustrate a notable trend of decreased training time per epoch with increases in both the number of peers and the batch size for the training of MobileNet V3 Small and Densenet121 models. Specifically, we observe that as the batch size doubles, the training time significantly decreases, particularly when the number of peers exceeds four. This effect is more pronounced for the Densenet121 model, which starts with higher absolute training times, suggesting that more complex models may benefit more substantially from scale in serverless distributed training environments.

7.5 DISCUSSIONS

This section critically examines the results of our study and the performance of SPIRT, identifying its implications for peer-to-peer serverless ML training and potential directions for future research.

7.5.1 SERVERLESS P2P FOR ML TRAINING

In our research, we created a distributed training workflow utilizing serverless computing, customized to inherit characteristics from a peer-to-peer (P2P) system, such as robustness, fault tolerance, and scalability. This design led to the creation of logical peers, each focusing on training a distinct part of the dataset. The statelessness of serverless computing necessitates the use of redis databases into our architecture for maintaining state information and enabling inter-component communication. To reduce communication overhead, we incorporated a modified version of RedisAI into our architecture, enabling immediate model updates within the database. Our proposed architecture paves the way for further exploration into the enhancement of in-database computing in distributed serverless ML systems.

7.5.2 SECURITY, RELIABILITY, AND FAULT TOLERANCE IN MACHINE LEARNING ARCHITECTURE

Exploring scalability, a cornerstone of distributed systems, we delved into balancing inter and intra-peer scalability. While handling multiple data batches within a single peer increases efficiency, it also risks a single point of failure. Conversely, extending processing power across multiple peers introduces challenges related to gradient averaging and multiple connections. By testing our system with varied model and batch sizes, and different numbers of peers, while further exploration is required to fully comprehend these trade-offs, our efforts lay a significant foundation towards improving scalability in distributed ML systems.

In terms of fault tolerance, our design incorporates a robust heartbeat system to monitor peer health. Although this method is generally effective, it can introduce minor delays in detecting failures due to its periodic checking nature. To enhance detection speed, more frequent heartbeats could be implemented. As for fault recovery, our architecture ensures swift

recovery time either by intra-peer scalability (adding a new peer) or by redistributing the load among existing peers (inter-peer scalability).

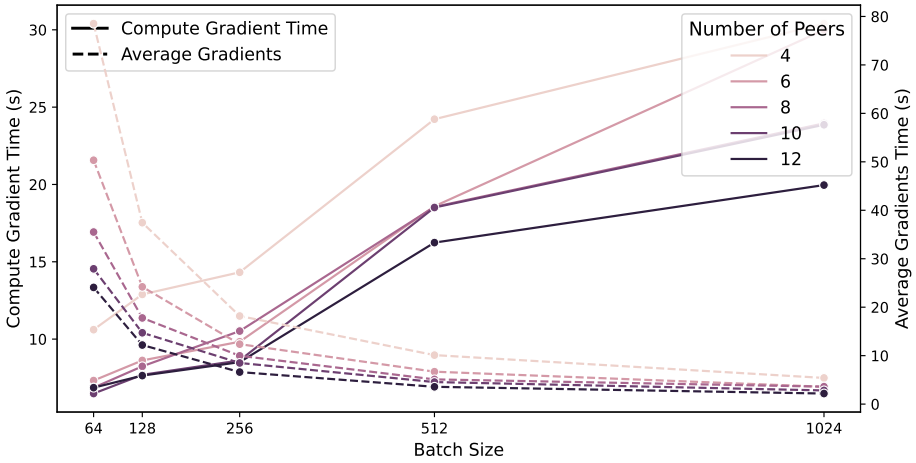
Aiming to augment the security of our architecture, we've implemented sophisticated cryptographic mechanisms which provide robust safeguards for data integrity, authenticity, and confidentiality. These measures ensure that data shared between two peers can only be understood by them. In addition, we've adopted robust aggregation that securely consolidates gradients from diverse peers. While the robust aggregation procedure may extend the aggregation time - for instance, take longer than the average aggregation - its capacity to safeguard against Byzantine attacks and guarantee model convergence underscores its importance.

7.6 CHAPTER SUMMARY

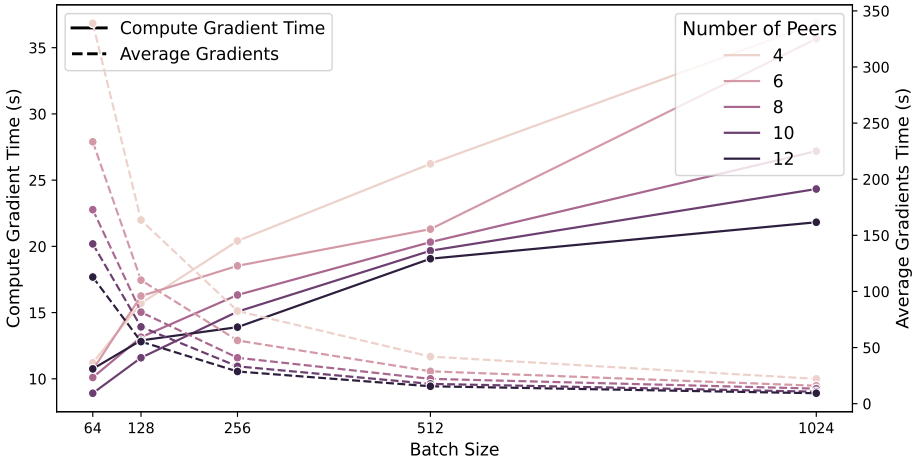
In this study, we present SPIRT, a serverless machine learning (ML) architecture that streamlines training in distributed settings. By minimizing communication overhead, demonstrating resilience to adversarial attacks, and seamlessly integrating new peers.

We found that SPIRT's use of in-database operations in RedisAI resulted in remarkable computational and communication efficiency, improving processing times by up to 82%. Moreover, SPIRT exhibited a high degree of resilience amidst disruptions, rapidly adapting to peer failures with minimal impact on training times and integrating new peers swiftly into the training network. When subjected to adversarial scenarios, the robustness of the architecture became more apparent. Through the use of Byzantine-resistant aggregation methods, Zeno and Meamed, high accuracy levels above 85% and 90% were maintained during sign flip and noise attacks, respectively.

These findings provide a robust foundation for the evolution of distributed ML training frameworks, paving the way for advancements in efficiency, resilience, and scalability in serverless environments.



(a) MobileNet V3 Small Model



(b) DenseNet-121 Model

Figure 7.2 : Comparison of Compute Gradient Time and Average Gradients Across Varying Batch Sizes for Different Number of Peers

CHAPTER VIII

ADVANCING SERVERLESS ML TRAINING ARCHITECTURES VIA COMPARATIVE APPROACH

8.1 CHAPTER OVERVIEW

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[192].

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) topologies [31]. 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 [30, 232]. 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 [59]. 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 [233].

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 [240]. 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 [241].

In response to these challenges, serverless computing has emerged as a revolutionary solution within the domain of distributed ML architectures [3, 168, 140]. Platforms such as Amazon Lambda[218], Google Cloud Functions[219], and Azure Functions[220] 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 [46], scalability [25, 70], and training time [5].

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[44, 23]. As a result, communication latency can arise, especially during the iterative process of model update and gradient aggregation[23, 234]. **(2) Fault Tolerance and Security Concerns**: Ensuring robust fault tolerance in distributed environments is another significant concern in distributed Machine Learning (ML) environments [221]. One issue is the absence of mechanisms to authenticate new nodes joining the network [226, 236]. 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 [45, 226, 111]. This risk emphasizes the importance of robust aggregation methods [91, 92, 227], which can help mitigate the effect of such outliers on the overall model's performance.

In the Chapter **??**, we addressed these challenges by presenting the Serverless Peer Integrated for Robust Training (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 [23] 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 [46] 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 chapter, 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 [8].

This chapter provides valuable insights for practitioners in selecting the most suitable serverless distributed ML architecture for specific scenarios, based on empirical evidence. Concurrently, the chapter 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.

8.2 COMPARATIVE ANALYSIS OF SERVERLESS ML FRAMEWORKS

In this section, we conduct a systematic comparison of serverless machine learning training frameworks, specifically SPIRT [4], MLLess [46], and the two proposed communication

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

patterns proposed by LambdaML [23]. 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.

8.2.1 SERVERLESS ML TRAINING WORKFLOW

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

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

8.2.2 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. In the following, we describe the communication

Table 8.1 : Comparative Analysis of Serverless Training Framework Architectures: An Overview of Key Training Computational Stages

Serverless Framework	Training Stage	Content of the Stage
SPIRT [4]	Fetch Dataset	1. Fetch a worker's minibatches.
	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.
	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.
	Model Update	9. Update the model.
MLLESS [46]	Fetch Dataset	1. Fetch a minibatch.
	Compute Gradients	2. Compute gradient of minibatch.
	Synchronisation	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.
	Model Update	7. Update the model.
Scatter Reduce [23]	Fetch Dataset	1. Fetch a minibatch.
	Compute Gradients	2. Compute gradient of minibatch.
	Synchronisation	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.
	Model Update	9. Update the model.
ALL Reduce [23]	Fetch Dataset	1. Fetch a minibatch.
	Compute Gradients	2. Compute gradient of minibatch.
	Synchronisation	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	8. Update the model.

mechanisms employed by these frameworks to enhance distributed ML tasks in a serverless environment.

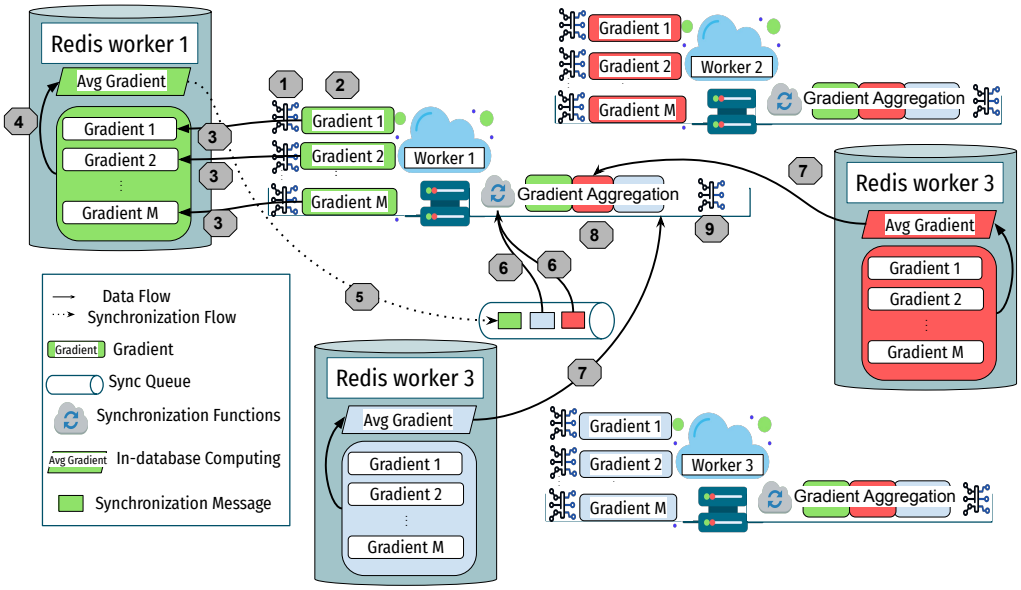


Figure 8.1 : SPIRT architecture workflow

In **SPIRT (Figure 8.1)**, 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.
4. The **averaging** of these gradients within each worker's database.
5. A **notification** is sent to the synchronization queue indicating the completion of gradient averaging.

6. Workers **poll** the synchronization queue until the message count aligns with the number of peers involved.

7. Peers **retrieve** the averaged gradients from each other's databases.

8. **Aggregate** these averages and **save** it within database.

9. Finally, **proceed** to update their local models within their databases.

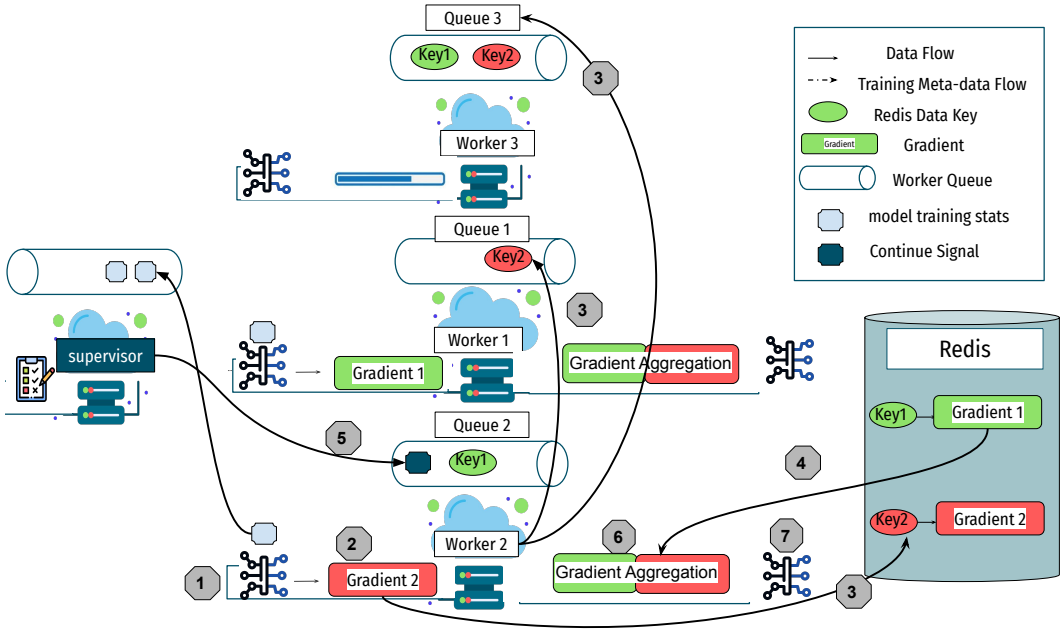


Figure 8.2 : MLLESS architecture workflow

In **MLLESS (Figure 8.2)**, the workflow begins with the following steps:

1. Each worker **fetches** a minibatch.

2. The workers **compute** the gradient of their respective minibatches.

3. Upon identifying a significant update, 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.

4. Workers continuously **monitor** their queues to accumulate the keys of received updates from other workers and to read from the 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.

7. Finally, the **aggregated gradients** are used to **update** the model.

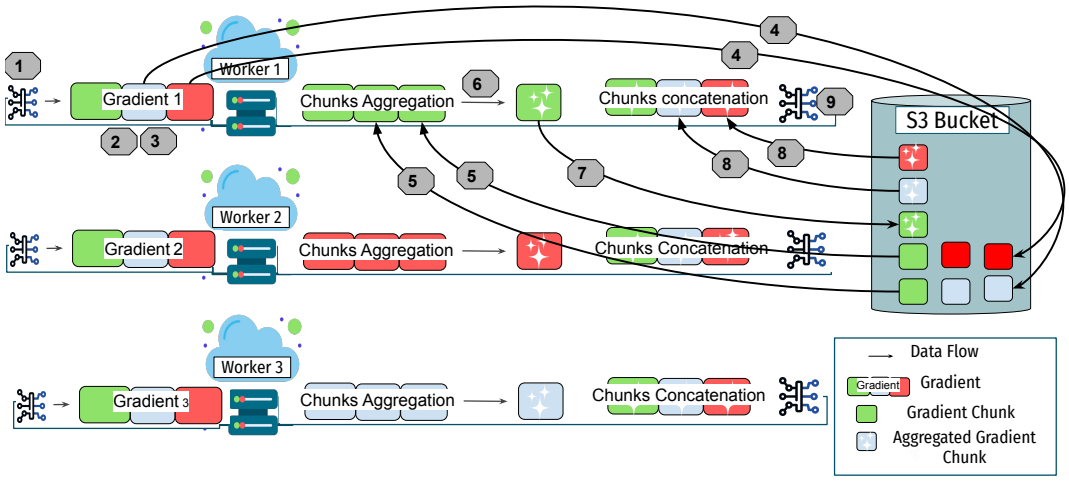


Figure 8.3 : ScatterReduce-LambdaML architecture workflow

In the ScatterReduce-LambdaML framework, as depicted in **Figure 8.3**, the workflow of each worker node is meticulously structured into nine distinct stages. The procedure is as follows: In **ScatterReduce-LambdaML (Figure 8.3)**, each worker proceeds as follows:

1. **Fetching** a minibatch of the dataset to process.
2. **Computing** the gradients of this minibatch.
3. **Dividing** the computed gradient into chunks, each intended for a different worker.
4. **Keeping** its respective chunk and **sending** the others to a shared database.
5. **Fetching and aggregating** the chunks assigned to them, which have been gathered from other workers.
6. **Sending** the aggregated chunk back to the database.
7. **Retrieving** all the aggregated chunks from the database.
8. **Concatenating** these aggregated chunks to assemble the full gradient.
9. **Updating** the model with this complete gradient.

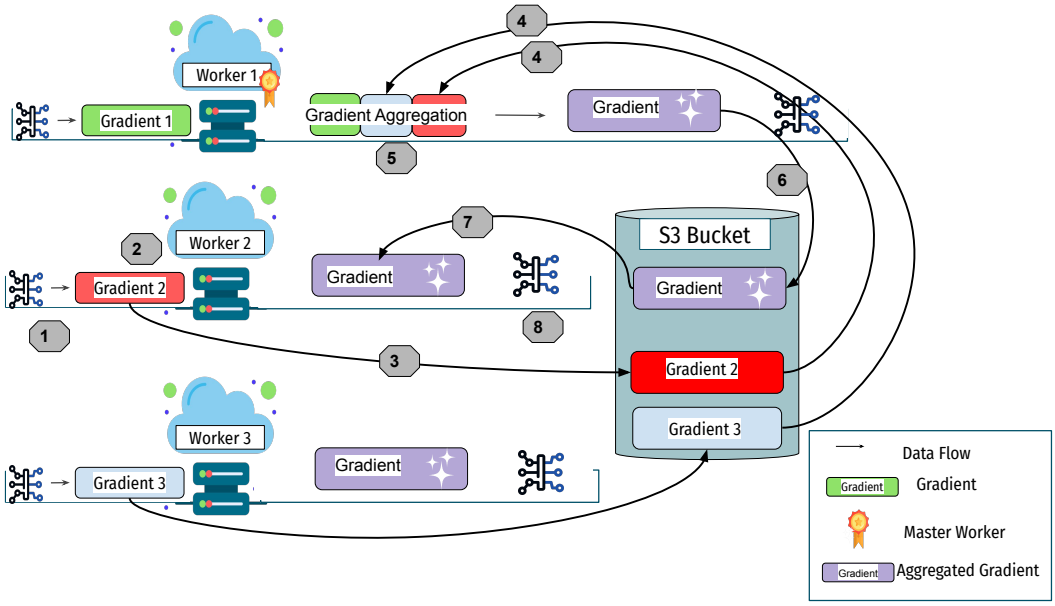


Figure 8.4 : AllReduce-LambdaML architecture workflow

In **AllReduce-LambdaML (Figure 8.4)**, each worker starts by carefully fetching a minibatch from the dataset.

1. **Fetching** a minibatch from the dataset.
2. **Computing** the gradients of the minibatch.
3. **Sending** these gradients to a shared database.
4. The master worker (usually worker with ID 1) **retrieves** all the gradients from the database.
5. The master worker **performs** the aggregation process to combine these gradients into a single, unified gradient.
6. The master worker **sends** this aggregated gradient back to the shared database.
7. Each worker **fetches** the aggregated gradient from the database.

8. Finally, each worker **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.

8.2.3 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.

Table 8.2 : Comparative analysis of Serverless Machine Learning frameworks: Gradients storage

SPIRT [4]	MLLess [46]	AllReduce [23]	ScatterReduce [23]
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:** 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.

Table 8.3 : Comparative Analysis of Serverless Machine Learning Frameworks: Communication channels

SPIRT [4]	MLLess [46]	AllReduce [23]	ScatterReduce [23]
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:** This criterion examines the strategies and techniques used to minimize the amount of communication required between workers.

Table 8.4 : Comparative analysis of Serverless Machine Learning frameworks: Communication overhead reduction

SPIRT [4]	MLLess [46]	AllReduce [23]	ScatterReduce [23]
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:** This assesses the synchronization mechanisms used to coordinate task progression among distributed workers, including conditions and constraints like conditional waits.

Table 8.5 : Comparative analysis of Serverless Machine Learning frameworks: Synchronization barrier

SPIRT [4]	MLLess [46]	AllReduce [23]	ScatterReduce [23]
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:**

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 [242], the parallelization within Lambda functions is constrained by their single-threaded design, resulting in sequential processing.

Table 8.6 : Comparative analysis of Serverless Machine Learning frameworks: Batch processing

SPIRT [4]	MLLess [46]	AllReduce [23]	ScatterReduce [23]
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

- **Fault Tolerance:** This evaluates the robustness of a distributed system against worker or supervisor node failures, detailing strategies for operational continuity, state recovery methods, and resilience mechanisms that enable the system to maintain or swiftly restore its functions.

Table 8.7 : Comparative analysis of Serverless machine learning frameworks: Fault tolerance

SPIRT [4]	MLLess [46]	AllReduce [23]	ScatterReduce [23]
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:** 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.

Table 8.8 : Comparative analysis of Serverless Machine Learning frameworks: Auto-Scaling

SPIRT [4]	MLLess [46]	AllReduce [23]	ScatterReduce [23]
A worker can be added at any point of the training without affecting the other workers.	None.	None.	None.

- **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.

Table 8.9 : Comparative analysis of Serverless Machine Learning Frameworks: Security measures

SPIRT [4]	MLLess [46]	AllReduce [23]	ScatterReduce [23]
Implemented robust aggregation techniques, ensured worker authentication upon network entry, and secured communications through encryption.	None.	None.	None.

8.3 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.

8.3.1 EXPERIMENTAL SETUP

8.3.1.1 DATASETS

We utilized two public datasets:

- **MNIST:** The MNIST Handwritten Digit Collection [204] consists of 60,000 handwritten digit samples, each belonging to one of ten classes.

- **CIFAR:** The CIFAR Image Dataset [205] 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.

To ensure the model's accuracy was measured on unseen data, we split each dataset into training and test sets during the training process.

8.3.1.2 MODEL ARCHITECTURES AND HYPERPARAMETERS

The experiments involve four 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 [207].

- **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 [238].

- **ResNet-50:** A more advanced deep learning CNN architecture, ResNet-50 encompasses approximately 25.6 million parameters across 50 layers [238].

8.3.2 AWS LAMBDA CONFIGURATION

Several AWS Lambda functions, discussed in Section 7.2, 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.

8.3.3 DATASET DIVISION FOR WORKERS AND BATCHES

During experimentation with datasets such as MNIST, the dataset is divided among the available workers, with each worker receiving a portion. Each worker then splits its dataset portion into batches, processing these batches to compute gradients during the learning process. For example, if the MNIST dataset is divided among 4 workers with a batch size of 128, each worker would handle 15,000 images, processing them in approximately 118 batches to compute gradients. 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.

8.3.4 SERVERLESS FRAMEWORKS REPLICATIONS

We have replicated LambdaML and MLLess, to explore their capabilities. Initially, MLLess was designed for compact models such as Logistic Regression, with mentions of potential adaptability for deep learning models. However, this adaptability was not actualized. We have since extended the framework to include implementations for deep learning model support.

8.4 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.

8.4.1 SERVERLESS FRAMEWORKS TRAINING TIME

8.4.1.1 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 8.2.1, 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. Furthermore, we seek to understand the scalability potential performance gains or bottlenecks when scaling up the number of workers.

8.4.1.2 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.

Building on this foundation, we expanded our analysis to explore scalability by conducting additional training sessions with varying numbers of workers, ranging from 4 to 16.

8.4.1.3 RESULTS:

We observed distinct performance characteristics across various stages of the process, including fetching the dataset, computing gradients, synchronization, and model updating. Detailed observations are illustrated in Figure 8.5.

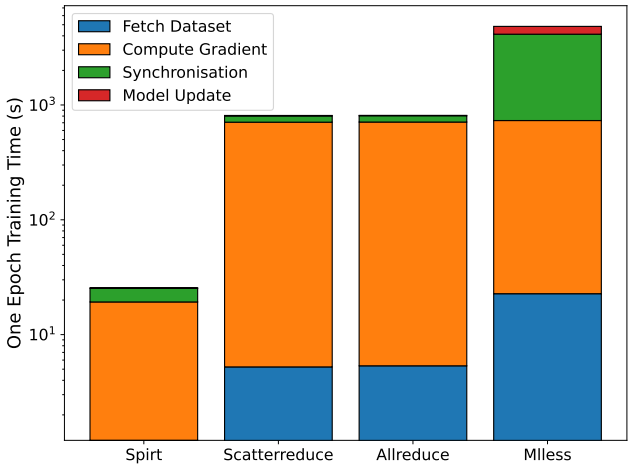


Figure 8.5 : Training time for one epoch across serverless training frameworks, depicted on a logarithmic scale.

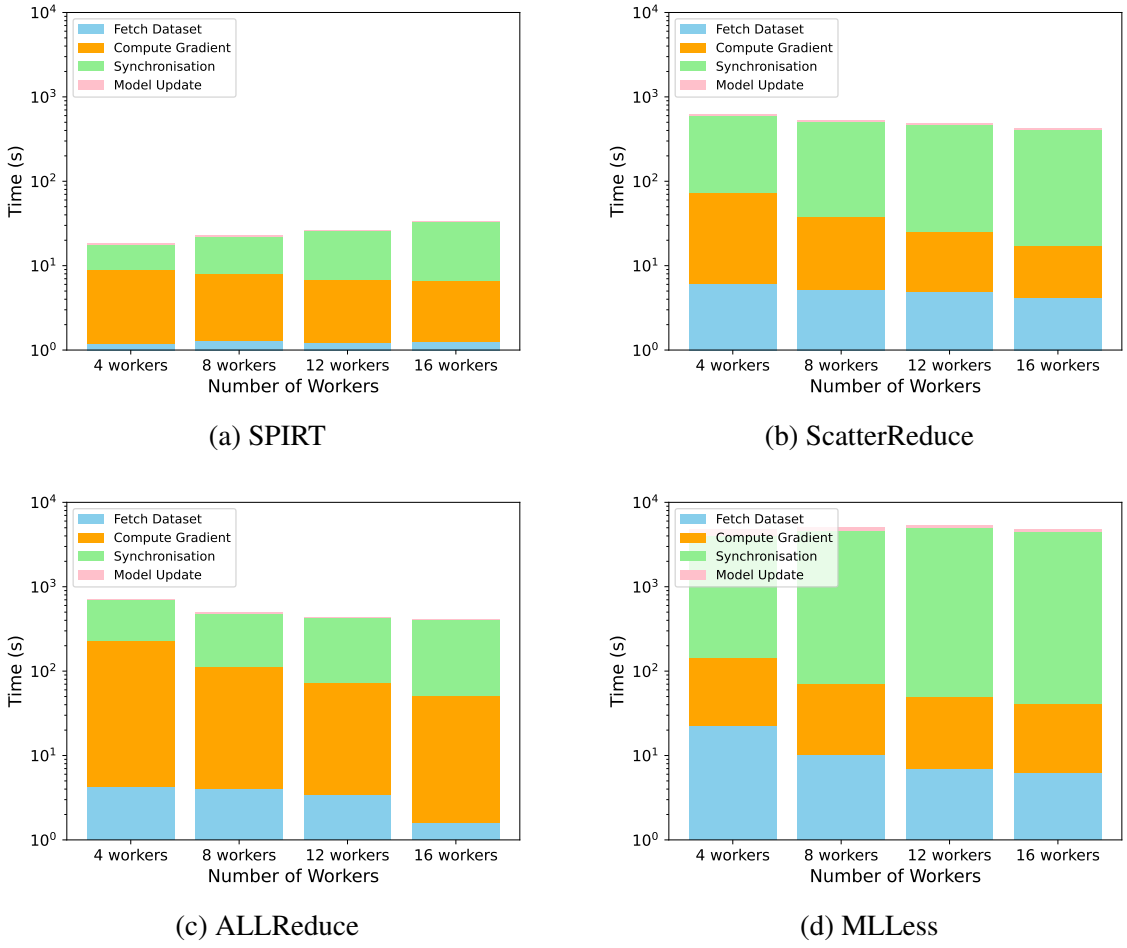
Spirt has the advantages to speed up its epoch by parallelizing the batch processing, with notably rapid dataset fetching (1.2s) and gradient computation (18.055s). To complete one epoch, Spirt needs one synchronisation between workers (6.03s) and one model update (0.28s).

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.



(a) SPIRT

(b) ScatterReduce

(c) ALLReduce

(d) MLLess

Figure 8.6 : Training Time Evolution with Increasing Workers: This figure illustrates how the training time for different stages evolves as the number of workers increases from 4 to 16, across four serverless architectures: SPIRT, ScatterReduce, ALLReduce and MLLess.

Training time evolution across various serverless architectures with an increasing number of workers is depicted in Figure 8.6.

The scalability analysis of SPIRT, ScatterReduce, ALLReduce, and MLLess architectures reveals that increasing the number of workers from 4 to 16 generally improves compute gradient times but highlights persistent synchronization bottlenecks. In SPIRT, compute gradient time decreases by 30.7% (from 7.71s to 5.34s), while synchronization time increases by

192.5% (from 9.03s to 26.42s). ScatterReduce shows an 80% reduction in compute gradient time (from 66.53s to 13.29s) but only a 25.7% decrease in synchronization time (from 530.24s to 393.73s). ALLReduce exhibits a 78.2% decrease in compute gradient time (from 227.54s to 49.61s) with synchronization time reducing by 25.3% (from 476.90s to 356.18s). In MLLess, compute gradient time drops by 71% (from 121.17s to 35.07s), yet synchronization time remains extremely high, reducing by only 16.2% (from 3891.8s to 4530.71s). Comparing the frameworks, ScatterReduce and ALLReduce show the best scalability for compute gradient, while MLLess suffers the most from synchronization bottlenecks. These results emphasize that while compute gradient benefits from increased parallelism, synchronization remains a significant scalability challenge.

8.4.2 SERVERLESS FRAMEWORKS COST ANALYSIS

8.4.2.1 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.

8.4.2.2 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 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 [5] 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

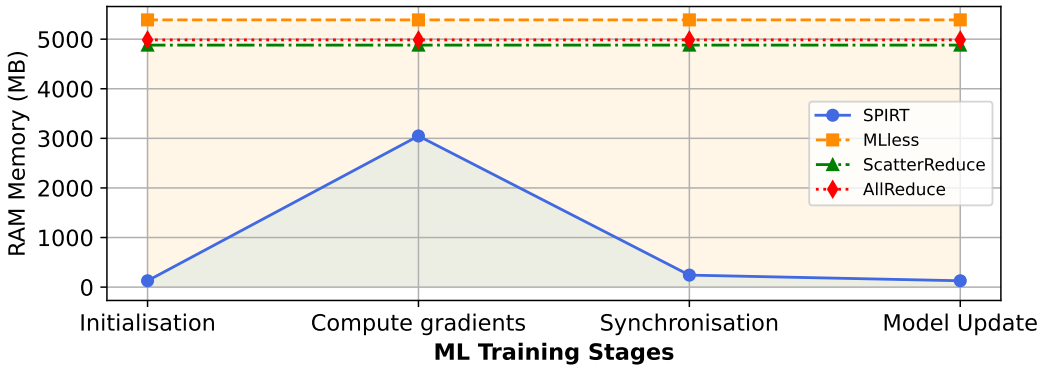


Figure 8.7 : Memory Allocation in Serverless Computing for Training Workflows: A Comparison between SPIRT and Other Serverless Training Frameworks.

each framework's lambda function reported its peak memory usage. Figure 8.7 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 [243]. Additionally, we used the AWS Pricing Calculator to precisely estimate the costs associated with each framework [244].

Tables 8.10 and 8.11 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.

8.4.2.3 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

Table 8.10 : 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 [4]	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 [23]	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 [23]	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 [46]	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.

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.

Table 8.11 : 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**

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 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.

8.4.3 COMMUNICATION OVERHEAD REDUCTION

8.4.3.1 SPIRT PARALLEL BATCH PROCESSING

Motivation: In training ML models, the process involves updating the model incrementally by processing data in batches. This often requires frequent updates between workers, introducing significant overhead.

Our approach differs by distributing the initial dataset among workers and then splitting these segments into minibatches. Each worker processes these minibatches in parallel. Instead of communicating updates after each minibatch, we aggregate the gradients from all minibatches and communicate these accumulated updates only once all parallel processing is complete.

Approach: In previous work [5], 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:

$$\text{Cost per worker}_{\text{parallel}} = \big[\text{Lambda Cost} \times \text{Num of batches} + \text{Trigger Instance Cost}\big] \times \text{Comp. Time} \tag{8.1}$$

$$\text{Cost per worker}_{\text{sequential}} = \text{Instance Cost} \times \text{Comp. Time} \tag{8.2}$$

Results: Our results in Table 8.12 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.

Table 8.12 : 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**

8.4.3.2 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 [5], LambdaML [23], SMLT [25], and MLLess [84] 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 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: Our experimental results demonstrate significant efficiency improvements facilitated by in-database operations. Notably, the time taken for gradient averaging calculations and model updates is substantially reduced when performed within the database. For example, the MobileNetV3 Small model showed an impressive decrease in computation time from 135.29 seconds outside the database to 78.52 seconds within, for a batch size of 64. Similarly, the larger ResNet-18 model exhibited a 44.43% improvement in processing efficiency for gradient averaging. Moreover, model update times for the MobileNetV3 Small

model decreased by 82%, and for the ResNet-18 model, the reduction was approximately 83%.

For a detailed examination of these results, please refer to Chapter 5.

8.4.3.3 LAMBDAML SCATTERREDUCE VS ALLREDUCE

Motivation: In LambdaML's study [23], authors explored the impact of communication patterns between scatter reduce and all reduce. Their evaluation with 10 workers on two models, MobileNet and ResNet, showed that as model size increases, scatter reduce becomes faster because communication gets heavier and the single reducer 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 the 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 8.8, 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

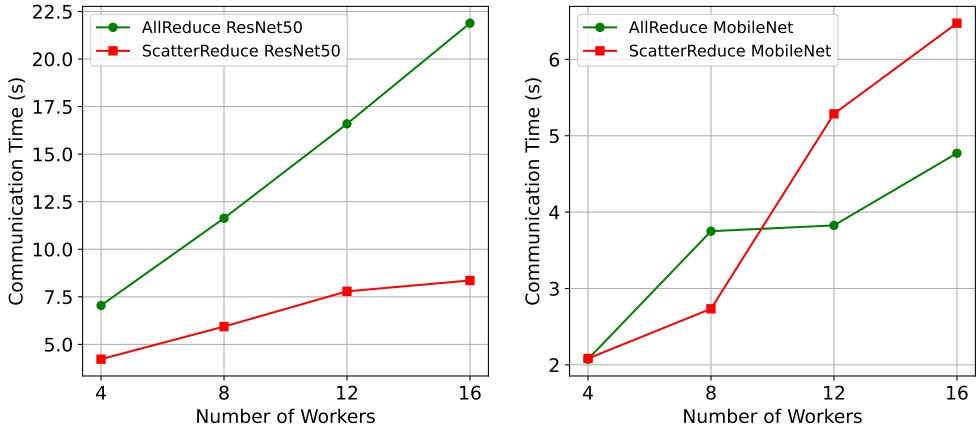


Figure 8.8 : Comparison of communication times between scatter reduces, and all reduce patterns as a function of the number of workers for 1 training step.

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.

8.4.3.4 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 [46] 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 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 8.9, the implementation of a significance filter within our experimental framework has improved the convergence rates over traditional training

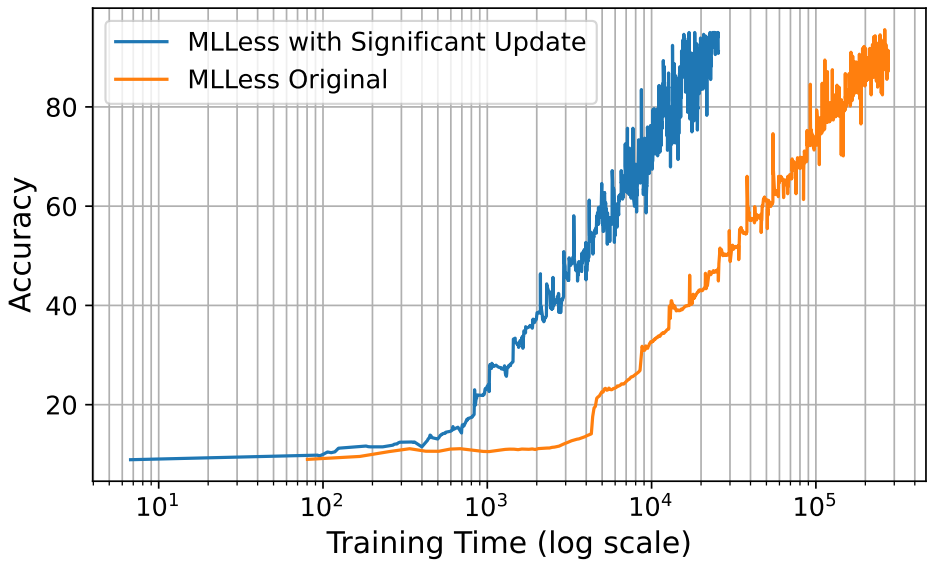


Figure 8.9 : Significant Update Evaluation on MLLess

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.

8.4.4 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.

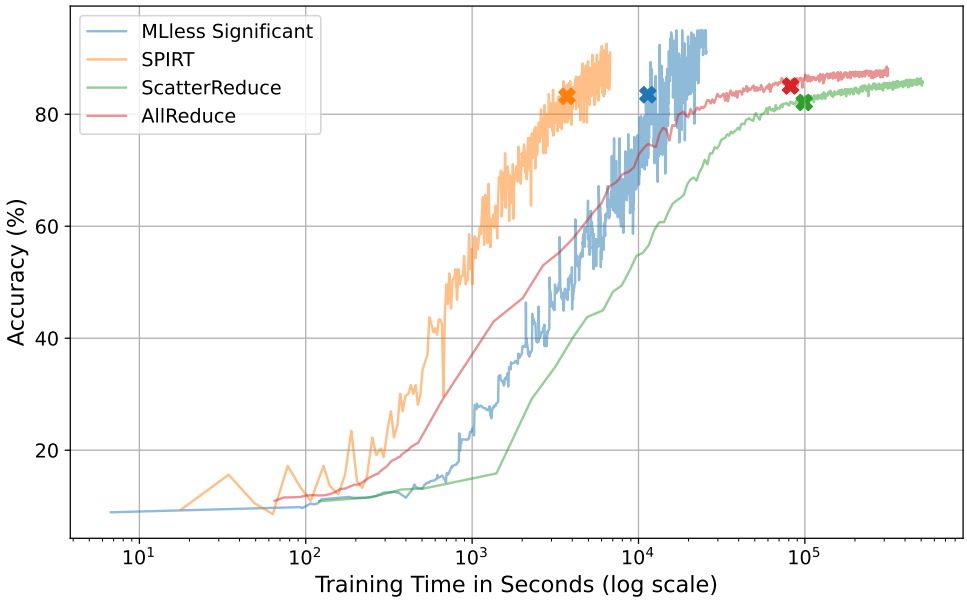


Figure 8.10 : Comparative Accuracy Evaluation of Serverless Training Frameworks

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.

8.5 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.

8.5.1 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.

8.5.2 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.

8.5.3 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.

8.5.4 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.

8.5.5 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.

8.6 CHAPTER SUMMARY

In this chapter, 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.

CHAPTER IX

CONCLUSION AND FUTURE WORK

In this thesis, we conducted a set of studies to understand and enhance the application of serverless computing in machine learning (ML) training, particularly focusing on distributed environments, including peer-to-peer (P2P) architectures. The rapid advancement in machine learning, coupled with the exponential growth of data, has led to the development of increasingly complex models that demand significant computational resources. Traditional single-machine training approaches are often insufficient to handle these demands, necessitating distributed training methods.

Serverless computing has emerged as a promising paradigm due to its scalability, cost efficiency, and ease of deployment. It allows developers to focus on application functionality without the need to manage the underlying infrastructure. However, the stateless and ephemeral nature of serverless functions poses unique challenges for ML training, which typically requires persistent state and extensive communication.

The main motivations for this research are: **scalability and cost efficiency**, as serverless computing provides dynamic resource allocation and a pay-as-you-go model, beneficial for ML training with variable workloads; **state management and synchronization**, due to the stateless nature of serverless functions, which complicates tasks requiring persistent state and coordination; **communication overhead**, since distributed ML training, especially in P2P architectures, involves significant data transfer, necessitating overhead reduction; **fault tolerance and security**, as distributed ML environments face challenges like node failures and malicious attacks; and **optimization of training architectures**, as current serverless

ML frameworks often have limitations, highlighting the need for optimized architectures, particularly P2P, to fully utilize serverless computing benefits. Our research hypothesis stated:

> **Thesis Hypothesis:**
> Serverless computing provides an optimal environment for scalable and reliable machine learning training.

We validate this hypothesis by confirming the sub-hypotheses presented in the next.

9.1 SUB-HYPOTHESIS ONE (CHAPTERS 3 AND 4)

Sub-hypothesis: Serverless functions can accelerate parallel gradient computation during ML training, although they introduce challenges related to state management, synchronization, and communication overhead.

We conducted a systematic mapping study to understand the usage of serverless computing in ML pipelines and developed a novel serverless-based architecture for P2P ML training. The study revealed a growing adoption of serverless computing for various stages of the ML pipeline, particularly in the training phase. Our proposed architecture demonstrated that serverless functions could significantly accelerate the training process by enabling dynamic resource allocation and parallel gradient computation.

The systematic mapping study involved reviewing over 150 studies to explore the integration of serverless computing within ML pipelines. We identified key benefits such as cost reduction, scalability, and simplified deployment. However, challenges were also noted, including state management, synchronization, and communication overhead.

In our novel serverless-based architecture for P2P ML training, serverless functions were utilized to perform parallel gradient computation across distributed nodes. To address the challenges of statelessness and coordination, we introduced cumulative gradients as a strategy to aggregate and synchronize updates efficiently across nodes. This approach leveraged the scalability and cost-efficiency of serverless computing, resulting in faster training times. However, the stateless nature of serverless functions still introduced challenges in maintaining persistent state and coordinating between functions, which required advanced state management and synchronization mechanisms.

This sub-hypothesis is validated, demonstrating that while serverless functions can accelerate parallel gradient computation, they necessitate solutions for managing state, synchronization, and communication overhead.

9.2 SUB-HYPOTHESIS TWO (CHAPTER 5)

Sub-hypothesis: In-database ML operations, such as gradient aggregation and model updates, can effectively reduce communication overhead in serverless distributed training environments by minimizing data transfer, with tools like RedisAI adapted for these operations.

We proposed an architectural framework that integrates customized Redis for performing gradient aggregation and model updates directly within the database. Our experiments with gradient compression and in-database model updates showed significant improvements in training efficiency.

The integration of RedisAI for gradient averaging and model updates effectively reduced communication overhead, demonstrating the potential of in-database operations to enhance serverless ML training environments. By performing these operations within the database, we minimized data transfer latencies and improved overall training performance. Additionally,

the use of gradient compression techniques further optimized data transfer, highlighting the benefits of combining in-database operations with efficient communication strategies.

This sub-hypothesis is validated, as the combination of in-database ML operations and gradient compression significantly reduced communication overhead in serverless distributed training environments.

9.3 SUB-HYPOTHESIS THREE (CHAPTER 6)

Sub-hypothesis: Implementing key mechanisms for fault tolerance and secure communication protocols improves the reliability and security of serverless ML training.

We developed and implemented several mechanisms to enhance fault tolerance and secure communication in serverless ML training. These included encryption protocols for secure communication, heartbeat mechanisms for failure detection, and methods to handle worker failures and exclude gradient outliers.

Secure communication protocols were established using encryption techniques to protect data integrity and confidentiality during training. Heartbeat mechanisms were employed to monitor the status of each worker, allowing for prompt detection and response to failures. To address worker failures, a mechanism was developed to redistribute the dataset among remaining workers, ensuring continuous training. Additionally, to protect against Byzantine attacks, we implemented methods to exclude gradient outliers, enhancing the reliability and security of the training process.

The implementation of these mechanisms significantly improved the reliability and security of the serverless ML training process. Secure communication protocols ensured data integrity and confidentiality, while fault tolerance strategies maintained training continuity

despite worker failures. The methods to exclude gradient outliers effectively protected against Byzantine attacks, validating our sub-hypothesis.

9.4 SUB-HYPOTHESIS FOUR (CHAPTERS 7 AND 8)

Sub-hypothesis: The proposed SPIRT architecture, a fully serverless training workflow, offers superior training speed, performance, and fault tolerance compared to existing serverless ML training frameworks.

We designed and evaluated the SPIRT architecture, a fully serverless ML training framework inspired by peer-to-peer mechanisms. Our comprehensive comparative study demonstrated that SPIRT significantly outperformed existing frameworks in terms of training time efficiency, cost-effectiveness, and fault tolerance.

The SPIRT architecture was designed to optimize serverless ML training by orchestrating training tasks across multiple serverless functions and peers. We proposed two dimensions of scalability: increasing the number of serverless functions to compute gradients within a peer and adding more peers. This architecture avoided single points of failure and optimized resource utilization, ensuring scalability and resilience.

Through extensive experimentation, we validated the scalability and robustness of the SPIRT architecture. Comparative evaluations with state-of-the-art frameworks like ScatterReduce, AllReduce, and MLLess showed that SPIRT achieved superior training time efficiency, cost-effectiveness, and fault tolerance. The results confirmed the advantages of the SPIRT architecture for serverless ML training, highlighting its potential to optimize training processes and enhance performance.

This sub-hypothesis is validated, as the SPIRT architecture demonstrated superior cost-efficiency, training time reduction, and fault tolerance compared to existing serverless ML training frameworks.

9.5 FUTURE WORK

Based on the findings and limitations identified in this thesis, our research contributions open a wide range of opportunities for future work. This section discusses some of them.

One potential direction is the **optimization of execution environments for small language models (SLMs)**. Future research can explore how serverless architectures can be tailored to efficiently train and infer small language models. Techniques such as model partitioning, compression, and adaptive scaling could be investigated to enhance performance while reducing costs. This would be particularly beneficial given the growing importance of SLMs in various applications.

Another promising area is the development of **hybrid architectures that combine serverless computing with traditional cloud resource**s. Such hybrid approaches could leverage the strengths of both paradigms, providing a balanced solution for dynamic workload distribution, cost optimization, and efficient resource allocation. Research could focus on strategies for seamlessly integrating these architectures to handle varying workloads more effectively.

Advanced fault tolerance and security mechanisms also present a significant opportunity for further study. As the reliance on serverless ML training grows, ensuring robust fault tolerance and secure communication becomes crucial. Future work could explore blockchain-based solutions for secure and decentralized coordination, as well as more sophisticated Byzantine fault-tolerant algorithms to enhance resilience against adversarial attacks.

Optimizing **resource management and scheduling algorithms** for serverless ML training is another critical area. Intelligent schedulers that dynamically allocate resources based on real-time workload demands and historical performance data could greatly improve performance and cost efficiency. Research in this area could lead to more effective and adaptive resource management strategies.

The application of serverless computing to **federated learning** offers another intriguing avenue for future research. Federated learning, which involves training models across multiple decentralized devices while keeping data localized, presents unique challenges related to data privacy, communication overhead, and model aggregation. Exploring serverless solutions to these challenges could provide valuable insights and advancements.

Integration with MLOps practices is also a vital area for future work. Developing frameworks and tools to automate end-to-end ML workflows within a serverless architecture could streamline the deployment, monitoring, and maintenance of ML models. This integration would ensure seamless operation within existing MLOps pipelines, enhancing overall efficiency and effectiveness.

Lastly, investigating the **energy efficiency and sustainability** of serverless ML training could contribute to greener computing practices. Future research could focus on optimizing resource utilization and minimizing energy consumption, as well as evaluating the environmental impact of serverless architectures. This work would be crucial in promoting sustainable and environmentally friendly computing solutions.

In conclusion, this dissertation advances the field of serverless computing for machine learning by addressing critical challenges and proposing innovative solutions. The findings highlight the potential of serverless architectures to optimize ML training processes, providing

practical frameworks for real-world applications and setting the stage for future advancements in serverless ML training.

REFERENCES

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

[2] A. Barrak, R. Trabelsi, F. Jaafar, and F. Petrillo, "Advancing serverless ml training architectures via comparative approach," *arXiv preprint*, 2024.

[3] 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.

[4] 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.

[5] 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.

[6] A. Barrak, "The promise of serverless computing within peer-to-peer architectures for distributed ml training," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, no. 21, 2024, pp. 23 383–23 384.

[7] ——, "Incorporating serverless computing into p2p networks for ml training: In-database tasks and their scalability implications (student abstract)," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, no. 21, 2024, pp. 23 439–23 440.

[8] A. Barrak, F. Petrillo, and F. Jaafar, "Architecting peer-to-peer serverless distributed machine learning training for improved fault tolerance," *arXiv preprint arXiv:2302.13995*, 2023.

[9] A. Barrak, G. Fofe, L. Mackowiak, E. Kouam, and F. Jaafar, "Securing aws lambda: Advanced strategies and best practices," in *2024 IEEE 11th International Conference on Cyber Security and Cloud Computing (CSCloud)*. IEEE, 2024, pp. 113–119.

[10] H. Bourreau, E. Guichet, A. Barrak, B. Simon, and F. Jaafar, "On securing the communica-

tion 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.

[11] F. Jaafar, D. Ameyed, A. Barrak, and M. Cheriet, "Identification of compromised iot devices: Combined approach based on energy consumption and network traffic analysis," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2021, pp. 514–523.

[12] S. K. Mohanty, G. Premsankar, and M. Di Francesco, "An evaluation of open source serverless computing frameworks," in *IEEE International Conference on Cloud Computing Technology and Science*. IEEE, 2018, pp. 115–120.

[13] S. Winzinger and G. Wirtz, "Model-based analysis of serverless applications," in *2019 IEEE/ACM 11th International Workshop on Modelling in Software Engineering (MiSE)*. IEEE, 2019, pp. 82–88.

[14] R. A. P. Rajan, "Serverless architecture-a revolution in cloud computing," in *2018 Tenth International Conference on Advanced Computing (ICoAC)*. IEEE, 2018, pp. 88–93.

[15] E. Van Eyk, L. Toader, S. Talluri, L. Versluis, A. Uță, and A. Iosup, "Serverless is more: From paas to present cloud computing," *IEEE Internet Computing*, vol. 22, no. 5, pp. 8–17, 2018.

[16] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Characterizing serverless platforms with serverlessbench," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 30–44.

[17] M. A. N. Rodríguez and F. U. I. Martínez, "Creation of serverless applications in the cloud," in *2022 11th International Conference On Software Process Improvement (CIMPS)*. IEEE, 2022, pp. 216–218.

[18] V. K. Thatikonda, "Serverless computing: Advantages, limitations and use cases," *European Journal of Theoretical and Applied Sciences*, vol. 1, no. 5, pp. 341–347, 2023.

[19] J. Wen, Z. Chen, Y. Liu, Y. Lou, Y. Ma, G. Huang, X. Jin, and X. Liu, "An empirical study

on challenges of application development in serverless computing," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 416–428.

[20] D. Chahal, M. Ramesh, R. Ojha, and R. Singhal, "High performance serverless architecture for deep learning workflows," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2021, pp. 790–796.

[21] L. Feng, P. Kudva, D. Da Silva, and J. Hu, "Exploring serverless computing for neural network training," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 2018, pp. 334–341.

[22] M. Elzohairy, M. Chadha, A. Jindal, A. Grafberger, J. Gu, M. Gerndt, and O. Abboud, "Fedlesscan: Mitigating stragglers in serverless federated learning," in *2022 IEEE International Conference on Big Data (Big Data)*. IEEE, 2022, pp. 1230–1237.

[23] 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.

[24] Y. Wu, T. T. A. Dinh, G. Hu, M. Zhang, Y. M. Chee, and B. C. Ooi, "Serverless model serving for data science," *arXiv preprint arXiv:2103.02958*, 2021.

[25] 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.

[26] M. S. Kurz, "Distributed double machine learning with a serverless architecture," in *Companion of the ACM/SPEC International Conference on Performance Engineering*, 2021, pp. 27–33.

[27] H. Wang, D. Niu, and B. Li, "Distributed machine learning with a serverless architecture," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 1288–1296.

[28] K. Assogba, M. Arif, M. M. Rafique, and D. S. Nikolopoulos, "On realizing efficient deep

learning using serverless computing," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2022, pp. 220–229.

[29] S. K. Lo, Q. Lu, C. Wang, H.-Y. Paik, and L. Zhu, "A systematic literature review on federated machine learning: From a software engineering perspective," *ACM Computing Surveys (CSUR)*, vol. 54, no. 5, pp. 1–39, 2021.

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

[31] 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.

[32] J. Geng, D. Li, and S. Wang, "Accelerating distributed machine learning by smart parameter server," in *Proceedings of the 3rd Asia-Pacific Workshop on Networking*, 2019, pp. 92–98.

[33] K. Zhang, S. Alqahtani, and M. Demirbas, "A comparison of distributed machine learning platforms," in *2017 26th international conference on computer communication and networks (ICCCN)*. IEEE, 2017, pp. 1–9.

[34] N. Daw, U. Bellur, and P. Kulkarni, "Speedo: Fast dispatch and orchestration of serverless workflows," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 585–599.

[35] A. Muhyiddeen, R. M. Nor, and M. H. Rahman, "Analyzing communication overhead in linearizing peer to peer system," in *2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*. IEEE, 2016, pp. 260–263.

[36] P. Flocchini, A. Nayak, and M. Xie, "Enhancing peer-to-peer systems through redundancy," *IEEE Journal on selected areas in communications*, vol. 25, no. 1, pp. 15–24, 2007.

[37] U. Bharti, A. Goel, and S. Gupta, "A scalable design approach for state propagation in serverless workflow," in *2022 IEEE 3rd Global Conference for Advancement in Technology (GCAT)*. IEEE, 2022, pp. 1–7.

[38] 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.

[39] Y. Li, L. Zhao, Y. Yang, and W. Qu, "Rethinking deployment for serverless functions: A performance-first perspective," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023, pp. 1–14.

[40] P. G. López, M. Sánchez-Artigas, G. París, D. B. Pons, Á. R. Ollobarren, and D. A. Pinto, "Comparison of faas orchestration systems," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE, 2018, pp. 148–153.

[41] H. Yu, H. Wang, J. Li, X. Yuan, and S.-J. Park, "Accelerating serverless computing by harvesting idle resources," in *Proceedings of the ACM Web Conference 2022*, 2022, pp. 1741–1751.

[42] A. Mampage, S. Karunasekera, and R. Buyya, "Deadline-aware dynamic resource management in serverless computing environments," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2021, pp. 483–492.

[43] 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.

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

[45] 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.

[46] 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

[47] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep gradient compression: Reducing the communication bandwidth for distributed training," *arXiv preprint arXiv:1712.01887*, 2017.

[48] H. Gao and H. Huang, "Adaptive serverless learning," *arXiv preprint arXiv:2008.10422*, 2020.

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

[50] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, "Theano: A cpu and gpu math compiler in python," in *Proc. 9th python in science conf*, vol. 1, 2010, pp. 3–10.

[51] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, 2014, pp. 675–678.

[52] 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.

[53] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, "Graphlab: A new framework for parallel machine learning," *arXiv preprint arXiv:1408.2041*, 2014.

[54] S. Lee, J. K. Kim, X. Zheng, Q. Ho, G. A. Gibson, and E. P. Xing, "On model parallelization and scheduling strategies for distributed machine learning," *Advances in neural information processing systems*, vol. 27, 2014.

[55] P. Baran, "On distributed communications networks," *IEEE transactions on Communications Systems*, vol. 12, no. 1, pp. 1–9, 1964.

[56] A. Agarwal, O. Chapelle, M. Dudík, and J. Langford, "A reliable effective terascale linear learning system," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1111–1133, 2014.

[57] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P.

Xing, "Managed communication and consistency for fast data-parallel iterative analytics," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015, pp. 381–394.

[58] I. Foster and A. Iamnitchi, "On death, taxes, and the convergence of peer-to-peer and grid computing," in *Peer-to-Peer Systems II: Second International Workshop, IPTPS 2003, Berkeley, CA, USA, February 21-22, 2003. Revised Papers 2*. Citeseer, 2003, pp. 118–128.

[59] 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.

[60] E. P. Xing, Q. Ho, P. Xie, and D. Wei, "Strategies and principles of distributed machine learning on big data," *Engineering*, vol. 2, no. 2, pp. 179–195, 2016.

[61] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project adam: Building an efficient and scalable deep learning training system," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 571–582.

[62] M. Han and K. Daudjee, "Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 9, pp. 950–961, 2015.

[63] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G. R. Ganger, P. B. Gibbons, and O. Mutlu, "Gaia: Geo-distributed machine learning approaching lan speeds." in *NSDI*, 2017, pp. 629–647.

[64] T. Elgamal, "Costless: Optimizing cost of serverless computing through function fusion and placement," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2018, pp. 300–312.

[65] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *Proceedings of the 2017 symposium on cloud computing*, 2017, pp. 445–451.

[66] B. Carver, J. Zhang, A. Wang, and Y. Cheng, "In search of a fast and efficient serverless dag engine," in *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*. IEEE, 2019, pp. 1–10.

[67] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, "Mllib: Machine learning in apache spark," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1235–1241, 2016.

[68] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: A serverless framework for end-to-end ml workflows," in *Proceedings of the ACM Symposium on Cloud Computing*, 2019, pp. 13–24.

[69] 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.

[70] 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.

[71] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 583–598.

[72] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang *et al.*, "Large scale distributed deep networks," *Advances in neural information processing systems*, vol. 25, 2012.

[73] I. MONGODB, "The most popular database for modern apps| mongodb," *Acesso em*, vol. 3, 2021.

[74] M. Zinkevich, M. Weimer, L. Li, and A. Smola, "Parallelized stochastic gradient descent," *Advances in neural information processing systems*, vol. 23, 2010.

[75] A. Koloskova, T. Lin, S. U. Stich, and M. Jaggi, "Decentralized deep learning with arbitrary communication compression," *arXiv preprint arXiv:1907.09356*, 2019.

[76] S. U. Stich, "Local sgd converges fast and communicates little. iclr-international conference on learning representations, art," *arXiv preprint arXiv:1805.09767*, 2019.

[77] Z. Li, D. Kovalev, X. Qian, and P. Richtárik, "Acceleration for compressed gradient descent in distributed and federated optimization," *arXiv preprint arXiv:2002.11364*, 2020.

[78] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent," *Advances in neural information processing systems*, vol. 24, 2011.

[79] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project adam: Building an efficient and scalable deep learning training system," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 571–582.

[80] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang *et al.*, "Large scale distributed deep networks," *Advances in neural information processing systems*, vol. 25, 2012.

[81] S. U. Stich, "Local sgd converges fast and communicates little. iclr-international conference on learning representations, art," *arXiv preprint arXiv:1805.09767*, 2019.

[82] 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.

[83] 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.

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

[85] "Redisai - a server for machine and deep learning models," https://oss.redis.com/redisai/#quick-links, (Accessed on 06/20/2023).

[86] B. Custers, A. M. Sears, F. Dechesne, I. Georgieva, T. Tani, and S. Van der Hof, *EU personal data protection in policy and practice*. Springer, 2019.

[87] R. Chard, Y. Babuji, Z. Li, T. Skluzacek, A. Woodard, B. Blaiszik, I. Foster, and K. Chard, "Funcx: A federated function serving fabric for science," in *Proceedings of the 29th International symposium on high-performance parallel and distributed computing*, 2020, pp. 65–76.

[88] 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)*, 2021, pp. 150–157.

[89] U. Aïvodji, S. Gambs, and T. Ther, "Gamin: An adversarial approach to black-box model inversion," *arXiv preprint arXiv:1909.11835*, 2019.

[90] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, "Membership inference attacks against machine learning models (2017)," *arXiv preprint arXiv:1610.05820*, 2016.

[91] 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.

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

[93] P. J. Rousseeuw, "Multivariate estimation with high breakdown point," *Mathematical statistics and applications*, vol. 8, no. 283-297, p. 37, 1985.

[94] R. Guerraoui, S. Rouault *et al.*, "The hidden vulnerability of distributed learning in byzantium," in *International Conference on Machine Learning*. PMLR, 2018, pp. 3521–3530.

[95] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, 1987, pp. 1–12.

[96] J. A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. H. Epema, M. Reinders, M. R. Van Steen, and H. J. Sips, "Tribler: a social-based peer-to-peer system," *Concurrency and computation: Practice and experience*, vol. 20, no. 2, pp. 127–138, 2008.

[97] D. Evans, V. Kolesnikov, M. Rosulek *et al.*, "A pragmatic introduction to secure multi-party computation," *Foundations and Trends® in Privacy and Security*, vol. 2, no. 2-3, pp. 70–246, 2018.

[98] Z. Wu, T. Chen, and Q. Ling, "Byzantine-resilient decentralized stochastic optimization with robust aggregation rules," *arXiv preprint arXiv:2206.04568*, 2022.

[99] T. Khalifa, "Secure data aggregation protocol with byzantine robustness for wireless sensor networks," Master's thesis, University of Waterloo, 2007.

[100] M. Chadha, A. Jindal, and M. Gerndt, "Towards federated learning using faas fabric," in *Proceedings of the 2020 sixth international workshop on serverless computing*, 2020, pp. 49–54.

[101] E. Rojas, D. Pérez, J. C. Calhoun, L. B. Gomez, T. Jones, and E. Meneses, "Understanding soft error sensitivity of deep learning models and frameworks through checkpoint alteration," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2021, pp. 492–503.

[102] 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.

[103] Y. Bouizem, D. Dib, N. Parlavantzas, C. Morin, and F. Lahfa, *Request Replication for FaaS Fault Tolerance*, 2022.

[104] K. Janocha and W. M. Czarnecki, "On loss functions for deep neural networks in classification," *arXiv preprint arXiv:1702.05659*, 2017.

[105] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*, pp. 400–407, 1951.

[106] "High availability and scalability on aws - real-time communication on aws," https://docs.aws.amazon.com/whitepapers/latest/real-time-communication-on-aws/high-availability-and-scalability-on-aws.html, (Accessed on 02/23/2023).

[107] "Cross-region replication in azure | microsoft learn," https://learn.microsoft.com/en-us/azure/reliability/cross-region-replication-azure, (Accessed on 02/23/2023).

[108] "Resilience in aws lambda - aws lambda," https://docs.aws.amazon.com/lambda/latest/dg/security-resilience.html, (Accessed on 02/23/2023).

[109] V. Biyani, "Fission | serverless functions for kubernetes," https://fission.io/docs/, (Accessed on 02/16/2023).

[110] J. Bernstein, J. Zhao, K. Azizzadenesheli, and A. Anandkumar, "signsgd with majority vote is communication efficient and fault tolerant," *arXiv preprint arXiv:1810.05291*, 2018.

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

[112] L. Chen, H. Wang, Z. Charles, and D. Papailiopoulos, "Draco: Byzantine-resilient distributed training via redundant gradients," in *International Conference on Machine Learning*. PMLR, 2018, pp. 903–912.

[113] S. Rajput, H. Wang, Z. Charles, and D. Papailiopoulos, "Detox: A redundancy-based framework for faster and more robust gradient aggregation," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[114] J. Weng, J. Weng, J. Zhang, M. Li, Y. Zhang, and W. Luo, "Deepchain: Auditable and privacy-preserving deep learning with blockchain-based incentive," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2438–2455, 2019.

[115] X. Chen, J. Ji, C. Luo, W. Liao, and P. Li, "When machine learning meets blockchain: A decentralized, privacy-preserving and secure design," 12 2018, pp. 1178–1187.

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

[117] "13 benefits of cloud computing for your business | globaldots," https://shorturl.at/bfoP3, july 2018, (Accessed on 04/16/2022).

[118] ReportsandData, "Function-as-a-service (faas) market size worth usd 31.53 billion at cagr of 32.3%, by 2026 - report and data - ein presswire," https://shorturl.at/cs0Ml, Oct 2021, (Accessed on 01/20/2022).

[119] M. Ribeiro, K. Grolinger, and M. A. Capretz, "Mlaas: Machine learning as a service," in *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2015, pp. 896–902.

[120] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Commun. ACM*, vol. 62, no. 12, p. 44–54, nov 2019. [Online]. Available: https://doi.org/10.1145/3368454

[121] K. Kanagaraj and S. Geetha, "A hybrid framework for effective prediction of online streaming data," in *Journal of Physics: Conference Series*, vol. 1767, no. 1. IOP Publishing, 2021, p. 012016.

[122] A. Kaplunovich and Y. Yesha, "Refactoring of neural network models for hyperparameter optimization in serverless cloud," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020, pp. 311–314.

[123] K. Petersen, S. Vakkalanka, and L. Kuzniarz, "Guidelines for conducting systematic mapping studies in software engineering: An update," *Information and Software Technology*, vol. 64, pp. 1–18, 2015.

[124] "The 4 types of cloud computing services | exitcertified," https://www.exitcertified.com/blog/4-cloud-computing-services, (Accessed on 07/13/2022).

[125] "Introduction to cloud computing for machine learning beginners," https://shorturl.at/eMKf0, (Accessed on 07/13/2022).

[126] "How serverless architecture can impact the future of ai and ml industries | engineering education (enged) program | section," https://shorturl.at/sK4l5, (Accessed on 07/13/2022).

[127] E. Mendes and F. Petrillo, "Log severity levels matter: A multivocal mapping," *arXiv preprint arXiv:2109.01192*, 2021.

[128] S. Keele *et al.*, "Guidelines for performing systematic literature reviews in software engineering," Technical report, ver. 2.3 EBSE Technical Report. EBSE, Tech. Rep., 2007.

[129] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "A case for serverless machine learning," in *Workshop on Systems for ML and Open Source Software at NeurIPS*, vol. 2018, 2018.

[130] A. Deese, "Implementation of unsupervised k-means clustering algorithm within amazon web services lambda," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2018, pp. 626–632.

[131] Z. Tu, M. Li, and J. Lin, "Pay-per-request deployment of neural network models using serverless architectures," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations*, 2018, pp. 6–10.

[132] V. Ishakian, V. Muthusamy, and A. Slominski, "Serving deep learning models in a serverless platform," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2018, pp. 257–262.

[133] A. Bhattacharjee, A. D. Chhokra, Z. Kang, H. Sun, A. Gokhale, and G. Karsai, "Barista: Efficient and scalable serverless serving system for deep learning prediction services," in *2019 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2019, pp. 23–33.

[134] D. Damkevala, R. Lunavara, M. Kosamkar, and S. Jayachandran, "Behavior analysis using serverless machine learning," in *2019 6th International Conference on Computing for Sustainable Global Development (INDIACom)*. IEEE, 2019, pp. 1068–1072.

[135] M. Fotouhi, D. Chen, and W. J. Lloyd, "Function-as-a-service application service composition: Implications for a natural language processing application," in *Proceedings of the 5th International Workshop on Serverless Computing*, 2019, pp. 49–54.

[136] C. Zhang, M. Yu, W. Wang, and F. Yan, "MArk: Exploiting cloud services for Cost-Effective, SLO-Aware machine learning inference serving," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 1049–1062. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/zhang-chengliang

[137] D. Barcelona-Pons, M. Sánchez-Artigas, G. París, P. Sutra, and P. García-López, "On the faas track: Building stateful distributed applications with serverless architectures," in *Proceedings of the 20th International Middleware Conference*, 2019, pp. 41–54.

[138] M. Zhang, C. Krintz, M. Mock, and R. Wolski, "Seneca: Fast and low cost hyperparameter search for machine learning models," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 404–408.

[139] A. Christidis, R. Davies, and S. Moschoyiannis, "Serving machine learning workloads in resource constrained environments: A serverless deployment example," in *2019 IEEE 12th Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE, 2019, pp. 55–63.

[140] 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

[141] T. Rausch, W. Hummer, V. Muthusamy, A. Rashed, and S. Dustdar, "Towards a serverless platform for edge AI," in *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)*. Renton, WA: USENIX Association, Jul. 2019. [Online]. Available: https://www.usenix.org/conference/hotedge19/presentation/rausch

[142] A. Dakkak, C. Li, S. Garcia de Gonzalo, J. Xiong, and W.-m. Hwu, "Trims: Transparent and isolated model sharing for low latency deep learning inference in function-as-a-service," *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, Jul 2019. [Online]. Available: http://dx.doi.org/10.1109/CLOUD.2019.00067

[143] Á. L. García, J. M. De Lucas, M. Antonacci, W. Zu Castell, M. David, M. Hardt, L. L. Iglesias, G. Moltó, M. Plociennik, V. Tran *et al.*, "A cloud-based framework for machine learning workloads and applications," *IEEE access*, vol. 8, pp. 18 681–18 692, 2020.

[144] A. Kaplunovich and Y. Yesha, "Automatic tuning of hyperparameters for neural networks in serverless cloud," in *2020 IEEE International Conference on Big Data (Big Data)*. IEEE, 2020, pp. 2751–2756.

[145] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "Batch: Machine learning inference serving on serverless platforms with adaptive batching," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–15.

[146] U. Elordi, L. Unzueta, J. Goenetxea, S. Sanchez-Carballido, I. Arganda-Carreras, and O. Otaegui, "Benchmarking deep neural network inference performance on serverless environments with mlperf," *IEEE Software*, vol. 38, no. 1, pp. 81–87, 2020.

[147] C. Zhang, M. Yu, F. Yan *et al.*, "Enabling cost-effective, slo-aware machine learning inference serving on public cloud," *IEEE Transactions on Cloud Computing*, 2020.

[148] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 419–433. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/shillaker

[149] J. R. Gunasekaran, C. S. Mishra, P. Thinakaran, M. T. Kandemir, and C. R. Das, "Implications of public cloud resource heterogeneity for inference serving," in *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, 2020, pp. 7–12.

[150] D. Chahal, R. Ojha, M. Ramesh, and R. Singhal, "Migrating large deep learning models to serverless architecture," in *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2020, pp. 111–116.

[151] J. Choi, J. Lee, and W. J. Cho, "Prognostics by classifying degradation stage on lambda architecture," in *2020 IEEE International Conference on Prognostics and Health Management (ICPHM)*. IEEE, 2020, pp. 1–9.

[152] M. Zhang, C. Krintz, and R. Wolski, "Stoic: Serverless teleoperable hybrid cloud for machine learning applications on edge device," in *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE, 2020, pp. 1–6.

[153] M. Chadha, A. Jindal, and M. Gerndt, "Towards federated learning using faas fabric," in *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, 2020, pp. 49–54.

[154] D. M. Naranjo, S. Risco, G. Moltó, and I. Blanquer, "A serverless gateway for event-driven machine learning inference in multiple clouds," *Concurrency and Computation: Practice and Experience*, vol. n/a, no. n/a, p. e6728. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6728

[155] J. Jarachanthan, L. Chen, F. Xu, and B. Li, "Amps-inf: Automatic model partitioning for serverless inference with cost efficiency," in *50th International Conference on Parallel Processing*, 2021, pp. 1–12.

[156] A. Kaplunovich and Y. Yesha, "Automatic hyperparameter optimization for arbitrary neural networks in serverless aws cloud," in *2021 12th International Conference on Information and Communication Systems (ICICS)*. IEEE, 2021, pp. 69–76.

[157] N. Shahidi, J. R. Gunasekaran, and M. T. Kandemir, "Cross-platform performance evaluation of stateful serverless workflows," in *2021 IEEE International Symposium on Workload Characterization (IISWC)*, 2021, pp. 63–73.

[158] J. Thorpe, Y. Qiao, J. Eyolfson, S. Teng, G. Hu, Z. Jia, J. Wei, K. Vora, R. Netravali, M. Kim, and G. H. Xu, "Dorylus: Affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, Jul. 2021, pp. 495–514. [Online]. Available: https://www.usenix.org/conference/osdi21/presentation/thorpe

[159] M. Zhang, C. Krintz, and R. Wolski, "Edge-adaptable serverless acceleration for machine learning internet of things applications," *Software: Practice and Experience*, vol. 51, no. 9, pp. 1852–1867, 2021.

[160] M. Yu, Z. Jiang, H. C. Ng, W. Wang, R. Chen, and B. Li, "Gillis: Serving large neural networks in serverless functions with automatic model partitioning," in *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2021, pp. 138–148.

[161] E. Paraskevoulakou and D. Kyriazis, "Leveraging the serverless paradigm for realizing machine learning pipelines across the edge-cloud continuum," in *2021 24th Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*. IEEE, 2021, pp. 110–117.

[162] D. Chahal, M. Mishra, S. Palepu, and R. Singhal, "Performance and cost comparison of cloud services for deep learning workload," in *Companion of the ACM/SPEC International Conference on Performance Engineering*, 2021, pp. 49–55.

[163] D. Chahal, S. Palepu, M. Mishra, and R. Singhal, "Sla-aware workload scheduling using hybrid cloud services," in *Proceedings of the 1st Workshop on High Performance Serverless Computing*, 2021, pp. 1–4.

[164] P. Patros, J. Spillner, A. V. Papadopoulos, B. Varghese, O. Rana, and S. Dustdar, "Toward sustainable serverless computing," *IEEE Internet Computing*, vol. 25, no. 6, pp. 42–50, 2021.

[165] A. Nesen and B. Bhargava, "Towards situational awareness with multimodal streaming data fusion: serverless computing approach," in *Proceedings of the International Workshop on Big Data in Emergent Distributed Environments*, 2021, pp. 1–6.

[166] J. Tagliabue, "You do not need a bigger boat: Recommendations at reasonable scale in a (mostly) serverless and open stack," in *Fifteenth ACM Conference on Recommender Systems*, 2021, pp. 598–600.

[167] T. P. Bac, M. N. Tran, and Y. Kim, "Serverless computing approach for deploying machine learning applications in edge layer," in *2022 International Conference on Information Networking (ICOIN)*. IEEE, 2022, pp. 396–401.

[168] 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.

[169] H. B. Hassan, S. A. Barakat, and Q. I. Sarhan, "Survey on serverless computing," *Journal of Cloud Computing*, vol. 10, no. 1, pp. 1–29, 2021.

[170] D. A. Tamburri, "Sustainable mlops: Trends and challenges," in *2020 22nd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE, 2020, pp. 17–23.

[171] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," *arXiv preprint arXiv:1812.03651*, 2018.

[172] H. van Kemenade, "python-pillow/pillow: 9.0.0," Jan. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.5813885

[173] D. G. McNeely-White, J. R. Beveridge, and B. A. Draper, "Inception and resnet: same training, same features," in *Biologically Inspired Cognitive Architectures Meeting*. Springer, 2019, pp. 352–357.

[174] Amazon, "Aws lambda," https://github.com/aws/aws-lambda-go, 2021.

[175] I. E. Akkus, "knix," https://github.com/knix-microfunctions/knix, 2021.

[176] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[177] "Aws lambda now supports up to 10 gb of memory and 6 vcpu cores for lambda functions," https://shorturl.at/CyfSO, (Accessed on 01/17/2022).

[178] J. R. Gunasekaran, P. Thinakaran, M. T. Kandemir, B. Urgaonkar, G. Kesidis, and C. Das, "Spock: Exploiting serverless functions for slo and cost aware resource procurement in public cloud," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 199–208.

[179] "Aws lambda vs ec2: Which to use and when | cbt nuggets," https://www.cbtnuggets.com/blog/certifications/cloud/aws-lambda-vs-ec2-which-to-use-and-when, (Accessed on 01/26/2022).

[180] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 133–146.

[181] D. Mvondo, M. Bacou, K. Nguetchouang, L. Ngale, S. Pouget, J. Kouam, R. Lachaize, J. Hwang, T. Wood, D. Hagimont *et al.*, "Ofc: an opportunistic caching system for faas platforms," in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 228–244.

[182] V. Tolpegin, S. Truex, M. E. Gursoy, and L. Liu, "Data poisoning attacks against federated learning systems," in *European Symposium on Research in Computer Security*. Springer, 2020, pp. 480–501.

[183] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A {Low-Latency} online prediction serving system," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 613–627.

[184] A. Fuerst and P. Sharma, "Faascache: keeping serverless computing alive with greedy-dual caching," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 386–400.

[185] J. Li, S. G. Kulkarni, K. Ramakrishnan, and D. Li, "Analyzing open-source serverless platforms: Characteristics and performance," *arXiv preprint arXiv:2106.03601*, 2021.

[186] "Serverless: Develop & monitor apps on aws lambda," https://www.serverless.com/, (Accessed on 07/11/2022).

[187] A. Gujarati, S. Elnikety, Y. He, K. S. McKinley, and B. B. Brandenburg, "Swayam: distributed autoscaling to meet slas of machine learning inference services with resource efficiency," in *Proceedings of the 18th ACM/IFIP/USENIX middleware conference*, 2017, pp. 109–120.

[188] "What is mlops? | nvidia blog," https://blogs.nvidia.com/blog/2020/09/03/what-is-mlops/, (Accessed on 03/09/2022).

[189] "Scopus vs web of science," https://www.internauka.org/en/blog/scopus-vs-web-of-science, (Accessed on 01/25/2022).

[190] P. Di Francesco, I. Malavolta, and P. Lago, "Research on architecting microservices: Trends, focus, and potential for industrial adoption," in *2017 IEEE International Conference on*

Software Architecture (ICSA). IEEE, 2017, pp. 21–30.

[191] I. B. Data and D. A. S. Cities, "The exponential growth of data," *Inside Big Data White paper. Retrieved online at https://insidebigdata. com/2017/02/16/the-exponential-growth-of-data*, 2017.

[192] 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

[193] "Zenops: A distributed learning system integrating communication efficiency and security," *Algorithms*, vol. 15, no. 7, Jul. 2022.

[194] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More effective distributed ml via a stale synchronous parallel parameter server," *Advances in neural information processing systems*, vol. 26, 2013.

[195] A. G. Roy, S. Siddiqui, S. Pölsterl, N. Navab, and C. Wachinger, "Braintorrent: A peer-to-peer environment for decentralized federated learning," *arXiv preprint arXiv:1905.06731*, 2019.

[196] A. Bellet, R. Guerraoui, M. Taziki, and M. Tommasi, "Personalized and private peer-to-peer machine learning," in *International Conference on Artificial Intelligence and Statistics*. PMLR, 2018, pp. 473–481.

[197] 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.

[198] Z. Tang, S. Shi, and X. Chu, "Communication-efficient decentralized learning with sparsification and adaptive peer selection," in *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2020, pp. 1207–1208.

[199] H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless computing: a survey of opportunities, challenges, and applications," *ACM Computing Surveys*, vol. 54, no. 11s, pp. 1–32, 2022.

[200] J. Sampé, G. Vernik, M. Sánchez-Artigas, and P. García-López, "Serverless data analytics in the ibm cloud," in *Proceedings of the 19th International Middleware Conference Industry*, 2018, pp. 1–8.

[201] "A survey of federated learning for edge computing: Research problems and solutions," *High-Confidence Computing*, vol. 1, no. 1, p. 100008, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S266729522100009X

[202] J. Kepner, V. Gadepally, H. Jananthan, L. Milechin, and S. Samsi, "Sparse deep neural network exact solutions," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–8.

[203] N. S. Sattar and S. Anfuzzaman, "Data parallel large sparse deep neural network on gpu," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2020, pp. 1–9.

[204] 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.

[205] A. Krizhevsky and G. Hinton, "Convolutional deep belief networks on cifar-10," *Unpublished manuscript*, vol. 40, no. 7, pp. 1–9, 2010.

[206] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[207] B. Koonce and B. Koonce, "Mobilenetv3," *Convolutional Neural Networks with Swift for Tensorflow: Image Recognition and Dataset Categorization*, pp. 125–144, 2021.

[208] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[209] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "Qsgd: Communication-efficient sgd via gradient quantization and encoding," *Advances in neural information processing systems*, vol. 30, 2017.

[210] X. Ma, M. Qin, F. Sun, Z. Hou, K. Yuan, Y. Xu, Y. Wang, Y.-K. Chen, R. Jin, and Y. Xie, "Effective model sparsification by scheduled grow-and-prune methods," *arXiv preprint arXiv:2106.09857*, 2021.

[211] A. Beznosikov, S. Horváth, P. Richtárik, and M. Safaryan, "On biased compression for distributed learning," *arXiv preprint arXiv:2002.12410*, 2020.

[212] P. Zhou, Q. Lin, D. Loghin, B. C. Ooi, Y. Wu, and H. Yu, "Communication-efficient decentralized machine learning over heterogeneous networks," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 384–395.

[213] S. S. Sandha, W. Cabrera, M. Al-Kateb, S. Nair, and M. Srivastava, "In-database distributed machine learning: demonstration using teradata sql engine," *Proc. VLDB Endow.*, vol. 12, no. 12, p. 1854–1857, aug 2019. [Online]. Available: https://doi.org/10.14778/3352063.3352083

[214] "Redisai - a server for machine and deep learning models," https://oss.redis.com/redisai/#quick-links, 2023, accessed: 06/20/2023.

[215] "Oracle report scaling r to the enterprise," https://www.oracle.com/a/otn/docs/bringing-r-to-the-enterprise.pdf, 2019, accessed: 06/20/2023.

[216] J. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li *et al.*, "The madlib analytics library or mad skills, the sql," *arXiv preprint arXiv:1208.4165*, 2012.

[217] M. E. Schüle, H. Lang, M. Springer, A. Kemper, T. Neumann, and S. Günnemann, "In-database machine learning with sql on gpus," *33rd International Conference on Scientific and Statistical Database Management*, 2021.

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

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

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

[221] 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.

[222] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," in *Concurrency: the works of leslie lamport*, 2019, pp. 203–226.

[223] "Cross-region replication in azure | microsoft learn," https://learn.microsoft.com/en-us/azure/reliability/cross-region-replication-azure, (Accessed on 05/29/2023).

[224] "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).

[225] 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.

[226] 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.

[227] 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.

[228] "Aws kms concepts - aws key management service," https://docs.aws.amazon.com/kms/latest/developerguide/concepts.html.

[229] "Redisai - a server for machine and deep learning models," https://oss.redis.com/redisai.

[230] 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.

[231] 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.

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

[233] 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.

[234] 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.

[235] R. Anthony, *Systems programming: designing and developing distributed applications*. Morgan Kaufmann, 2015.

[236] 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.

[237] "Lambda quotas - aws lambda," https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html, (Accessed on 06/24/2023).

[238] 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.

[239] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.

[240] 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.

[241] 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.

[242] "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).

[243] "Serverless computing – aws lambda pricing – amazon web services," https://aws.amazon.com/lambda/pricing/, (Accessed on 04/11/2024).

[244] "Create estimate: Configure aws lambda," https://calculator.aws/#/createCalculator/Lambda, (Accessed on 04/11/2024).