Towards Workload-aware Efficient Machine Learning Systems

Redwan Ibne Seraj Khan

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Ali R. Butt, Chair
Kirk W. Cameron
Bo Ji
Xun Jian
Yue Cheng

February 14, 2025
Blacksburg, Virginia

Keywords: Machine Learning, Deep Learning, Federated Learning, High Performance Computing, Cloud Computing, Storage Systems, Data Storage, Data Management, Machine Learning Systems, Job Scheduling, Resource Management, MLSys, SysML, Efficiency, Flexibility
Copyright 2025, Redwan Ibne Seraj Khan

Towards Workload-aware Efficient Machine Learning Systems

Redwan Ibne Seraj Khan

ABSTRACT

Machine learning (ML) is transforming various aspects of our lives, driving the need for computing systems that efficiently support large-scale ML workloads. As models grow in size and complexity, existing systems struggle to adapt, limiting both performance and flexibility. Additionally, ML techniques can enhance traditional computing tasks, but current systems lack the adaptability to integrate these advancements effectively.

Building systems for running machine learning workloads, and running workloads using machine learning - both require a careful understanding of the nature of the systems and ML models. In this dissertation we design and develop a series of novel storage and scheduling solutions for ML systems by bringing attention to the unique characteristics of workloads and the underlying system. We find that by designing ML systems that are finely tuned to workload characteristics and underlying infrastructure, we can significantly enhance application performance and maximize resource utilization.

In the first part of this dissertation (Ch- 3), we analyze popular ML models and datasets, uncovering insights that inspired SHADE, a data-importance-aware caching solution for ML. The second part of this dissertation (Ch- 4) proposes to leverage system characteristics of hundreds of client devices along with the characteristics of the samples within the clients to design novel sampling, caching and client scheduling mechanisms to tackle the data and system heterogeneity among client devices and thereby fundamentally improve the performance of federated learning using edge devices in the cloud. The third part of this dissertation (Ch- 5) proposes to leverage multi-agent LLM application and user request characteristics to design an efficient request scheduling mechanism that can serve clients in multi-tenant environments in a fair and efficient manner while preventing abuse.

My dissertation demonstrates that leveraging workload-aware strategies can significantly enhance the efficiency (e.g., reduced training time, increased throughput, lower latency) and flexibility (e.g., improved ease of use, deployment, and programmability) of machine learning systems. By accounting for workload dynamicity and heterogeneity, these principles can guide the design of next-generation ML systems, ensuring adaptability to emerging models and evolving hardware technologies.

Towards Workload-aware Efficient Machine Learning Systems

Redwan Ibne Seraj Khan

General Audience Abstract

Machine learning (ML) has become an integral part of our daily lives, powering applications from virtual assistants to medical diagnostics. As ML models grow larger and more complex, the systems that run them must evolve to keep pace. This dissertation explores how we can build more efficient and adaptable computing systems to support large-scale ML workloads.

Traditional computing systems often struggle to accommodate the ever-changing demands of ML applications. Similarly, ML techniques can be leveraged to improve the performance of non-ML workloads, but existing systems lack the flexibility to integrate these advancements seamlessly. This research tackles both challenges: designing systems optimized for ML workloads and enhancing traditional systems using ML-driven insights.

By designing intelligent, workload-aware strategies, this research demonstrates substantial improvements in the speed, efficiency, and flexibility of ML systems. These principles will help shape the next generation of computing infrastructure, ensuring that future ML models and applications can be deployed smoothly, regardless of scale or complexity.

Dedication

Dedicated to my family—whose unwavering support, boundless motivation, endless encouragement, and unconditional love have made this journey possible.

Acknowledgments

Before I began drafting this dissertation, I had envisioned countless times how I would express my gratitude to the many individuals who have supported, encouraged, and accompanied me through the highs and lows of my Ph.D. journey. Now, that moment has arrived. With deep appreciation and heartfelt thanks, I extend my gratitude to the following incredible individuals!

I would first like to express my deepest gratitude to my advisor, Ali R. Butt. My journey with Ali began in 2019, during the graduate recruiting weekend at Virginia Tech. At the time, I was searching for an advisor for my PhD, and by sheer luck, I found myself sitting beside Ali during a lunch session. That conversation would turn out to be one of the most pivotal moments in my academic career. Ali shared stories of his conference travels and introduced me to the fascinating world of computer systems research. I was already interested in Machine Learning, and our discussion sparked the idea of pursuing research at the intersection of ML and Systems—a direction that would ultimately define my PhD. Ali's generosity and belief in my potential led him to recommend me for a GTA position, allowing me to embark on this transformative journey. During the course of my Ph.D. he has been more than just an advisor; he has been a mentor in the truest sense, providing the perfect balance of guidance and independence. Ali gave me the space to grow from a graduate student into an independent researcher, instilling in me the confidence to tackle challenging problems, think critically, and push the boundaries of my field. Under Ali's mentorship, I have learned invaluable skills—how to write compelling papers and proposals, deliver engaging presentations, build meaningful collaborations, and navigate the intricacies of research logistics. He has been a steady pillar of support, always encouraging me to see the bright side of research, even in the face of rejections. His unwavering confidence in me has been a source of motivation, inspiring me to persevere and embrace every challenge as an opportunity to learn and improve. There are far too many things to list when it comes to what I have gained from Ali's mentorship. More than just shaping my academic career, he has profoundly influenced my mindset, work ethic, and approach to research. I will always be grateful for his guidance, patience, and belief in me. Without Ali, this journey would have been far more difficult, and I owe much of where I stand today to his mentorship and support.

My special thanks goes to Yue Cheng, whose mentorship has played an instrumental

role in shaping my Ph.D. journey. Since the summer of 2020, Yue has been more than just a mentor—he has been a source of inspiration and guidance, profoundly influencing the way I think about research. He is one of the most talented data systems researcher I have had the privilege to work with, and his approach to tackling complex research problems has fundamentally shaped my own. Yue taught me how to think critically about research—how to uncover the true novelty and motivation behind a problem, ask the right questions, and effectively communicate research findings. His mentorship has significantly refined my analytical and organizational skills, and I have benefited immensely from our countless discussions over the years. I still vividly remember our Zoom sessions where he meticulously sketched out different experiment designs, brainstorming ways to strengthen our contributions. His insightful feedback has consistently elevated the quality of our work, and his far-reaching vision and relentless pursuit of meaningful research have always been a source of motivation for me. Perhaps one of the most valuable lessons I learned from Yue is how to effectively "sell" a research idea—crafting compelling narratives that make an impact. His attention to detail, strategic thinking, and dedication to high-quality research have left a lasting impression on me. I would not be the researcher that I am today without Yue's help and guidance. Yue is the MVP in any collaborative work. I am Yue's big fan and I constantly get inspired by his on-going research. I hope I can pass down whatever I learned from him to my future mentees. For all of this, I owe him a heartfelt thank you!

I would like to thank Arnab K. Paul for being a supportive and caring mentor when I first joined the DSSL lab. I first met Arnab during the graduate recruiting weekend at VT in 2019, and it was through him that I developed a strong interest in joining the DSSL lab. From the very beginning, Arnab played a crucial role in guiding me through my first project, and his mentorship had a lasting impact on my research journey, significantly shaping my core research projects. Arnab's work ethic left a deep impression on me, and beyond research, he was always someone I could turn to for insightful conversations and advice. He was like a big brother—approachable, encouraging, and always willing to help with any questions I had. His support extended far beyond just research; he taught me how to navigate challenges, handle failures and rejections with resilience, and always keep my eyes on the bigger picture. No matter the setbacks, Arnab reminded me that better opportunities were ahead, and his optimism and encouragement were truly invaluable. Thank you, Arnab, for your mentorship, support, and unwavering belief in me!

I would like to sincerely thank my Ph.D. committee members—Professors Xun Jian, Bo Ji, and Kirk Cameron—for their invaluable feedback and insightful discussions throughout various stages of my Ph.D. Their guidance has been instrumental in refining my research and strengthening my dissertation. A special thanks to Dr. Xun Jian for his thoughtful insights on my SHADE and FedCaSe projects, and to Dr. Bo Ji for his valuable feedback on the SHADE project. Their feedback improved the quality of those projects.

This dissertation has greatly benefited from the support and collaboration of my colleagues at Microsoft (M365 Research)—Ankur Mallick, Anjaly Parayil, Chetan Bansal, and Kunal Jain. Ankur and Anjaly provided invaluable insights and engaging discussions throughout

my research internship at Microsoft in the summer of 2024, helping shape key ideas and directions. Ankur also helped me immensely in job search and provided me with helpful suggestions to improve my profile. Chetan offered essential guidance on navigating logistical challenges and adapting to the fast-paced dynamics of industry research. I also had the pleasure of working closely with Kunal on evaluating the FairServe project, an experience that was both intellectually stimulating and rewarding. Collaborating with these exceptional individuals has been an enriching experience, and I am truly grateful for their support, mentorship, and camaraderie.

I would like to take this opportunity to express my gratitude to Dr. Wu Feng, who was my first point of contact at Virginia Tech and generously recommended me for a spot at the graduate recruiting weekend in 2019. I feel incredibly fortunate to have worked with some awesome people and an amazing group of labmates at DSSL and outside. Collaborating with Ahmad Yazdani and Yuqi Fu (University of Virginia) on SHADE was a fantastic experience. I had the privilege of working closely with Hadeel, Shruti, and Sabiha on multiple projects - from late night meetings to discussions and brainstorming sessions, our interactions were always insightful and rewarding. Thank you Hadeel, Shruti, and Sabiha for being awesome collaborators! My advisor, Ali, has always encouraged us to grow into independent researchers, and part of that involved assisting him with grant proposals to gain real-world experience in academia. In that regard, Ahmad Faraz has been my constant buddy, brainstorming and co-writing several proposal grants for the lab. It's unfortunate that we didn't get the chance to collaborate on a research project, but I'm hopeful that we will soon! A huge shoutout to my incredible DSSL labmates at Virginia Tech: Debasmita, Shunyu, Yuze, Tanuj, Subil, Abhijit, Anish, Bharti, Diego, Jingoo, Nannan, Nameer, Lucas, Rishika, Hyogi, and Yanlin. Whether it was sharing life updates, discussing research, or just goofing around in the lab and at conferences, our interactions enriched my grad school experience in ways I can't fully put into words. The memories we created—both in the lab and across different conference venues—were truly some of the best moments of my Ph.D. journey.

I am extremely thankful to the wonderful staff in the Computer Science department at Virginia Tech. Special mention needs to be given to Sharon, who has helped me numerous number of times. Teresa - without you none of my conference travels would have been possible. I would also like to thank the Mike, Robert, and the tech staff, for being there always whenever our lab needed technical assistance and printing posters for conferences. I would also like to acknowledge my friends and community members in Blacksburg who have made my stay in Blacksburg enjoyable. Special thanks goes out to Rafid, Ishrat, Rahat, Uzma, Rahi, Fatema, Efaz, Faria, Shawal, and Azmat for their help and support before and after the birth of my first son. I have very fond memories at the Masjid Al Ihsan, especially during Ramadan. I had volunteered to teach Arabic to the kids at the masjid, and those were definitely some of the most encouraging and delightful moments during my Ph.D.

I am immensely grateful to my family, without whom this Ph.D. journey would have been empty and incomplete. My father, Mohammed Serajul Islam Khan, and my mother, Zerina

Akter, have been my unwavering pillars of support, always encouraging my academic pursuits. Their hard work and dedication created the best possible environment for me to learn, grow, and thrive. My brother, Wasif Adnan Khan, played a pivotal role in shaping my academic journey from my childhood through my teenage years. It was his passion and excellence in Computer Science that worked as a major contributing factor towards pursuing this field during my undergraduate studies. My sister, Ramisa Binta Seraj, has filled my life with countless joyful moments, breaking the monotony and keeping me grounded with her infectious energy. I am also deeply thankful to my mother-in-law, Akther Jahan, as well as Jinat and Taspia, for their constant encouragement, support, and the wonderful moments we have shared. A special and heartfelt thanks to my wife, Afrina Tabassum, who has been my rock, my motivator, and my companion through every high and low of this journey. Her unwavering support, selfless sacrifices, and boundless encouragement have been instrumental in keeping me focused and driven toward my goals. Whether it was coursework, research, or everyday life, she stood by my side, making this journey so much more meaningful. Together, we created countless unforgettable memories—random outings, camping trips, hiking, kayaking, skating, jet skiing, parasailing, horse riding, and so many more that I can't possibly list them all. Without her, this Ph.D. would have felt incredibly lonely. As I approached the final stretch of my Ph.D., we were blessed with the most precious blessing—our son, Raaed. His presence has become my greatest source of motivation, reminding me every day to cherish life and strive for something greater. To my family—thank you for everything. This accomplishment is as much yours as it is mine.

Funding Acknowledgement My research was supported by National Science Foundation (CSR-2106634, CCF-1919113/1919075, CNS- 2045680, OAC-2004751, OAC-2106446, SR-2312785, CCF 2318628, CCF 1919075, CMMI 2134689, CNS 2322860), and CS at Virginia Tech. Many thanks to all for making this dissertation possible.

Declaration of Collaboration In addition to my adviser Ali R. Butt, this dissertation has benefited from many collaborators, especially:

- Yue Cheng contributed to the work included in Chapter 3, Chapter 4, and Chapter 5 of this dissertation that resulted in [1], [2], and [3].
- Arnab K. Paul contributed to the work included in Chapter 3 and Chapter 4 of this dissertation that resulted in [1], [2], and [4].
- Xun Jian have contributed to the work included in Chapter 3 and Chapter 4 of this dissertation.
- Bo Ji have contributed to the work in Chapter 3 of this dissertation.
- Ahmad Hossein Yazdani and Yuqi Fu contributed to the work in Chapter 3. They made valuable contribution in both the motivation and evaluation of SHADE.
- Kunal, Ankur, Anjaly, Haiying, and Chetan along with Microsoft M365 Research team contributed to the work in Chapter 5. Kunal contributed a significant amount in the analysis and evaluation part of the work in Chapter 5.

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.1.1 Workload-aware Caching in Machine Learning Training . . . 2

1.1.2 Workload-aware Client Scheduling in Machine Learning Training . . 3

1.1.3 Workload-aware Job Scheduling & Fairness in LLM inference . . . 3

1.2 Research Contributions . . . 4

1.3 Terminology . . . 5

1.4 Dissertation Organization . . . 6

2 Background **7**

2.1 Workload-aware Caching in ML Training . . . 7

2.2 Workload-aware Client Scheduling in ML Training . . . 9

2.3 Workload-aware Job Scheduling & Fairness in LLM Inference . . . 10

3 Workload-aware Caching in Machine Learning Training **14**

3.1 Introduction . . . 14

3.2 Background & Motivation . . . 17

3.2.1 DL Training with Importance Sampling . . . 17

3.2.2 Exploiting Importance Sampling . . . 18

3.3 System Design . . . 21

3.3.1 Challenges . . . 21

3.3.2 SHADE Overview . . . 23

3.4 Implementation . . . 30
3.5 Evaluation . . . 31
3.5.1 Experimental Setup . . . 31
3.5.2 Cache Hit Ratio . . . 32
3.5.3 Accuracy vs. Time . . . 33
3.5.4 Throughput . . . 35
3.5.5 Minibatch Load Time . . . 36
3.5.6 End-to-End System Comparison . . . 37
3.6 Related Work . . . 39
3.7 Summary . . . 40

4 Workload-aware Client Scheduling in Machine Learning Training 41
4.1 Introduction . . . 41
4.2 Background & Motivation . . . 43
4.2.1 Distributed Machine Learning . . . 44
4.2.2 Federated Learning . . . 44
4.2.3 Exploiting Client Experience . . . 45
4.2.4 Exploiting Sample Experience . . . 46
4.3 FedCaSe Design . . . 49
4.3.1 Challenges . . . 49
4.3.2 FedCaSe Overview . . . 50
4.4 Implementation . . . 58
4.5 Evaluation . . . 58
4.5.1 Experimental Setup . . . 58
4.5.2 Impact on Read Hit Ratio (RHR) . . . 59
4.5.3 Impact on Accuracy Improvement . . . 61
4.5.4 Impact on Rounds . . . 64
4.6 Assumptions and Limitations . . . 68
4.7 Summary . . . 68

5 Workload-aware Job Scheduling & Fairness in LLM Inference 69
5.1 Introduction 69
5.2 Production Workload Analysis 71
5.2.1 Impact of User Behavior 71
5.2.2 Impact of Application Behavior 73
5.2.3 Impact of LLM Agents and System Prompts 74
5.2.4 Impact of RPM Throttling 75
5.3 FairServe Design 77
5.3.1 FairServe Overview 78
5.3.2 Overload & Interaction-driven Throttling (OIT) 79
5.3.3 Weighted Service Counter (WSC) 81
5.4 Evaluation 82
5.4.1 Demonstration of Fairness Objective: Equity 84
5.4.2 Curbing Abusive Behaviour 85
5.4.3 Reducing Queueing Delays 85
5.4.4 Improvement in Throughput and Latency 86
5.4.5 Improvement in Served Experience. 86
5.4.6 A Case Study: Comparing Diverse Apps 87
5.5 Related Work 88
5.6 Summary 89

6 Conclusion 90
6.1 Summary & Impact 91
6.2 Future Directions 93
6.2.1 Client-Side Optimizations for GenAI Systems: Context Aware Caching 93
6.2.2 Server-Side Optimizations For GenAI Systems: Model Caching and Routing. 95
6.2.3 Cross-Stack Optimization for Resource Management 95

Bibliography **95**

List of Figures

2.1 Example of LLM inference. 11

2.2 Example of an LLM interaction. 12

3.1 Training throughput and time comparison of a single job using 2 nodes and 8 GPUs with ResNet-18. Remote storage comprises of SSDs on a BeeGFS server. 15

3.2 Frequency of samples accessed across different epochs in default single process importance sampling (CIFAR-10). 18

3.3 Distribution of data importance as the number of epochs increases in single process default importance sampling on the CIFAR-10 dataset. Data importance is the ability of a sample to contribute towards improving the accuracy of the model. 19

3.4 Comparison of loss and accuracy convergence of ResNet-18 model using single process default importance sampling against baseline training on the CIFAR-10 and CIFAR-100 datasets. 20

3.5 Comparison of different caching policies during ResNet-18 model training over the CIFAR-10 dataset. Working Set Size (WSS) denotes the percentage of cached dataset. 21

3.6 SHADE architecture overview. (b) In illustration of how SHADE's components interact in a single epoch. 22

3.7 Comparison of the read hit ratio of various caching policies and cache sizes. The sh_ prefix denotes a baseline version of SHADE that uses the coarse-grained importance. SHADE denotes our contribution, SHADE, with all techniques enabled. WSS denotes working set size. 33

3.8 Accuracy improvement rate of SHADE against baseline LRU when different portions of the entire dataset is cached (denoted by the percentages). 34

3.9 Throughput of SHADE against baseline LRU when different portions of the entire dataset is cached (denoted by the percentages). 35
3.10 GPUs' minibatch load time when training ResNet-50. Percentages denote the amount of cached dataset. 36
3.11 Comparison of SHADE and NoPFS [5]. Percentage denotes the percentage of cached dataset. 37
3.12 Comparison of the read hit ratio of different caching policies at 20% WSS of CIFAR-10. 38

4.1 Training time comparison between clients using memory cache and flash storage. 42
4.2 Improvement of test accuracy for three different clients using different samples for training from the EMNIST dataset. 43
4.3 CDF of the number of clients selected during random scheduling on the FEMNIST dataset. 46
4.4 Global read hit ratio (RHR) of different caching policies in a heterogenous FL setup of 1000 clients. 48
4.5 Heterogeneous memory creates problems in efficient utilization of memory cache of FL clients. 48
4.6 (a) FedCaSe architecture overview. (b) An illustration of how FedCaSe's components interact in a single FL round. 49
4.7 Sample sizes of clients differ greatly. Sample sizes normalized with respect to their sizes across x-axis. 59
4.8 Memory cache space of the clients differ greatly. X-axis denotes the Client IDs. 60
4.9 Global read hit ratio (RHR) of different caching policies in a heterogenous FL setup of 2800 clients. Ablation study on three components of FedCaSe (C: Caching, S: Sampling, and Sched: Scheduling). 60
4.10 FedCaSe outperforms other state-of-the-art policies in local RHR at the scale of thousands of heterogeneous clients. 61
4.11 Accuracy improvement of FedCaSe vs. `vanilla` configurations using ResNet-18 on FEMNIST. 62
4.12 Accuracy improvement rate of FedCaSe vs. Oort. Training ResNet-18 on FEMNIST for 150 rounds. 63

4.13 Adaptivity of RO policy in scheduling experienced clients. 65
4.14 Experienced clients get scheduled for participation in future training more frequently in FedCaSe. 65
4.15 Sensitivity analysis on the impact (Top-1 accuracy and training time) of placing weight, α on I/O utility when determining client experience. 66
4.16 Accuracy improvement rate of FedCaSe vs. Oort with respect to rounds. Training ResNet-18 on CIFAR-10. 66
4.17 FedCaSe vs. Oort round time improvement using different cache policies during training MobileNetV2 on FEMNIST. 67
4.18 Accuracy improvement rate of FedCaSe vs. Oort with respect to rounds. Ablation study on FedCaSe's model and I/O utility fucntions. Training ResNet-18 on FEMNIST. 67

5.1 Users' varying RPM and TPM across apps make prior policies ineffective for ensuring fairness and deterring abusive behavior. 72
5.2 Token counts differ across applications, suggesting that LLM scheduling must consider variations in user applications. 73
5.3 The variability of LLM calls across apps must be considered to reduce latencies and queueing delays in multi-agent apps. 74
5.4 System prompts lead to varying characteristics in total tokens of each interaction and the number of output tokens. 75
5.5 A throttling system without user and application-awareness can lead to resource wastage. 76
5.6 The design overview of FairServe. 78
5.7 (a) User request characteristics and their arrival patterns for the illustrations in (b)-(d). (b) Illustration of how FairServe ensures equitable fairness, reduces latency and queueing delays while curbing abusive behavior during processing user requests. (c) & (d) Illustration of two scenarios on how equality-based fairness schemes (e.g., VTC [6]) can lead to abuse, resource wastage, longer latencies, and degraded user experience. 79
5.8 FairServe does not incur resource wastage as it is multi-agent LLM interaction-aware. 83
5.9 FairServe (FS) maintains a higher throughput and a lower TTFT (time-to-first-token) latency across apps. 86

6.1 Future Directions Overview 94

List of Tables

4.1 Contribution towards accuracy improvement is different in each round for different client samples. . . . 47

4.2 The improvement of FedCaSe in number of rounds clients get scheduled in different ranges over vanilla and Oort. $\geq$ X means a single client gets called $\geq$ X times throughout training. . . . 64

5.1 Comparison of FAIRSERVE and current methods. . . . 70

5.2 Distribution of graph sizes and tokens in graphs 76

5.3 Synthetic benchmark setup 84

5.4 Number of responses received and TPOT latency. . . . 84

5.5 Throughput and latency during real-time processing. . . . 87

5.6 Case study: Comparison of service provided to specific applications by each strategy 88

Chapter 1

Introduction

Modern world is increasingly becoming dependent on Artificial Intelligence (AI). Along with solving complex problems in a wide variety of domains, it is also helping in making our lives easier. However, such convenience is not going to come free. As using AI requires processing huge amounts of data, it is both compute-intensive and data-intensive. To meet the demands of compute and data, large-scale accelerator-based clusters and other systems are increasingly being adopted in research and industry. The vast heterogeneity in the compute and communication power of the systems, in the available tera and petabytes of data, in the models deployed has made running AI applications challenging. Without proper guiding frameworks and solutions, AI applications and the systems used to run them often end up with subpar performance and resource under-utilization. As such, we end up in a place where the AI or machine learning(ML) application requirements need to be matched with available system resources in order to have an optimized performance.

1.1 Motivation

The area of AI is vast, with a wide range of applications targeting numerous areas including but not limited to autonomous systems, medical imaging, surveillance, chatbots, voice assistants, financial forecasting, stock market analysis, weather prediction, signal processing, fraud detection, image synthesis, data augmentation, content creation, etc. The difference in the nature of the applications has given rise to many broad areas of ML including computer vision (CV), natural language processing (NLP), Generative models, reinforcement learning, etc. Larger and better models are continously getting deployed to carry on tasks in these domains.

Different models for different tasks vary in characteristics related to the number of parameters, layers, weights, etc. Moreover, the data necessary for carrying on these tasks are hugely different from one another. For example, NLP requires text data, computer

vision requires video and image data, etc. Hence, along with the difference in model characteristics, there are also notable differences in data characteristics among the huge pool of available data. Furthermore, the system resources deployed to run these tasks having differing data and model characteristics also differ in capabilities, e.g., different CPU, GPU, storage configurations. With the advent of federated learning (FL), allocating resources and clients to carry on training tasks have become even more complicated. Without careful investigation, running such AI applications inevitably leads to wastage of resources and energy.

To tackle above challenges, this dissertation proposes, designs, and implements a suite of novel techniques, algorithms, and frameworks that optimize AI systems by aligning them with the underlying workload and system characteristics. This dissertation selects three application scenarios—high-performance large-scale homogeneous clusters, high-performance large-scale heterogeneous clusters and federated heterogeneous client devices targeting two general modern workloads—AI/DL workloads and parallel large-scale cluster workloads. This dissertation aims to enhance application and system performance by developing workload-aware solutions tailored to dynamic computational demands.

The next section provides a concise overview of the research challenges, proposed solutions, and key evaluation results presented in this dissertation.

1.1.1 Workload-aware Caching in Machine Learning Training

Modern world is the age of AI revolution. This revolution is supported through the usage of huge AI/DL models trained on large corpuses of data spanning tera and peta bytes. Large-scale homogeneous clusters spanning multiple number of nodes are typically employed for training which fetch data from remote storage nodes. Along with lots of computations, such a setup poses a major I/O bottleneck as data samples need to be fetched in every iteration throughout the training. This bottleneck becomes specially pronounced when the compute time obtained from powerful accelerators is drastically lower than the time required for I/O from remote storage hosted on a filesystem. In practice, it has been observed that I/O takes around 85-90% of the total training time [1, 7, 8]. As a result, not only does it lead to compute resource underutilization but also increased training time and energy costs. Hence, there is a clear need to reduce the I/O time to improve the training performance.

In the first part of this dissertation, we propose SHADE— a **SHA**red, **D**istributed, and **E**lastic data characteristic-aware and DL training-aware caching solution to mitigate the I/O bottleneck during DL training. SHADE puts forward a novel data-importance aware caching solution that finds the most important samples during DL training, caches them in-memory during run time according to the available memory capacity, fetches them repetitively from cache instead of the remote storage, and thereby reduces the DL training time. SHADE assigns relative importance scores to each individual data sample, and

dynamically adjusts the cache according to the changing patterns of importance of the data samples throughout the training. To address the issue of making models biased, SHADE proactively monitors the change of accuracy and loss and thus it manages to reach training convergence faster while not sacrificing the accuracy. Evaluations on real-world traces demonstrate that SHADE is able to achieve accuracy convergence 3.3× faster compared to a baseline LRU caching policy.

1.1.2 Workload-aware Client Scheduling in Machine Learning Training

AI systems of today are not confined to in-house cluster setups. With the vast pool of data that is being generated by people all around the world, systems for running AI has evolved and made its way into people's cellphones and tablets. This new field of AI, namely federated learning, is a process where multiple entities or clients having varying compute and commute abilities take part in training a single global model. These entities, known as executors have heterogeneous datasets both in terms of size and variety. This is in contrast to existing modes of DL training where there exists no data heterogeneity among the different participating clients. Due to the existence of heterogeneity in terms of available data, compute, and communication abilities between the available clients, performing training becomes much more challenging than usual. Furthermore, as clients have to fetch the data of varying sizes from available flash memory, I/O quickly becomes a bottleneck. Hence, there is a need for proper policies to address these limitations to enhance the performance of federated learning.

In the second part of this dissertation, we propose FedCaSe—a caching and client selection mechanism that performs context-aware data sampling at the client data level to derive I/O benefits and thereby leverage it to adaptively take decisions on client selection in each round. Experimental evaluation on real-world datasets show that FedCaSe can increase the global read hit ratio (RHR) across all clients by up to 81.7×.

1.1.3 Workload-aware Job Scheduling & Fairness in LLM inference

The rapid expansion of multi-tenant platforms serving personalized large language models (LLMs) has revolutionized applications like interactive question answering, summarization, and coding assistance. However, this surge in demand presents critical challenges: abusive user behavior can overwhelm systems, leading to unfair service allocation, resource wastage, and degraded user experiences. Existing rate-limiting solutions, such as request-per-minute (RPM) limits [9, 10], fail to address these issues comprehensively. They often throttle legitimate requests during low-load periods, causing resource underutilization, incomplete responses in multi-agent LLM workflows, and inefficiencies that hinder platform scalability [11].

To address these limitations, the third part of this dissertation introduces FairServe, a request scheduling system designed to ensure fair, abuse-free, and efficient LLM serving. Extensive experiments on real-world testbeds show that FairServe can increase throughput by $1.03 - 1.75\times$ (with 0% token wastage) across different applications while curbing abusive behaviour and enabling better service to users by $99.45 - 100\%$.

1.2 Research Contributions

From the above three aspects, *we demonstrate in this dissertation that we can improve the performance of AI applications by closely matching the needs or available characteristics of these applications with the available characteristics of system resources.*

Overall, this dissertation introduces novel systemic and algorithmic approaches to address inefficiencies and inflexibilities in AI system execution and scheduling. Next we highlight the key research contributions of this dissertation.

Exploiting Training and Data Characteristics for ML Training in HPC Environments

In this work, we propose a system—SHADE for training DL workloads that leverages insights from continuously changing training parameters and importance of available data samples to increase the training performance through a distributed cache built across all of the training nodes in a cluster. First, we perform a detailed analysis of the existing training patterns that we observe during running DL workloads. Our analysis reveals useful insights about the potential of exploiting the importance of data samples involved in training, i.e, the characteristics associated with the data samples. Second, we leverage the insights to develop a novel policy of priority-based repetitive data sampling and ranking, known as *PADS*. Third, we develop a novel data caching protocol, known as *APP* that exploits the repetitive samples provided by *PADS* to form a cache and dynamically updates it during training time. Fourth, we compare our proposed caching and eviction strategy against state-of-the-art caching and eviction policies like least-recently-used (LRU), least-frequently-used (LFU), priority-based eviction, priority and frequency-based eviction, and random and show that SHADE can get 4.5$\times$ better hit rates in available cache size and upto 3$\times$ better accuracy convergence rate. We also show that SHADE is able to out-perform state-of-the-art caching policies like CoorDL [12] and Quiver [13] by 3.6$\times$.

Exploiting Client & Data Characteristics for ML Training in the Cloud

In this work, we propose FedCaSe—a system that leverages the unique system-level characteristics (i.e., computation and communication abilities) of hundreds and thousands participating clients along with their local data sample characteristics (e.g., size, context) to derive caching and selection decisions to ultimately improve the training time and accuracy convergence. First, we conduct a thorough analysis on the training convergence patterns of client image samples having heterogeneous image sizes and cache working set sizes

and hence motivating the need for a unique caching and client selection protocol. Second, we design a caching protocol based on the context of the available data of each individual client. Third, we quantify the benefits obtained from caching and data sampling for each client through a new metric called experience and use that metric to adaptively drive the client selection and scheduling procedure in federated learning. Fourth, we experiment with representative workloads and policies show that compared to the state-of-the-art, FedCaSe improves the experienced client participation up to 29.1×, improves the global read hit ratio (RHR) across all clients by up to 81.7× (locally up to 318.58×), and thus ensures accuracy improvement rate up to 2.06× faster based on wall clock time, up to 1.4× faster based on number of rounds while keeping the round duration up to 2.4× less.

Exploiting Multi-agent AI Application Characteristics for Multi-tenant LLM Inference

In this work, we propose FairServe–a fair scheduling system based on the principle of equity and abuse prevention mechanism for serving user requests belonging to multi-agent AI applications. First, we conduct large-scale analysis on millions of requests across 34 applications on a Microsoft's CoPilot platform, revealing intricate characteristics of modern LLM applications and guiding future research. Second, we introduce a novel approach, Overload and Interaction-driven Throttling (OIT), for throttling LLM requests to curb abusive behavior. Third, we design a novel mechanism, Weighted Service Counter (WSC) for scheduling LLM requests to ensure fairness amid diverse applications. Fourth, we integrate FairServe in a open-source LLM serving platform [14] using state-of-the-art iteration-level continuous batching [15] and compare against a series of policies for LLM serving fairness. Our results on a testbed of over a thousand users show that FairServe reduces waiting queue delays by $10.67 - 93\times$, decrease latency (time-to-first-token) by $1.03 - 1.06\times$, increase throughput by $1.03 - 1.75\times$ (with 0% token wastage) across apps while curbing abusive behaviour and enabling better service to users by $99.45 - 100\%$.

1.3 Terminology

Note about terminology for readers of this dissertation: throughout the dissertation the readers will come across the terms memory-cache/cache and flash cache. Memory cache is a software-managed, application-level cache that stores data in the system's main memory, enabling fast access. The flash cache is also software-managed application-level cache but stores the cached data on flash-based SSDs. Throughout the different works, we target the following metrics for denoting the performance improvement — *accuracy convergence time, training time or workload makespan or completion time, throughput, latency, read-hit ratio, minibatch load time, queueing delays*. For prediction model, we focus on the – *Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and Accuracy*. For cost efficiency, we focus on the *required cost* to run a workload. For resource efficiency we focus on the *utilization* of the assigned resource. Additionally, we emphasize *ease of use, deployment, programmability,*

and fairness to enhance accessibility and integration.

1.4 Dissertation Organization

The rest of the dissertation is organized as follows. In Chapter 2 we introduce the background technologies and state-of-the-art related work that lay the foundation of the research conducted in this dissertation. Chapter 3 presents a deep learning caching solution, SHADE by exploiting data sample characteristics and training patterns. Chapter 4 presents FedCaSe, a novel unified client scheduling, sampling and caching solution for improving the performance of edge devices in the cloud during Federated Learning. Chapter 5 presents FairServe, a fair and efficient system for scheduling user requests belonging to multi-agent LLM applications. Chapter 6 concludes this dissertation and discusses the future directions.

Chapter 2

Background

This chapter provides the necessary background for our dissertation, which explores workload-aware strategies to enhance existing machine learning systems. By aligning ML workloads with system resources, we aim to bridge performance gaps and improve efficiency. We summarize state-of-the-art research in this domain and highlight the effectiveness, novelty, and impact of our proposed techniques and algorithms.

2.1 Workload-aware Caching in ML Training

Distributed Machine Learning. With the rise in the adoption of machine learning it is no longer limited to single-node or single process settings. At present although it might be run in a single node for research purposes, it is often, if not always, run with multiple accelerators. This procedure of running machine learning workloads using multiple accelerators or nodes to satisfy the computational requirements is known as distributed machine learning. These kind of distributed training are often conducted in large in-house clusters formed of computational nodes having the same kind of configurations. Moreover, the dataset used is also partitioned in equal chunks across the participating nodes. In short, often there does not exist any heterogeneity in terms of data quantity and quality along with system configuration.

There are mainly three types of distributed Deep Learning training techniques: *data-parallel training* [16], *model-parallel training* [17], and *pipeline parallelism* [18] that combines data-parallel and model-parallel training. While this section focuses mainly on data-parallel training based on Stochastic Gradient Descent (SGD), our approach is applicable to other training methods as well.

A Deep Neural Network (DNN) model consists of multiple layers of computation units whose output is the input for subsequent units. DNN model training consists of a forward

propagation method, which sequentially moves information related to the input data through all model layers and generates a prediction. For example, in an image recognition application, image pixels information is moved through the layers for predicting image contents. To generate the prediction, DL defines a cost/loss function with respect to the forward propagation output and ground truth labels. The DL process aims to minimize the cost function through a process of increasing or decreasing the weights of the outputs of the intermediary layers of the model so that it can improve its prediction. This step is known as backward propagation, which adjusts the parameters of the DL model starting from the outermost layer back up to the input layer through a technique known as gradient descent optimization. Gradient descent adjusts the parameters in the opposite direction of the gradient. SGD is a stochastic approximation of gradient descent optimization; instead of calculating the gradient from the entire data set, SGD randomly selects a subset of training data samples from the entire dataset to reduce computation cost.

In a typical data-parallel, SGD-based training, the whole training dataset is partitioned and processed in parallel by multiple GPU devices. Each GPU has a replica of the same DNN model, which is iteratively synchronized with other GPUs using centralized communication techniques, e.g., parameter server [19]) or decentralized communication techniques, e.g., all-reduce [20].

I/O Characteristics of Data-Parallel DL Training. DL training applications feature unique characteristics that differentiate them from conventional data-intensive applications such as big data analytics [21, 22] and web applications [23, 24]. A DL training job typically runs multiple epochs, with each epoch consuming the entire training dataset once in a random permutation order. Each epoch is further divided into multiple batches. At the beginning of processing a batch, each GPU process loads a randomly-sampled bulk (i.e., a minibatch) of training data whose size is configurable. These behaviors lead to highly-concurrent, read-only, repetitive, and totally random I/O accesses. Therefore, a common belief is that such I/O patterns are not cache-friendly to traditional caching policies that exploit recency and/or frequency-based data locality, such as the widely used LRU, LFU, and ARC [25].

Importance Sampling for Optimizing System Efficiency. Importance sampling is a data selection technique that prioritizes samples for each training round based on a predefined importance metric. Recent works have leveraged importance sampling to enhance the system efficiency of deep learning workloads [26, 27]. iCACHE [27] employs importance sampling for deep learning training caches but lacks a rank-based relative scoring mechanism and SHADE 's *PADS* sampling approach, potentially leading to a lower cache hit ratio. Additionally, iCACHE relies on substitutability in case of cache misses, which may affect training accuracy convergence. Mercury [26] optimizes DL training by prioritizing important samples but does not function as an I/O cache. Unlike SHADE, it lacks data replacement and eviction management, limiting its caching applicability.

Caching and Prefetching for Optimizing I/O. CoorDL [12] examines PyTorch's data retrieval process and introduces MinIO caching, where a random subset of data is cached

during the first epoch and retained throughout training without eviction. However, caching random samples fails to yield significant performance improvements.

Several works focus on optimizing the I/O components of deep learning applications. NoPFS [8] employs a prefetching strategy that leverages hardware configurations to make caching decisions based on approximating Belady's MIN. However, in dynamic training scenarios such as hyperparameter tuning [28], sample access patterns shift unpredictably, limiting NoPFS's applicability. SHADE addresses this challenge through dynamic cache management, eliminating reliance on hardware-based configurations.

Hoard [29], Quiver [13], and FanStore [30] introduce global caching layers for GPU clusters to enhance deep learning training performance. DeepIO [31] proposes an entropy-aware method for minibatch selection but lacks cache eviction policies and dataset randomization. DIESEL [32] offers a storage solution with key-value metadata services, task-level caching, and chunk-based shuffling. However, these approaches overlook fundamental data locality for deep learning training (DLT) jobs. In contrast, SHADE leverages importance sampling to enhance data locality, optimizing cache efficiency for DLT workloads.

2.2 Workload-aware Client Scheduling in ML Training

Federated Learning. Federated Learning [33] follows the same paradigm of distributed machine learning in the sense that the computation is not limited to one single client. In this case, the agenda is for multiple clients to train a global model with their respective local datasets and then pass on the trained local models back to a global server which aggregates the local model parameters of the different clients to produce a more powerful global model which will later be used by all of the clients. The idea is that this kind of collaboration can benefit all of the clients — on one hand they do not have to share their private data with others, and at the same time they can also get a powerful model trained on data that was unavailable to them. The main difference between federated learning and traditional distributed learning is however in terms of data heterogeneity, system heterogeneity and network heterogeneity among the participating clients which makes federated learning significantly more challenging in comparison to traditional datacenter-based distributed ML.

Based on the nature of the training participants, Federated Learning can be divided into two categories—cross-silo [34] and cross-device [35, 36]. Cross-silo federated learning is when the participating clients are silos or departments, each having their own unique datasets. Cross-device federated learning is when the participating clients are user devices having their own datasets.

Again, based on the data and feature partitioning, federated learning is divided into three main categories—horizontal [37], vertical [38, 39] and transfer learning [40]. Features are

the characteristics through which data samples can be identified. In horizontal settings, different clients or entities will share the same feature space but will have different samples, i.e., the different samples will be identified by the variations within the same set of features. In vertical federated learning, different entities share a common sample space but a different feature space, i.e., the same samples might be identified using different samples across different entities. Federated transfer learning is a combination of horizontal and vertical federated learning where clients can have different samples as well as features.

Last, based on the nature of the update of the global model, federated learning is of two types—synchronous [41, 42, 43] and asynchronous [44, 45, 46, 47]. In the synchronous setting, the aggregator server holding the global model needs to wait to get the local model updates from all of the clients before it updates the global model. In the asynchronous setting, the aggregator does not wait for all of the clients to send local updates and instead keeps updating the global model as it receives local model updates.

Federated learning can be used in a wide variety of scenarios, including computer vision, speech recognition, language modeling, etc. To tackle the heterogeneity efforts have been made towards guided client selection [48], clustering [49], or improving upon existing ML algorithms [50, 51]. Although these works focus on improving the communication, computation, or client selection of clients, none of these works consider the I/O impact of client samples when in a scaled-up federated setting. Although recent works [1, 13] look at client data sampling, these are limited to in-house homogeneous cluster settings. FedCaSe considers client system efficiency and sample context while sampling and caching data at the scale of hundreds of clients. Through devising a new adaptive composite metric based on client efficiency, it manages to select and schedule clients for training.

2.3 Workload-aware Job Scheduling & Fairness in LLM Inference

Transformer Architecture. Modern LLMs are built on multiple transformer blocks [52]. Each block consists of an attention operation which applies three weight matrices W_K, W_Q, and W_V respectively to the encoded represention of the input tokens, X of an user, to calculate the key K, query Q, and value V respectively associated with the corresponding input prompt.

$$Q = XW^Q, K = XW^K, V = XW^V$$
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{2.1}$$

Next, the K, Q, and V matrices are used to calculate the attention head following Eq. 2.1,

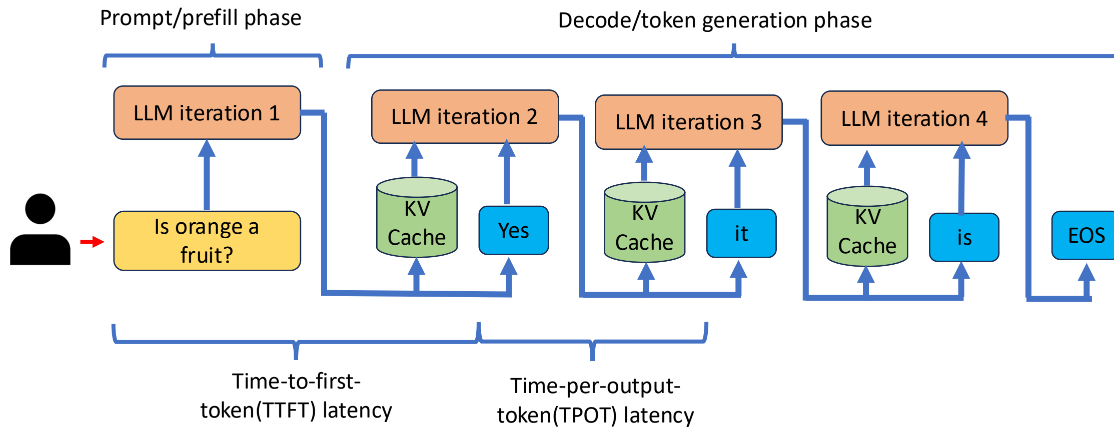


Figure 2.1: Example of LLM inference.

which is later linearly combined with other attention heads to get the multi-head attention (MHA), allowing the model to focus on different parts of the sequence in different representational spaces. After the MHA block, the output undergoes layer normalization and is passed through a position-wise feed-forward network (FFN). The FFN consists of two linear transformations with an activation function in between, allowing further refinement of the information processed by the MHA block. The final output retains the same dimensionality as the input.

LLM Inference Procedure. LLM inference procedure consists of two phases–a prefill phase and a decode phase. In the prefill phase, the LLM processes all input tokens from the user in parallel in each iteration and generates the first output token. Parallel processing makes efficient use of GPU resources. During input token processing, the context from those tokens are computed by the model's attention layers and cached in the KV cache [53].

In the decode phase, the LLM autoregressively generates one output token at a time. The decode phase requires access to the KV cache of all previously processed tokens in order to perform the attention operation. As the model predicts the next token, it appends the newly generated token to the original prompt and updates the KV cache, making it to grow linearly with the number of tokens generated. Once the first token is generated, subsequent token generation only requires the last generated token and the accumulated KV cache as inputs. The decode phase ends when a special end-of-sentence (EOS) token is generated or the request reaches its pre-defined maximum length. The decode phase is primarily memory-bound with low compute utilization, since the model processes one token at a time. Figure 2.1 shows an example of how LLM inference takes place across multiple iterations along with metrics like Time-to-first-token (TTFT) latency and Time-per-output-token (TPOT) latency.

Multi-agent LLM inference. Modern LLM inference applications comprises of complex LLM calls at multiple-levels to generate a response to a user query. User queries can be complex, requiring decomposition into simpler sub-queries. Each sub-query is addressed

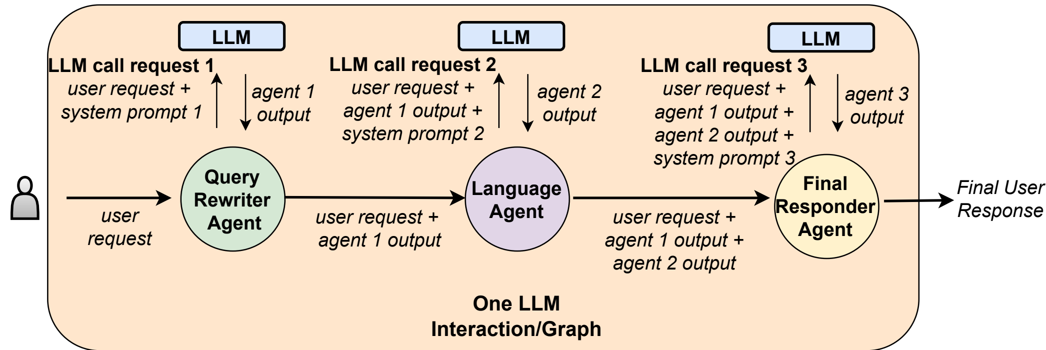


Figure 2.2: Example of an LLM interaction.

by an LLM agent utilizing a specialized LLM. Once the answers to these sub-queries are generated, they are aggregated, and a contextualized query is constructed. This refined query is subsequently processed by another LLM to produce the final response for the user. In short, multiple LLM agents, each in charge of a separate, simpler task, can collaborate together to perform a larger, but more complex task [54, 55, 56]. For example, Figure 2.2 shows how multiple LLM agents can collaborate together to generate a single user response. We denote the entire graph of LLM calls formed to generate a user response as one **LLM interaction**. **System prompts** are pre-defined prompts or instructions sent to the language model to establish the context, rules, or behavior before processing a request.

LLM Inference in Multi-tenant Environments. In multi-tenant environments, sequentially handling user requests increases latency and GPU underutilization, necessitating parallel processing through either (1) request-level batching or (2) iteration-level (continuous) batching. In request-level batching, a batch is selected from waiting requests and processed until completion. Inference engines like FasterTransformer [57] and Triton [58] use this method, which simplifies operations but wastes GPU resources and increases wait times due to zero-padding shorter requests to match longer ones.

Request-level batching's resource wastage problem has been solved using iteration-level batching introduced by Orca [15]. Iteration-level batching allows requests to dynamically enter and exit the current batch of requests in processing at the end of each model iteration. Modern LLM serving systems like vLLM [59], LightLLM [60], and S-LoRA [14] use iteration-level batching to significantly increase system throughput and utilization.

Abuse Prevention in Multi-tenant Environments. High volumes of requests in modern LLM applications increase the risk of abusive behavior, potentially impacting fairness for other users. Rate-limiting solutions, such as RPM, can prevent abuse and ensure fairness by imposing limits on user request rates within a period of time [61]. However, in multi-agent scenarios, throttling a request in the middle of an interaction can lead to resource wastage. We discuss the implications of RPM solutions in § 5.2.4.

A recent approach to mitigating unfairness is VTC [6]. VTC ensures fairness by allocating an equal number of tokens per minute (TPM) to all users, regardless of whether they are abusive or benign. This strategy treats every user equally by consistently serving the same TPM and refraining from throttling any user's requests. However, we argue that this approach is inadequate for mitigating abusive behavior and ensuring fairness in complex multi-tenant, multi-agent environments. The diverse nature of applications leads to distinct user behaviors and resource demands, which require a more nuanced approach to ensure equitable service. Furthermore, an abusive user might be a bot flooding the system with requests to degrade the service quality for benign users. Providing equal service to such users without throttling their requests consumes unnecessary resources and increases the response times for genuine users. We discuss implications of VTC approach in § 5.2.1 and § 5.2.2.

Chapter 3

Workload-aware Caching in Machine Learning Training

3.1 Introduction

Deep learning (DL) approaches are increasingly being employed to solve crucial complex problems. The use of DL has become common in disparate domains such as health sciences [62, 63, 64, 65], environmental sciences [66, 67], bio-technical systems [68], high-energy scientific experiments [69], finance [70, 71, 72, 73], smart cities [74, 75], industrial production [76, 77], autonomous vehicles, and IoT systems [78, 79, 80]. Moreover, DL has given rise to a huge market that is expected to reach 12.12 billion dollars by 2025 [81]. To meet the demands of unprecedented scale and performance, DL researchers and practitioners are developing distributed DL, which employs distributed computing and storage resources to support DL. While promising, the approach poses numerous challenges in handling massive workloads while keeping the usage cost in check.

DLT is extremely compute-intensive and data-intensive [82], and the resource demands vary at different phases of the process [83, 84]. A key challenge is efficiently matching the DL application needs with available system resources. A common practice is to scale up/out a DL training job using multiple compute accelerators such as GPUs, FPGAs, or custom ASICs; that is, by using data parallelism [16] with each accelerator, e.g., GPU, holding a replica of the model and processing a subset of the training data in parallel.

A large body of research has focused on optimizing the efficiency of computing [85, 86], scheduling [87, 88], and data communication [89, 90, 91] for DL jobs. This is because data-parallel DL training is both compute-intensive—typically requiring multiple GPUs to train in parallel—and communication-intensive [92, 19, 20]—newly calculated model gradients are transferred or broadcast to all the involved GPUs for iterative model updates. However, as state-of-the-art research [93, 13] demonstrates, the efficiency of data storage

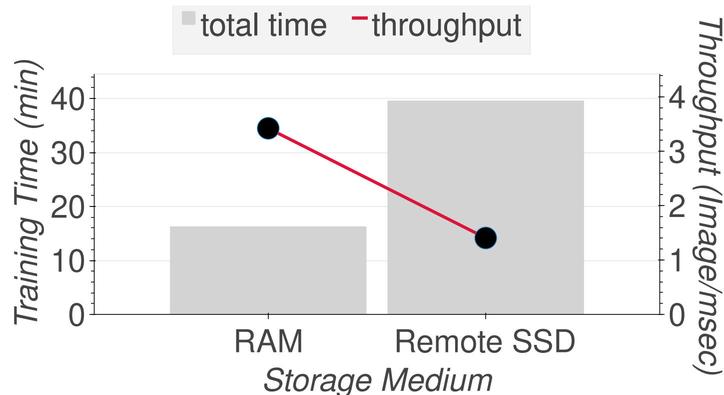


Figure 3.1: Training throughput and time comparison of a single job using 2 nodes and 8 GPUs with ResNet-18. Remote storage comprises of SSDs on a BeeGFS server.

and retrieval can also significantly impact the end-to-end performance of DL training.

To better understand the impact of data storage configuration on distributed DL training efficiency, we perform an experiment to study the performance difference when distributed DL jobs are run using a local or a remote storage medium. Figure 3.1 shows that remote storage mediums can significantly impact the training time ($\sim 2.5\times$) compared to faster storage mediums, i.e., RAM, even though all the rest of the training configurations were kept the same. This result is in line with recent studies [7, 8], which show that I/O can take up a considerable portion of the total training time. Efficient I/O can significantly reduce training time, as high-performing accelerators consume training data samples quickly.

However, it remains challenging to improve the I/O efficiency for distributed DLT as the I/O workloads of a DLT job exhibit unique patterns: (1) full-object, sequential, read-only accesses at per-object level; (2) dominant, small, random I/Os spread across the whole training sample dataset [31]; and (3) highly concurrent I/Os [7]. Today's high-performance and distributed storage systems, such as parallel file systems (PFS) [94, 95], network file systems (NFS) [96], and cloud object stores [97, 98] are inefficient at supporting dsitributed DL applications. This is especially true given the excessive metadata overhead for small-I/O-intensive accesses [99].

For efficient I/O, faster storage mediums like RAM are needed, but compared to increasingly large training datasets that can range from terabytes [100, 101] to petabytes [102], these resources are often too small, even on large supercomputers like Piz Daint (64GB RAM/node) [103] and Fugaku (32GB RAM/node) [104]. Moreover, because of the high cost of GPUs, DL jobs are mainly run by renting GPU Spot VMs [13, 105, 106, 107, 108, 109] that are 6-8$\times$ cheaper than dedicated VMs. As these VMs are preemptive, meaning these can be terminated at any time depending on available resources, DL training has to be resumed from a checkpoint on a different VM leading to the loss of local SSD state. As a result, instead of local SSDs, large datasets are put in persistent cloud storage, and training is conducted on VMs that access the cloud storage remotely.

Worse, conventional wisdom holds that the I/O workload of a DL training job is not *cache-friendly* due to the aforementioned I/O randomness and lack of data locality [99]. This property renders existing caching policies (such as LRU and LFU) ineffective, as there is no recency or frequency pattern to exploit. Recent work such as Mercury [26], CoorDL [12], Quiver [13], and Hoard [29] try to solve this I/O problem by employing caching techniques. Unfortunately, none of them provides fundamental solutions that enable the ability to cache (i.e., *cacheability*) a DLT job's working set. The main reason is that these works consider that each sample will only be accessed once in every epoch (one iteration over the dataset). However, as has been shown by prior work [110], some samples are more important than others in DL training. Hence, if we can design effective mechanisms and policies to exploit this importance variance, we can fundamentally improve the cacheability for DL training.

In this paper, we show that we can deliver better cacheability by designing a new dataset sampling algorithm inspired by importance-sampling [111] and an effective caching policy atop that. DL models are trained on a dataset in batches (multiple equal partitions of the entire dataset). Our sampling algorithm combines the intra-batch importance of individual data samples with inter-batch importance to detect the most important samples for placing in the in-memory pooled cache. We develop a novel technique of *rank-based importance* that ranks the training samples within a batch based on their contributions to increasing the overall accuracy of the model. Rank-based importance further helps increase the probability of identifying (predicting) the most important samples in later epochs. Using this technique, we further design a *priority-based sampling strategy* that ensures multiple accesses to the important samples within an epoch to train more on hard-to-learn samples to increase the accuracy improvement rate. As a result, our caching solution keeps the most important samples in the cache and avoids random evictions, which in turn improves the cache hit ratio and training throughput.

Specifically, this chapter makes the following contributions.

- We introduce a novel, rank-based importance calculation approach to precisely identify the relative importance of data samples for DLT jobs.
- We design a priority-based sampling policy to exploit the data locality of samples.
- We present the design and implementation of SHADE, a new DLT-aware caching system that incorporates rank-based importance scores and the priority-based sampling policy to improve the I/O efficiency for DLT jobs.
- We incorporate and evaluate SHADE in the widely-used DLT framework PyTorch and compare SHADE against a series of baseline and advanced caching and sampling methods. Our results show that SHADE: improves the read hit ratio by up to $4.5\times$ given the same cache size, increases the training throughput by up to $2.7\times$, and reaches accuracy convergence by up to $3.3\times$ faster compared to a baseline LRU caching policy.

SHADE is open source and publicly available at:

https://github.com/R-I-S-Khan/SHADE.

3.2 Background & Motivation

3.2.1 DL Training with Importance Sampling

Traditionally, SGD-based DL training is oblivious to the "importance" of training samples and simply applies random sampling or shuffling to generate a random permutation order at the end of each training epoch, thereby treating all training samples equally. Recently, researchers found that in SGD-based DL training, a specific set of training samples tend to generate little-to-no impact on the model quality and, therefore, can be ignored [110, 111]. This process of finding the set of training samples that are more important than others, i.e., contribute the greatest towards the loss function, is known as importance sampling. That is, a few samples would lead to a higher loss between hidden layer output and target label in backward propagation after a few epochs.

Therefore, by prioritizing training using samples with relatively higher importance, i.e., the ability to contribute towards building model accuracy, a DL training job can achieve improvement in both training time and test errors [110, 112].

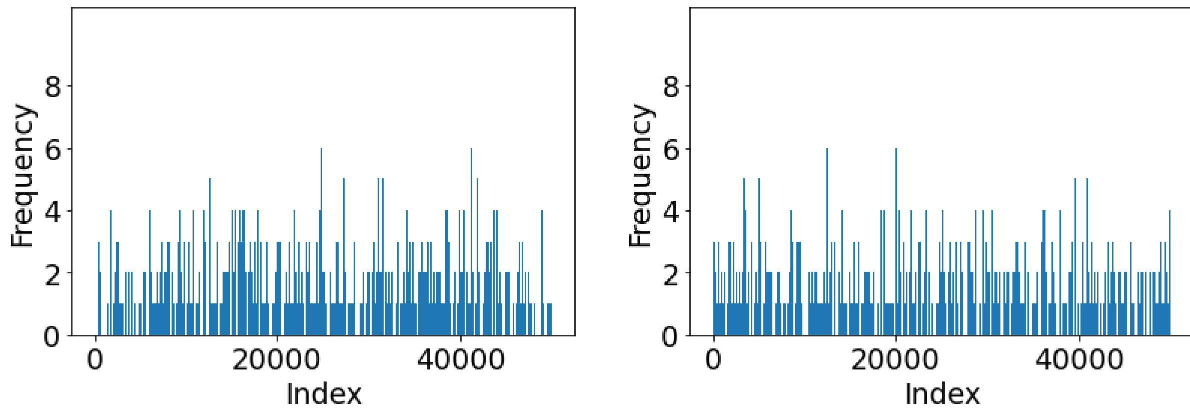
In SGD, gradient $g(x)$ is estimated by sampling from a uniform distribution p where x is a data sample from a minibatch. Importance sampling estimates $g(x)$ using a new data distribution q (such that $q(x) > 0$ whenever $p(x) > 0$) to speed up the process. That is,

$$\mathbb{E}_{p(x)}[g(x)] = \mathbb{E}_{q(x)}[\frac{p(x)}{q(x)}g(x)] \tag{3.1}$$

It has been proved [113] that the variance of gradient is minimized when Eq. 3.2 is maintained, i.e., to ensure gradient variance reduction, optimal data distribution $q^*(x)$ should be proportional to sample's gradient norm $|g(x)|$.

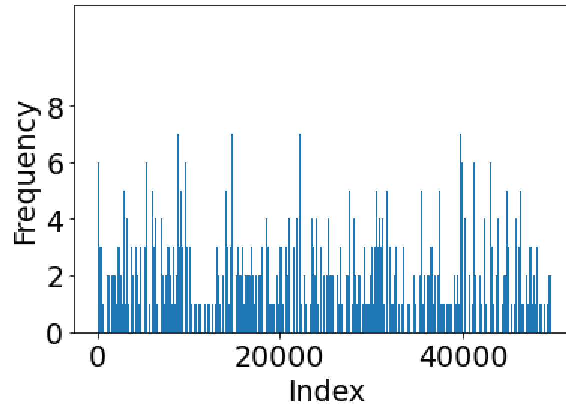
$$q^*(x) \propto p(x)|g(x)| \tag{3.2}$$

In practice, the feed-forward loss is often used to measure the importance of each data sample as an alternative of gradient.



(a) Sample access pattern in epoch 1.

(b) Sample access pattern in epoch 20.



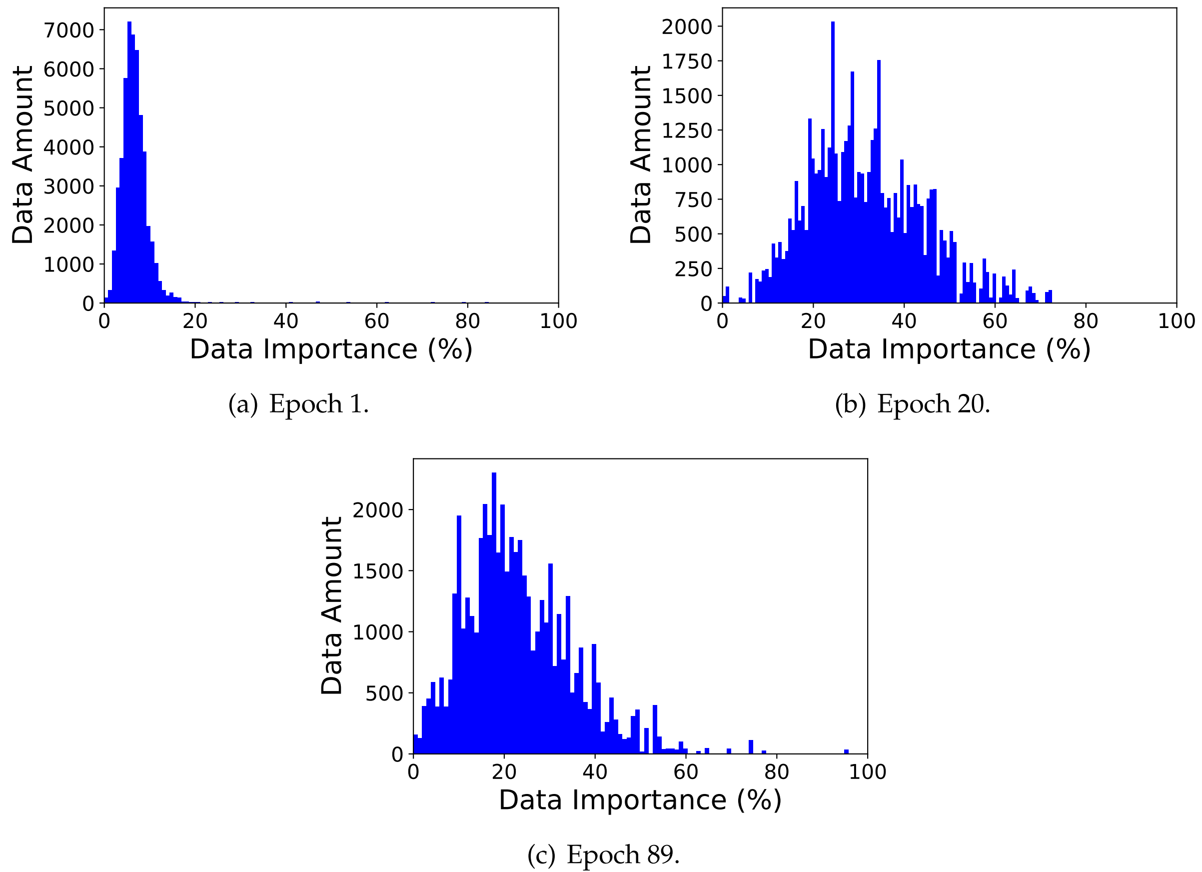
(c) Sample access pattern in epoch 89.

Figure 3.2: Frequency of samples accessed across different epochs in default single process importance sampling (CIFAR-10).

3.2.2 Exploiting Importance Sampling

As discussed earlier, the shuffling-based sampling method passes over the entire training dataset in each epoch, making DL training not cache-friendly and failing to make efficient use of faster storage mediums such as main memory or SSDs. However, as observed in Figure 3.1, there exists ample opportunity to make efficient use of faster storage devices to enhance the performance of DL training. In this regard, importance sampling treats training samples differently and introduces inherent data locality that can be exploited by a caching system to make better use of faster storage mediums.

To better understand the implication of importance sampling on DL training and dataset caching, we analyze importance sampling based training over benchmarking datasets.



(a) Epoch 1.

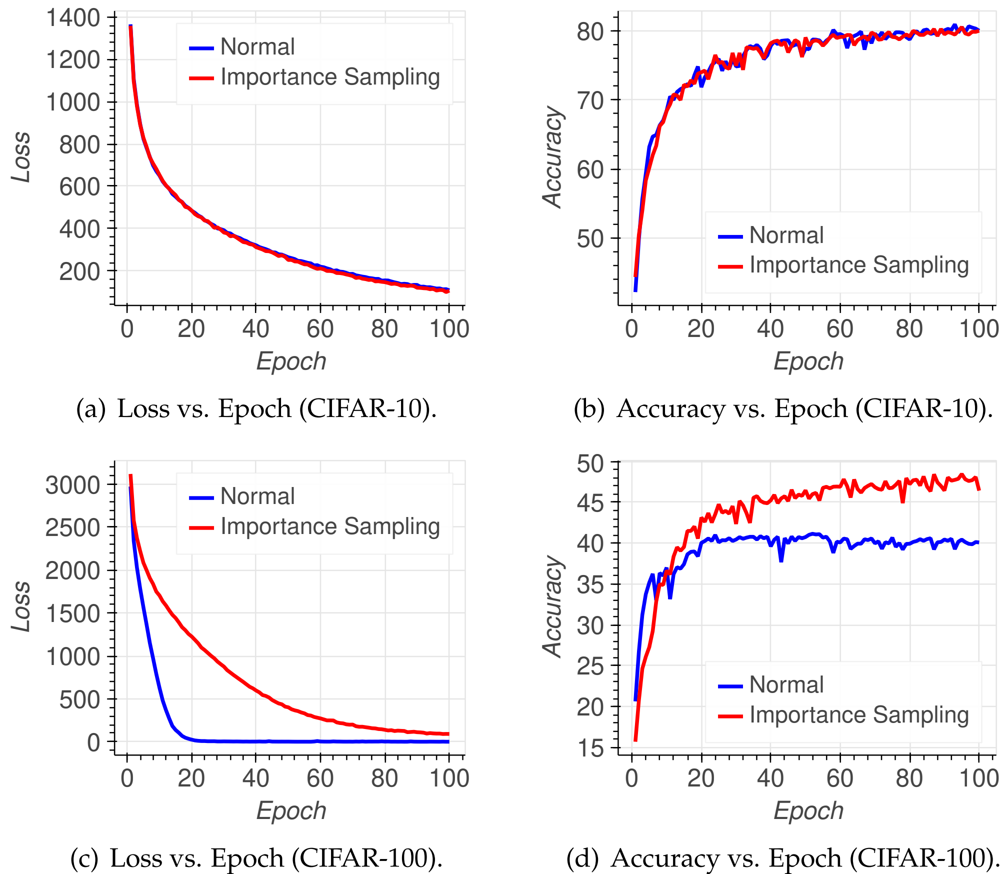
(b) Epoch 20.

(c) Epoch 89.

Figure 3.3: Distribution of data importance as the number of epochs increases in single process default importance sampling on the CIFAR-10 dataset. Data importance is the ability of a sample to contribute towards improving the accuracy of the model.

Sample Access Pattern. We first analyze the sample access pattern of importance sampling. We use the CIFAR-10 dataset [114] to train a ResNet-18 DNN model using a loss-based importance sampling algorithm [110] for a single GPU. As shown in Figure 3.2, 26.5%, 26.6%, and 26.2% of the samples are accessed more than once in epochs 1, 20, and 89, respectively. More importantly, 9.6% of the samples are accessed 3 times or more in epoch 89, indicating a good data locality within a training epoch.

Data Importance Distribution. We further analyze the importance scores of training samples. Unlike standard random sampling, which treats each sample equally, not all samples contribute equally to model training. As shown in Figure 3.3(a)-(c), in epoch 1, the importance scores of all samples are clustered towards the least-important end of the spectrum; whereas during epochs 20 and 89, more samples become more important. In particular, in epoch 20, around 49.91% of samples have a normalized importance score



(a) Loss vs. Epoch (CIFAR-10).

(b) Accuracy vs. Epoch (CIFAR-10).

(c) Loss vs. Epoch (CIFAR-100).

(d) Accuracy vs. Epoch (CIFAR-100).

Figure 3.4: Comparison of loss and accuracy convergence of ResNet-18 model using single process default importance sampling against baseline training on the CIFAR-10 and CIFAR-100 datasets.

greater than 30%. This observation further implies that the importance information could be exploited by a priority caching policy to optimize the I/O efficiency of DL training.

Impact of Importance Sampling on Training. Next, we analyze the impact of importance sampling on training quality. In this test, we use the CIFAR-10 and CIFAR-100 datasets and train a ResNet-18 model using standard random sampling and a loss-based importance sampling method. As shown in Figure 3.4(a)-(b), we verify that importance sampling incurs negligible impact on the model accuracy for the CIFAR-10 dataset. The CIFAR-100 dataset is much harder to predict than CIFAR-10 due to the larger number of classes present in the dataset. Figure 3.4(c) shows that importance sampling does not have a drastic loss degradation implying that it has a good learning rate, which further contributes to its improved accuracy. Figure 3.4(d) shows that importance sampling can achieve better accuracy in under 20 epochs than the accuracy achieved by normal baseline random

sampling in 100 epochs. This is because random sampling just shuffles the dataset indices, which does not contribute much towards quickly learning fine-grained details of the dataset.

3.3 System Design

Our study in section 3.2 sheds light on the potential to enable fundamental data locality for DL training workloads and motivates a new caching system co-designed with the DL training framework. This section presents the challenges and design principles of SHADE, followed by the design detail.

3.3.1 Challenges

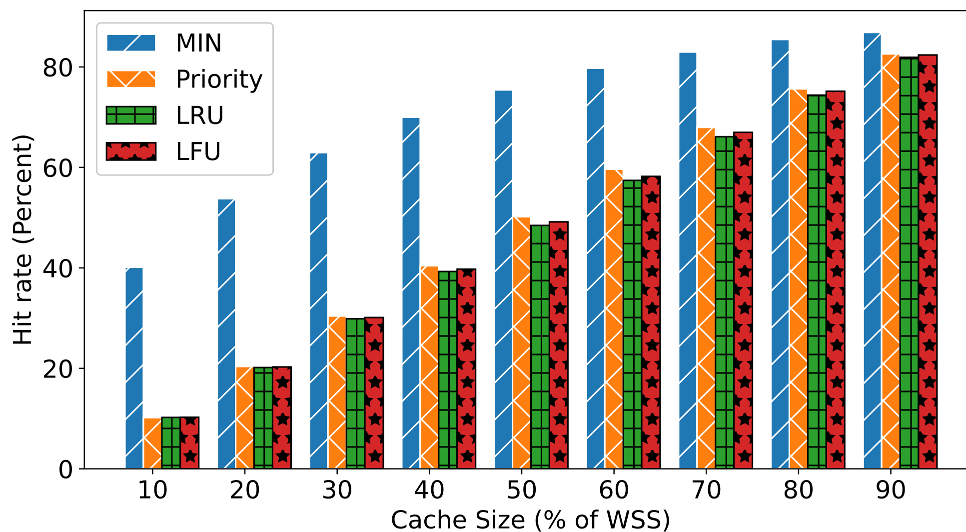
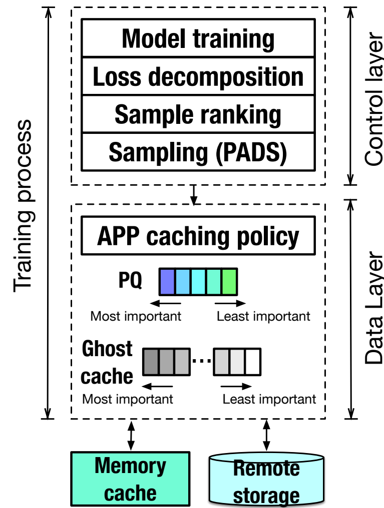
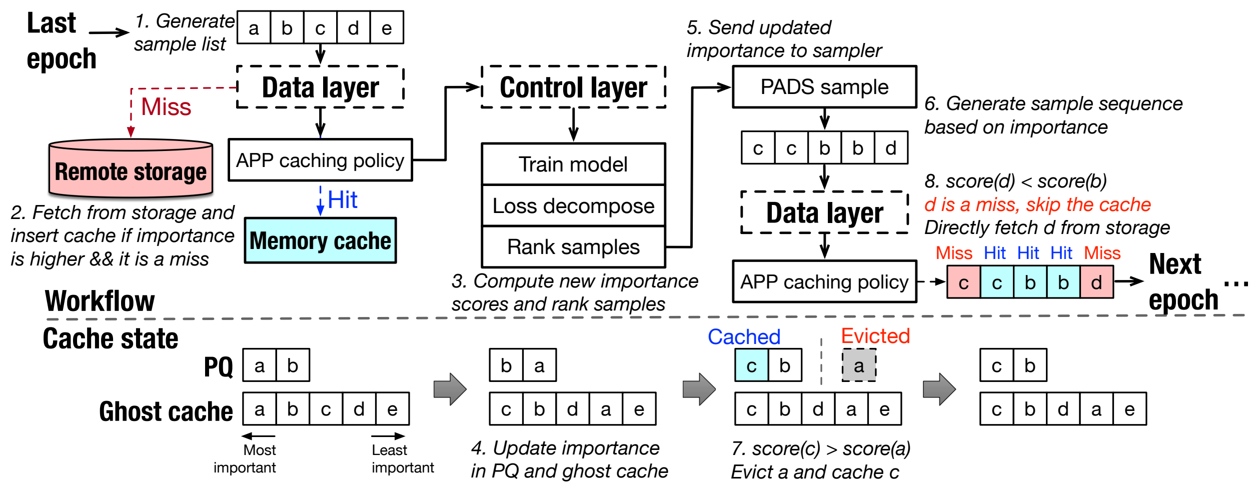


Figure 3.5: Comparison of different caching policies during ResNet-18 model training over the CIFAR-10 dataset. Working Set Size (WSS) denotes the percentage of cached dataset.

Our goal is to achieve a caching system that can exploit importance sampling to improve the cache efficiency for DLT's I/O workload. One may think that a priority-based caching policy that always prioritizes the most important samples could effectively improve the read hit ratio.



(a) SHADE's logical architecture.



(b) SHADE's workflow.

Figure 3.6: SHADE architecture overview. (b) In illustration of how SHADE's components interact in a single epoch.

However, as shown in Figure 3.5, a naive priority-based caching policy achieves the same low read hit ratio as standard LRU and LFU. Ideally, Belady's MIN cache replacement policy [115] achieves an optimal read hit ratio assuming perfect future knowledge: Belady's MIN replaces the item that will be accessed furthest in the future (Figure 3.5). In the context of DLT, an ideal priority caching policy would accurately capture the priorities (i.e., importance scores) of training samples, resembling the optimal behavior of the offline MIN. To make it even better, the policy could take advantage of the importance information to prefetch important samples into the cache. This way, the new policy can potentially

outperform Belady's MIN when incorporating prefetching [116].

The key insight of this paper is that DLT treats different training samples differently and that the priorities of I/O accesses are inherently predictable, therefore exposing interesting exploitation opportunities to fundamentally improve the I/O efficiency. However, it also poses non-trivial challenges to effectively translate the potential exploitation opportunities to the I/O efficiency improvement.

First, default importance sampling (importance sampling considered in prior works) assigns per-minibatch scores, which are too coarse-grained and inaccurate. That is, all samples of a single minibatch are, by default, assigned the same importance scores. This creates ambiguity, which leads to inaccurate estimation of the per-sample importance and, thus, loss of cache efficiency. Ideally, we would want an importance score that precisely tells us the relative importance that each sample carries within a minibatch.

Second, even with proper identification of important samples, excessively feeding the DL model repetitive samples can make the training model biased. Therefore, it is crucial to maintain accuracy while increasing the sample hit rate.

Third, importance scores are constantly changing and may get stale quickly. The same sample in a later minibatch may contribute differently toward the model than it did in an earlier minibatch. Thus, capturing the most up-to-date importance score information is imperative to make informed caching decisions.

In the next section, we discuss four novel techniques to address these challenges.

3.3.2 SHADE Overview

SHADE consists of two main components—the control layer and the data layer. The control layer provides the data layer with the list of samples needed for training. For the first iteration, the data layer fetches samples from a remote storage and populates the cache with the samples that are to be accessed first. During training, the control layer finds the importance (loss decomposition + ranking) associated with the samples and the priority queue (PQ) and ghost cache tracking the importance of samples in the data layer are updated. Based on the newer importance, a *sampler* in the control layer prepares a samples list with associated repetitions information. When the data layer receives the list of samples, it checks whether it is beneficial to cache a newer item instead of evicting a cached prior sample. Let's suppose the sample being accessed has higher importance than the min_sample (sample having lowest importance in the current cache). In this case, the min_sample is evicted, and the current sample is cached using our new Adaptive Priority-aware Prediction (APP) cache policy. This process is repeated throughout the entire DL training. As SHADE keeps the most important samples in the distributed cache and repeatedly uses these hard-to-learn samples for training, it can ensure improved rate of accuracy and a good cache hit ratio. Figure 3.6 shows the architecture of SHADE along

with the components and interactions therein.

Control Layer

The SHADE control layer performs two main functions. (1) It calculates the importance scores associated with data samples, and (2) it samples the data for different training processes. The importance scores are then transferred to the data layer in real time for making prefetching and caching decisions.

The SHADE control layer features three techniques to find accurate, fine-grained importance scores for each data sample. The first technique finds out the importance of samples in per-sample granularity (i.e., fine-grained). The second technique uses fine-grained importance and ranks the samples to make them suitable for priority-based caching. Finally, the third technique uses rank-based importance to build a list of important samples with repetitions to be used for training that will increase the read hit ratio and maintain a good learning curve.

Loss Decomposition. In default importance sampling, the forward training loss is calculated for minibatches, and this forward loss is then assigned as the importance score for all the samples in the minibatch [111]. As a result, the default importance sampling method calculates the ability of a minibatch to contribute towards improving the overall accuracy of the model instead of the data samples themselves. However, as expected, not all the samples of a minibatch contribute equally to the accuracy improvement of the model. Therefore, we need the sample-level loss information, i.e., the loss of individual samples of a minibatch, to calculate the importance score of each data sample.

To address the first challenge concerning the coarse-grained importance scores at the minibatch granularity, SHADE uses both the sample-level and minibatch-level cross entropy loss information to decompose the coarse-grained importance scores into per-sample scores. The cross entropy for each sample denotes the uncertainty with which the model could predict the class label for a sample. Measuring the uncertainty helps SHADE to detect the importance of a sample.

Assume a minibatch has S data samples, and T represents the number of class labels. This constructs an output layer for the DNN model with a matrix that has a dimension of $S \times T$. Each row of the matrix encodes the raw likelihood or logits of a sample for each of the T-class labels. To capture the contribution of each data sample, SHADE decomposes the loss function and calculates the loss corresponding to each sample in the minibatch. SHADE decomposes the loss function using two steps. In the first step, SHADE calculates the categorical-cross entropy for each sample in the minibatch. The categorical-cross entropy for each sample, E_{sample}, is defined in Eq. 3.3:

$$E_{sample} = -\sum_{i=1}^{T} T_i \log S_i, \tag{3.3}$$

where T_i represents the hot-encoded truth label for a given sample under class i, and S_i denotes the softmax probability for a sample in a minibatch for the class i.

S_i is calculated using Eq. 3.4:

$$S_i = \frac{e^{r_i}}{\sum_{j=1}^{T} e^{r_j}}, \tag{3.4}$$

where r_i denotes the raw likelihood of a sample for class i, and the denominator is a normalization term. SHADE uses a softmax normalization over a standard normalization method for two reasons. (i) This method can effectively identify small and large variations in raw logit values and thus assign the importance scores accordingly. (ii) The raw logit values can be negative, so taking exponents ensures that we always end up with a positive value. SHADE uses the per-sample-based entropy loss for finding and feeding the model with the most important samples.

The second step of the loss decomposition involves calculating the minibatch importance necessary for adjusting the model weights. As softmax is continuously differentiable, it is possible to calculate the derivative of the cost function with respect to every weight of a DNN model. SHADE uses all the per-sample-based entropy losses found from the first stage (Eq. 3.3) for calculating a mean entropy loss according to Eq. 3.5:

$$E_{batch} = \frac{\sum_{k=1}^{S} E_{sample_k}}{S} \tag{3.5}$$

A higher entropy for a sample means that the model generates multiple predictions for a single sample out of the several T different possibilities, i.e., the model faces more difficulties in generating a single accurate prediction for that sample. Correspondingly, a lower entropy for a single sample signifies that the model can generate a single prediction for it with high enough accuracy. Thus, a lower entropy value for a sample means that the sample is not highly important in increasing the accuracy of the model in later epochs, and a higher entropy value signifies the opposite. The reason is straightforward: samples that the model has already learned cannot help much in increasing the accuracy of the model in later epochs, and only by learning the harder samples can the DLT job improve the accuracy. Ideally, an entropy value of zero means that the difference between the predicted and ground-truth label is an absolute zero and that the sample is accurately learned. In practice, however, the entropy cannot reach zero as there are no useful models that have 100% accuracy.

Our goal is to prioritize samples that have higher entropy during model training so that the model can learn these hard-to-learn samples better. Consequently, the loss decomposition

method enables SHADE to capture hard-to-learn samples from a minibatch without extra transformation of the raw data.

Rank-based Importance Score. Even though the per-sample entropy score provides a simple tool for quantifying the importance of different samples, it does not tell how much different samples contribute to the accuracy of the model when sampled together in a single minibatch. The relative rankings allow SHADE to prioritize the most important set of samples from each minibatch and, thus, the entire training dataset. To identify the *relative* contribution of a sample in a minibatch, we derive a log-based ranking method shown in Eq. 3.6:

$$rank_i = \log\left(\sum_{k=1,k\neq i}^{B} I(l_i > l_k) + b_0\right) \tag{3.6}$$

The rank-based importance score for the ith sample in a minibatch with B samples is denoted by $rank_i$. l_i and l_k denote the entropy loss for the ith and k^{th} sample, respectively. b_0 is a bias term used for fixing the range of ranks in the log scale. I is an identity function that returns 1 when the condition $l_i > l_k$ is true, and 0 otherwise. For each k item in a batch B, this condition helps to place each sample in the proper rank in a minibatch. A sample having a higher loss gets a higher rank.

Consider the following example. Assume two minibatches, $B1$ and $B2$, contain samples $< 4, 5, 6 >$ and $< 7, 8, 9 >$, respectively. Assume the samples in $B1$ have entropy scores of $< 0.3, 0.5, 0.4 >$ and samples in $B2$ have $< 0.6, 1.2, 0.8 >$, respectively. These entropy values are raw values, which will be problematic when comparing the sample importance across minibatches. For example, a priority-queue-based cache would rank sample 5 from $B1$ in the lower half globally when sorting all samples from both the two minibatches, even though sample 5 is the most important one in $B1$. Sorting these samples by the entropy scores would give us a relative rank with respect to each of the two minibatches $B1$ and $B2$, meaning sample 5 and 8 are the most important in $B1$ and $B2$, respectively.

To get the accurate changes in importance score, i.e., whether the importance score is increasing, staying unchanged, or decreasing, SHADE uses the log scale. In this case, the relative importance scores remain in the same range and can be used for priority differentiation in a priority queue data structure. Relative scores are desirable for three reasons.

First, our method guarantees that the per-sample-based importance scores from different minibatches and epochs are in the same range in order to precisely differentiate their priorities in the cache. Different samples from different minibatches would share the same rank in our method if they contributed the same proportion in improving the accuracy of the model when grouped in their corresponding minibatches. Whereas in the default importance sampling method, all samples from the same minibatch are assigned the same

importance, resulting in at most one minibatch that will have the highest importance score, r_{max}. This is erroneous as all samples in the same minibatch do not contribute equally to improving the accuracy. When the number of minibatches is more than the minibatch size, SHADE is guaranteed to capture more important samples than the default method, and this, in turn, helps in improving the read hit ratio of the cache. Specifically, if the DLT job is configured to train on N samples with a minibatch size of B, where N is significantly larger than B (which is the common case in DLT), then SHADE can effectively identify (N/B) important samples in one epoch. In contrast, the default method can only capture B important samples.

Second, although models are constantly being updated during the training, training against harder-to-learn samples may help mitigate the high volatility of the accuracy rate, therefore leading to smooth model training.

Third, the relative ranks make it easy to predict the data importance online: the top $x\%$ important data in a minibatch is guaranteed to be within the set of the top $x\%$ of the whole dataset according to our defined importance score. Based on this property, SHADE effectively offers an implicit prefetching mechanism, which will be described in Algorithm 1.

Priority-based Adaptive Data Sampling (*PADS*). At the end of an epoch, when SHADE has calculated the (rank-based) importance of samples, the SHADE sampler sends the data layer the list of sample indices that should be used for training.

However, instead of naive random shuffling, the SHADE sampler first prioritizes the samples that contributed the most to the accuracy of the model by constructing a multinomial probability distribution of the data samples, as there are many possible outcomes/selections of the dataset. Based on the generated distribution, the sampler builds a list of important samples for shuffling and sharding across different DLT processes. SHADE seamlessly combines prefetching and caching: based on the updated importance scores and the list of samples provided by the sampler, SHADE data layer prefetches the most important samples to the distributed in-memory cache. The sampler provides the data layer with a list of repetitive samples, which helps the data layer automatically prefetch important samples in real-time. The design strikes a balance between the cache efficiency (read hit ratio) and model accuracy. SHADE keeps track of the loss convergence of the model in real-time to decide the number of repetitive sample accesses in order to boost the hit rate without sacrificing the accuracy of the model. To do so, on noticing a steep decline in the loss convergence curve or a stagnant accuracy curve, SHADE intentionally shuffles the important samples to avoid training against a small subset of the most important samples. This way, the system is able to mitigate aggressive importance sampling, which minimizes training biases.

SHADE's *PADS* policy plays a crucial role in increasing the hit rate of a limited-sized cache and, in certain cases, can even outperform offline MIN. Consider the following example. Assume we have ten samples $< 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 >$ during training. The

samples have no repetitions, as random sampling does not consider the importance of the samples. Assume each training process trains on five samples, and the cache holds two samples. Assume the sampler provides samples $< 1, 3, 5, 6, 8 >$ randomly for training. Then, offline Belady's MIN will put any of the two samples that will be used by the training process. Assume it puts $< 1, 3 >$ in the cache. In this case, the hit rate will be only 40% $< hits - 1, 3 >$.

In the case for SHADE, *PADS* would create a samples list with repetitions based on importance. Assume the samples are $< 4, 7, 3, 5, 3, 5, 2, 2, 1, 10 >$ and samples 3, 5, and 2 are the most important. Then *PADS* would provide the training process with the samples $< 3, 3, 5, 5, 2 >$ from the list of samples so that $< 3, 5 >$ can be cached. The hit rate is now 80% $< hits - 3, 3, 5, 5 >$. Even if we consider repetitions in the case of Belady's MIN, it is bounded by the access pattern of the sample list provided by the random sampler. For example, if the sampler provides $< 4, 5, 5, 1, 10 >$ to the training process, on knowing the future, Belady's MIN can, at most, cache 5 (as it is needed twice) along with another sample. Assume it caches $< 5, 10 >$, so the hit rate will be only 60% $< hits - 5, 5, 10 >$.

SHADE's sampling method is fully decentralized and does not require a centralized server to coordinate. Decentralized sampling means that each training process derives the importance scores of the samples independently based on its own local model training.

Data Layer

The SHADE data layer provides mechanisms and policies for cache eviction and prefetching. A challenge regarding sample caching is that the importance scores are constantly changing, even within an epoch, as one data sample can be accessed in multiple minibatches. In order to address this challenge, we design a new cache policy called *Adaptive Priority-Aware Prediction (APP)*, a dynamic policy that updates the importance score of a data sample as soon as the importance score changes.

SHADE's data layer consists of two components: *an indexer* and *a pooled in-memory cache* that spans multiple key-value storage (KVS) servers. The index uses two heap-based priority queues (PQ and ghost cache) for tracking the samples along with their associated rank and access frequency for each training process.

The data layer introduces index numbering for each individual data sample. Index numbering enables the control layer to assign importance score at the sample granularity. Once the control layer calculates and assigns importance scores for the data samples, the indexer inserts the data sample index numbers, but not the actual sample data, into the PQ (priority queue for the current state of the cache). During data loading, the cache will use the importance scores provided by the indexer to make informed prefetching and eviction decisions. The data layer also performs serialization and deserialization when inserting and fetching image samples to and from the cache. The APP caching policy is shown in

Algorithm 1: Adaptive Priority-aware Prediction (APP) Cache.

```
1  Input and Initialization:
2  PQ: Priority queue for currently-cached samples, ghost_cache: Priority queue for all
     previously-trained samples
3  for epoch in total_epochs do
4      for s in sample_dataset do
5          v = score(s) # Calculate importance score based on Eq 3.3 and Eq 3.6
6          ghost_cache.set(s,v) # Insert/update in ghost_cache
7          if cache_hit then
8              cache.get(s)
9          else if cache_miss and cache_not_full then
10             cache.insert(s) and PQ.set(s,v)
11         else
12             if cache_miss and ghost_cache.exist(s) then
13                 x = ghost_cache.get(s).score # Get the most recent score of data sample
14                 min_sample,min_score = PQ.min() # Find the sample with the minimal
                     score in the cache
15                 # Check if the sample to be processed is more important than the least
                     important one stored in the cache
16                 if x ≥ min_score then
17                     cache.evict(PQ.pop(min_sample)) # Evict the least important sample
                         from cache
18                     cache.insert(s) and PQ.set(s,v) # Insert this sample into the cache and
                         PQ
19                 else
20                     read_from_storage(s) # The data is less important than any samples
                         currently cached, skip caching
21             else
22                 read_from_storage(s) # Evicting a known sample for an unknown one
                     may not be beneficial, skip caching
```

Algorithm 1.

The PQ and the ghost cache are sorted by the importance scores. PQ keeps track of the metadata state of currently cached samples in the cache, while the ghost cache tracks all the metadata state of the samples that have ever been cached (including those that have been evicted). The ghost cache entries do not store the actual data samples, but rather store a mapping between the data sample ID (the sample index) and the metadata tuple record of $< ir, af >$, where ir is the importance score and af is the access frequency.

During training, a cached sample might lose its importance if it is well-learned; that is, it might lose its priority in the cache and gets evicted. At the same time, another sample that has been evicted previously might turn out to be important and therefore gets inserted into the cache. The ghost cache helps decide whether a previously-evicted sample could be brought back into the cache.

When the cache is full, and the data sample to be processed is a miss, SHADE checks ghost cache for the previous importance score of the data sample (Algorithm 1 line 12): if this data sample had been previously evicted out of the cache, it should be included in the ghost cache. If the most recent importance score of this data sample is greater than or equal to that of the cached sample that has the smallest importance score, the currently-cached least-important sample (min_sample) is evicted from the data cache as well as from PQ. After the eviction, the data sample that is to be processed (and was previously evicted) is inserted in the cache (line 18).

In summary, by comparing the current importance scores of data samples already in the cache and that of the most recent importance score of the current data sample being processed, SHADE's data layer predicts and maximizes the likelihood of a sample being reused in the cache in the future.

3.4 Implementation

SHADE is implemented in PyTorch 1.7. PyTorch has three main components: `Dataset`, `Sampler`, and `DataLoader`. Dataset class provides the image dataset access points and exposes a *__get_item__* method that fetches a sample along with its target label for a given index. `Sampler` provides subsets of samples of the dataset to the training processes in random permutations. `DataLoader` uses the information provided by the sampler to load the samples in minibatches with the help of worker processes. In SHADE, we implement a new class by inheriting the PyTorch `Dataset` class. The `ShadeDataset` class has functionalities to combine the samples and their corresponding class labels so that `Dataloader` can fetch data samples easily from the remote storage.

We extend the `DistributedSampler` class to prepare the `ShadeSampler` class that has the main logic of the SHADE's *PADS* policy. It provides APIs to communicate with training processes and receive the per-sample entropy values for each minibatch. At the end of each epoch, `ShadeSampler` forwards the important samples to the training processes.

We introduce the logic for the data layer by overriding the *__get_item__* and *__len__* method in the `ShadeDataset` class. The *__len__* method returns the total length of the `ShadeDataset`, and the *__get_item__* method exposes the indices associated with the data samples that enables the client layer to find importance in per-sample granularity. In addition, the *__get_item__* method is connected to the in-memory pooled cache to make decisions on

caching and eviction based on the heap-based PQs of the data layer using the *APP* cache policy. For the in-memory pooled cache, we use Redis [24].

We have implemented an analysis framework atop the setup to facilitate experimentation and statistics collection. The framework takes as input the configurations of the experiment, which include paths of the training dataset, master's address:port, number of training nodes, number of GPU devices, number of epochs to run the test, the batch size, and the DNN model to be trained. The framework then sets up the environment accordingly. It collects GPU-related statistics using nvidia-smi [117] and I/O-related statistics using sar [118]. SHADE's implementation is system agnostic – DL practitioners do not need to write new code to use SHADE in their systems. SHADE does not use any extra system-level resources compared to a normal DL training with local/global caching.

3.5 Evaluation

3.5.1 Experimental Setup

Our study covers distributed training with multiple GPUs and a remote storage deployed on Chameleon Cloud [119]. Several recent works [120, 121, 122] have used Chameleon Cloud for conducting high-performance experiments, making it a representative testing platform. Our method is compared against baseline distributed training in PyTorch—one of the most popular frameworks for deep learning [123]. Although caching policies are not publicly available in PyTorch, we have built an LRU caching policy on top of PyTorch to ensure a thorough evaluation of our proposed storage caching policy. In addition, we have evaluated SHADE against importance sampling with six different caching policies to perform a robust ablation study. For our analysis, an HDD-based NFS Server [96] is used as remote storage. For training, we have used eight NVIDIA P100 GPUs (PCIe with 16 GB memory) spread across four nodes. All the GPU nodes and storage nodes are connected via a 10 Gbps interconnect.

Our experiments primarily use the ImageNet-1K dataset [124], which contains $\sim$1.2 million images with a total size of 138 GB spanning 1,000 object classes. We also use CIFAR-10 [114]. We conduct our study on four representative computer vision (CV) distributed DL models, namely, Alexnet [125], ResNet-18, ResNet-50 [126], and VggNet [127]. The setup we use to evaluate our system is representative of CV distributed DL training, which has been used to evaluate prior research works [26, 29, 13, 8]. In the following, the reported percentages for cache size indicate the portion of the dataset that had been cached.

3.5.2 Cache Hit Ratio

In this set of experiments, we evaluate the performance of our APP caching policy against several other policies. This will help explain the extent to which SHADE is able to make better utilization of the limited cache space. The `baseline` uses PyTorch's default random sampling and LRU caching policy for eviction. We also implement and evaluate the offline Belady's MIN policy. In addition, we evaluate six SHADE policy variants based on importance sampling:

1. Priority-based LFU policy (`sh_pqlfu`), which evicts the samples with the least importance score based on a hybrid priority that combines the batch-forward loss (i.e., coarse-grained score) and sample access frequency. If the forward loss is the same, then eviction decision is made based on the access frequency of samples.
2. Priority-based policy (`sh_pq`), which uses the batch-based, goarse-grained forward loss as the importance score.
3. LRU (`sh_lru`), which uses the coarse-grained forward loss but evicts samples based on the recency of the items and not the importance scores.
4. LFU (`sh_lfu`), which uses the coarse-grained forward loss but evicts the least-frequently-used sample.
5. Random (`sh_rand`), which performs random evictions.
6. APP (`sh_app`), which makes eviction decisions using our APP policy but does not use loss decomposition, rank-based importance score, and *PADS* sampling.

Figure 3.7 shows the the average read hit ratios when training the three DL models (Alexnet, ResNet-50, VggNet) over the CIFAR-10 dataset under the studied policies with different cache sizes.

We observe that the margin by which SHADE performs better than policy 1–6 increases as the cache size becomes smaller. When only 10% of the WSS is cached, SHADE, with all techniques incorporated, shows a $4.5\times$ higher read hit rate than the baseline LRU. SHADE without the importance derivation techniques (`sh_app`) can achieve $2.67\times$ and SHADE without the APP cache policy (all the SHADE `sh_` baselines except `sh_app`) can achieve around $1.94\times$ higher hit rates than the baseline. The reason for the improved hit rate is that our techniques are able to predict which samples the training processes would need in the future for better accuracy, and hence it approaches the hit rates of the optimal MIN and even outperforms MIN in some cases (WSS 50% and 75%). This is because MIN's knowledge about samples' future access pattern relies on the sampler. However, SHADE manipulates the sampler to create the desired future sample access pattern, which will benefit the DLT job the most in terms of both training duration and accuracy. This sample

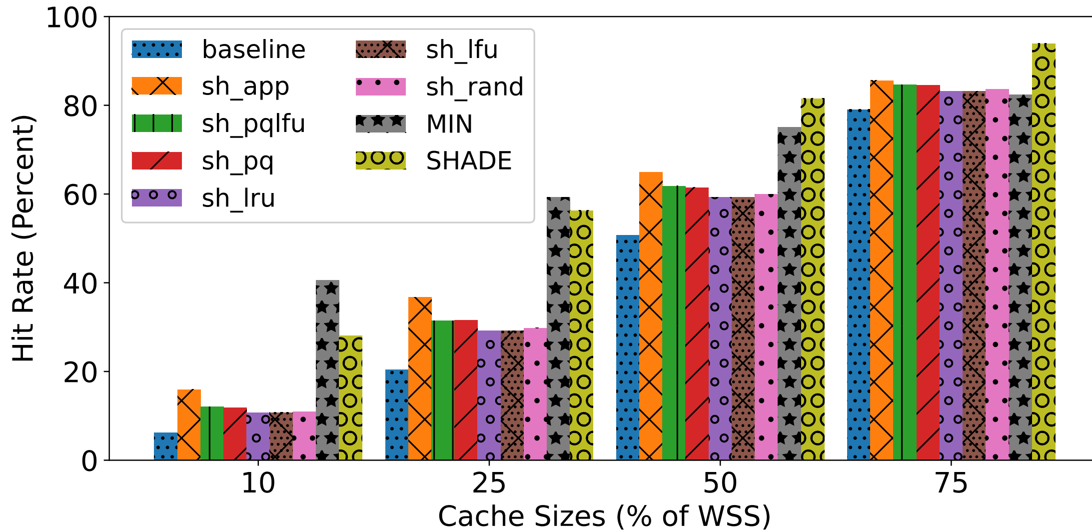


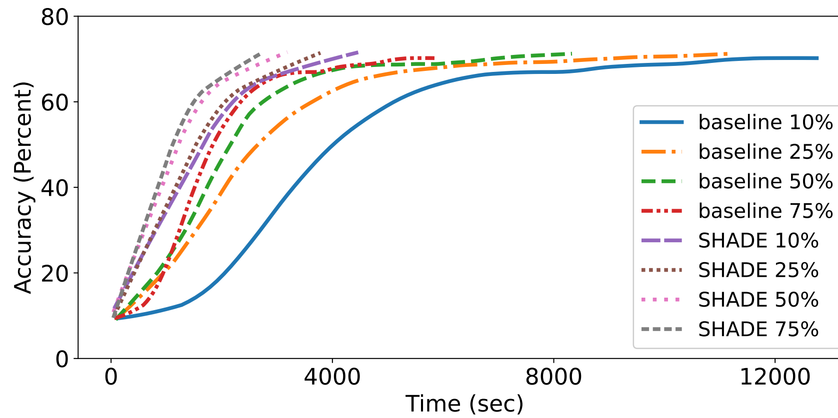
Figure 3.7: Comparison of the read hit ratio of various caching policies and cache sizes. The `sh_` prefix denotes a baseline version of `SHADE` that uses the coarse-grained importance. `SHADE` denotes our contribution, SHADE, with all techniques enabled. `WSS` denotes working set size.

access pattern comprising multiple hard-to-learn samples enables precise I/O prediction and maximizes the likelihood of a sample being reused in the cache in the future. By ensuring a higher hit rate with limited available cache space, SHADE holds effectively more data in the limited cache space, therefore achieving higher DLT efficiency.

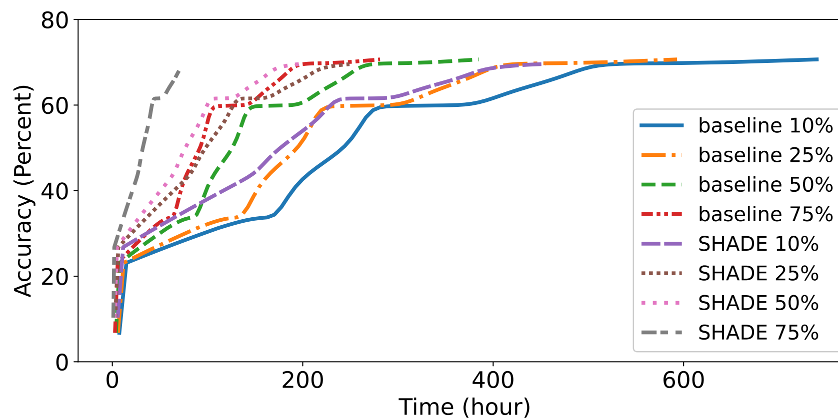
3.5.3 Accuracy vs. Time

In our next tests, we evaluate how model accuracy changes over time for SHADE when compared to the baseline at different WSS. This shows how quickly SHADE is able to train a model and increase the accuracy using a very small cache even when the baseline has the advantage of using more cache space than SHADE.

Figure 3.8 shows that SHADE has a better accuracy improvement rate compared to baseline policy. For example, SHADE can achieve up to $3\times$ faster accuracy convergence compared to baseline storing 10% of the dataset in the cache when being trained on the Alexnet model. Fine-grained relative importance of samples helps SHADE detect the most important, i.e., hard-to-learn samples, train more on them and thus improve the accuracy quickly to reach convergence. Again when being trained on the ResNet-50 model, SHADE continuously maintains a better accuracy improvement rate compared to baseline at similar WSS. SHADE can reach accuracy convergence $3.3\times$ faster compared to baseline at 75% WSS. The accuracy improvement rate of the baseline with a larger cache is not always better in Figure 3.8(a)



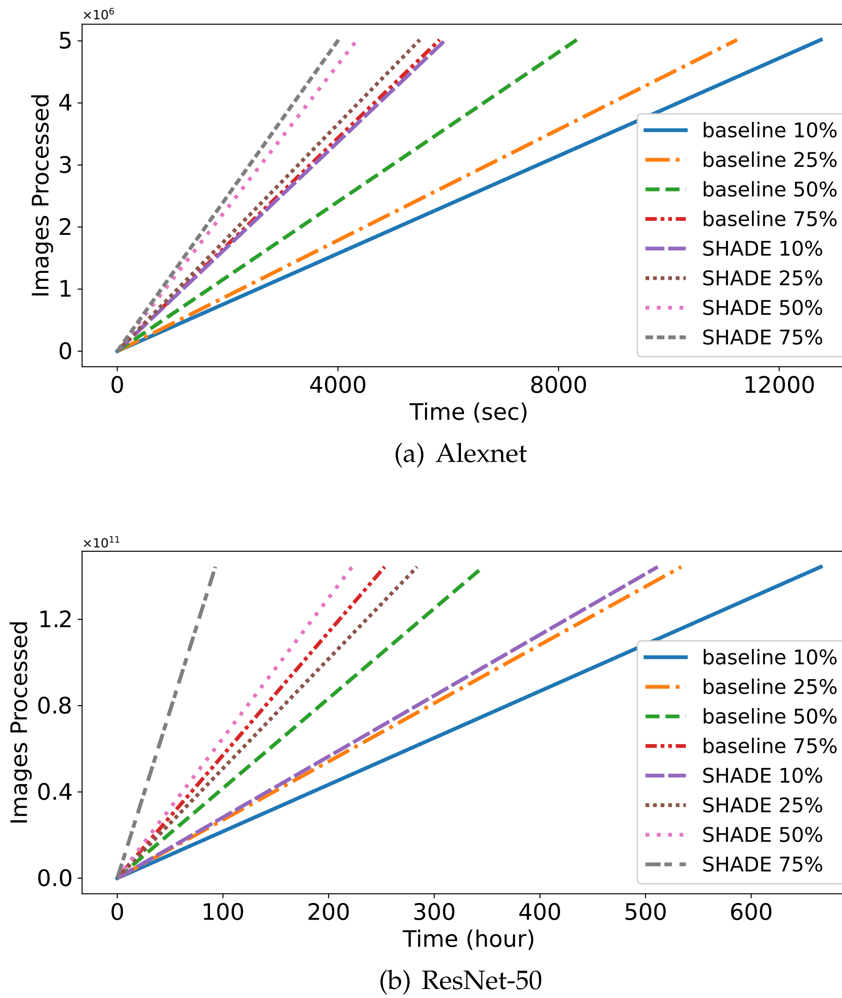
(a) Alexnet



(b) ResNet-50

Figure 3.8: Accuracy improvement rate of SHADE against baseline LRU when different portions of the entire dataset is cached (denoted by the percentages).

because the baseline uses random sampling. Random sampling places equal emphasis on all of the samples and hence cannot improve the accuracy quickly by training more on the hard-to-learn samples. The improvement in accuracy vs. time curve for the baseline comes only from caching more data. Hence, our results show that even if a larger portion of the dataset is cached, naively caching data items without proper techniques to exploit data locality can not guarantee improved performance.



(a) Alexnet

(b) ResNet-50

Figure 3.9: Throughput of SHADE against baseline LRU when different portions of the entire dataset is cached (denoted by the percentages).

3.5.4 Throughput

In our next experiment, we evaluate the throughput of SHADE, which help demonstrate the superiority of SHADE in processing data while storing a limited portion of the dataset. Figure 3.9(a) shows that SHADE while caching just 10% of the dataset has around 2.3× better throughput compared to baseline policy caching 10% of the dataset. The baseline matches the throughput performance of SHADE only when the baseline is caching 7.5× more data compared to SHADE.

In the experiment with ResNet-50, shown in Figure 3.9(b), we observe that SHADE at 75% WSS has 2.7× higher throughput compared to baseline at similar WSS. Even at lower

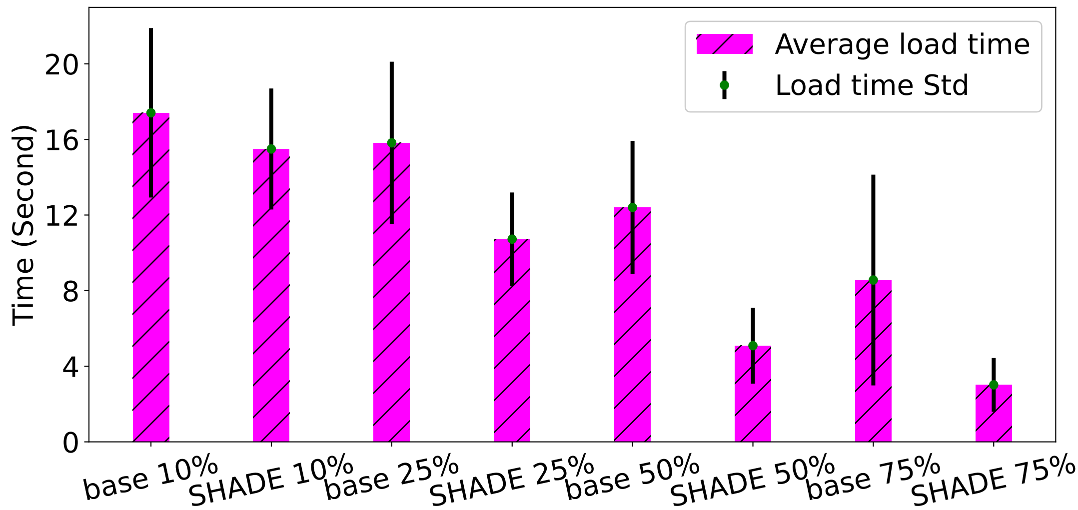


Figure 3.10: GPUs' minibatch load time when training ResNet-50. Percentages denote the amount of cached dataset.

WSS, SHADE can achieve higher throughput compared to baseline at higher WSS. For example, SHADE at 50% WSS has a 1.14× higher throughput than baseline at 75% WSS. The improvement in the ability to process more images is due to the ability of SHADE to exploit data locality with APP cache policy. Although baseline at 75% WSS has a slightly higher throughput compared to SHADE at 25% WSS, it is unable to get a better accuracy improvement rate seen in Figure 3.8(b). This is because SHADE can exploit data locality and has techniques to train on important samples which ensures a better accuracy improvement rate.

3.5.5 Minibatch Load Time

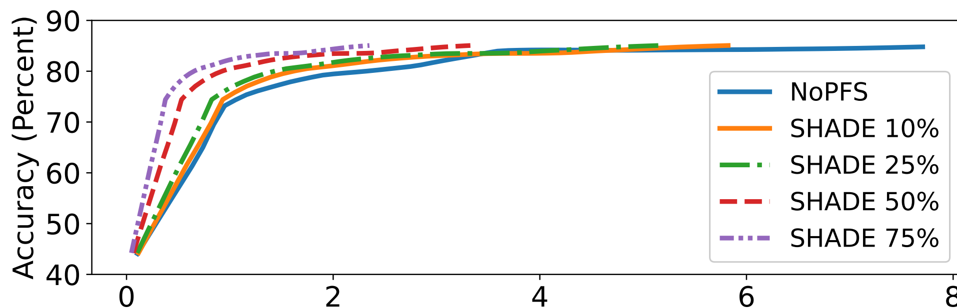
In our next test, we evaluate the performance gain observed in minibatch load time. Consistency in minibatch load time is important so that all the training processes can remain coordinated. It also shows the effectiveness that a caching policy has in exploiting data locality. Figure 3.10 shows the average minibatch load time of the GPUs during the course of training with the vertical lines showing the standard deviation of the load time within a single epoch.

As we can see in the figure, SHADE can achieve a lower mean load time compared to baseline at similar and higher WSS. The baseline at 50% and 75% WSS has 2.5× and 1.7× higher minibatch load time compared to SHADE at 50% WSS. Moreover, SHADE can maintain a small standard deviation in minibatch load time. Ideally, we would expect the standard deviation in minibatch load times to be low if larger portions of the dataset

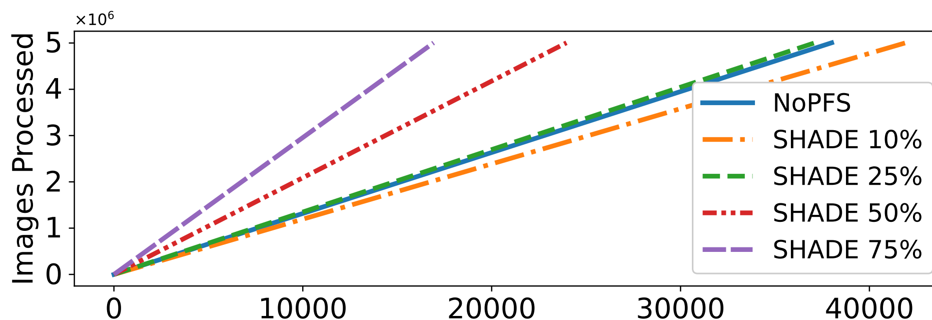
get cached because of using a higher portion of the fast RAM storage. However, it is not the case for baseline, even if it caches larger portions of the dataset in the cache. Average minibatch load time is highly variant for baseline caching 50% and 75% of the dataset. The baseline at 75% WSS has a 3.9× higher standard deviation in minibatch load time compared to SHADE at 75%.

3.5.6 End-to-End System Comparison

In our last set of experiments, we compare the performance of SHADE against NoPFS [5], a state-of-the-art storage system for improving the I/Os of DLT workloads. NoPFS exploits the seeds that generate the random access pattern when performing SGD-based DLT to predict when and where a training sample will be accessed. Similar to our baseline, NoPFS uses random sampling of indices. The difference lies in that NoPFS does not consider importance and uses *Clairvoyance* (i.e., seeds that generate random access patterns) to approximate "future distances" of Belady's MIN [115]. SHADE considers fine-grained importance of samples and uses *PADS* policy to prioritize samples for training.



(a) Accuracy vs. time.



(b) Throughput.

Figure 3.11: Comparison of SHADE and NoPFS [5]. Percentage denotes the percentage of cached dataset.

For fair comparison, we keep the experimental setup and training parameters the same for both SHADE and NoPFS while training on the CIFAR-10 dataset. Figure 3.11 shows that NoPFS incurs a $4.5\times$ and $2.4\times$ increase in training time to reach accuracy convergence compared to SHADE at 75% and 50% WSS, respectively. At the same time, SHADE has $2.2\times$ and $1.6\times$ better throughput compared to NoPFS when working at 75% and 50% WSS, respectively. SHADE can still attain accuracy convergence faster even at 10% WSS. SHADE performs better than NoPFS as it adopts a prefetching system that aims to approximate Belady's MIN; it is bounded by the sample access pattern provided by the sampling policy and hence prioritizes all samples equally. As a result, it takes more time to reach accuracy convergence compared to SHADE, which trains more on hard-to-learn samples to increase the accuracy improvement rate faster.

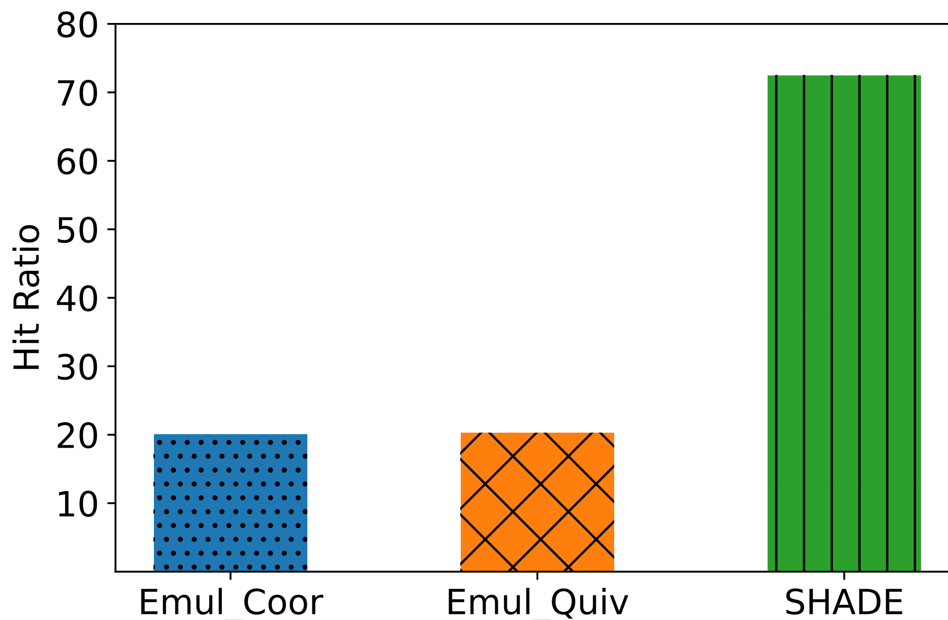


Figure 3.12: Comparison of the read hit ratio of different caching policies at 20% WSS of CIFAR-10.

We further compare the cache hit ratios of SHADE with state-of-the-art DLT caching policies. we configure a small cache space of 20% of the WSS over the CIFAR-10 dataset using the ResNet-18 model against emulated caching policies including CoorDL [12] and Quiver [13]. To understand the impact of these techniques in importance-aware training, we use a loss-based importance sampling technique [110] inspired by Mercury [26]. For emulating CoorDL and Quiver, we create our own implementations of the core techniques of CoorDL and Quiver, which we name as `Emul_Coor` and `Emul_Quiv`, respectively. Both `Emul_Coor` and `Emul_Quiv` use a KVS as a cache similar to SHADE. `Emul_Coor` ensures that no items are ever evicted from the cache once these are inserted in the cache. In the case of `Emul_Quiv`,

we implement the substitutability technique, which replaces a missed sample with a sample already in the cache to avoid memory thrashing.

Figure 3.12 shows that both `Emul_Coor` and `Emul_Quiv` can only extend their utilization up to the size of the cache ($\sim 20\%$) because these caching policies are not importance-aware and therefore cannot exploit the data locality of the important samples perfectly in importance-aware training. Both of these policies populate the cache using random samples and hence are unable to get a good hit rate by exploiting the repetitions among samples that occur throughout training. On the other hand, as SHADE can manipulate the sampling process (*PADS* policy) and keep repeated samples in the cache, it can achieve a higher hit ratio (72.5%) and thus outperforms both `Emul_Quiv` and `Emul_Coor` by $3.6\times$.

3.6 Related Work

Several recent works have explored the use of importance sampling for optimizing the system efficiency of DL workloads [26, 27]. iCACHE [27] is an importance-sampling-informed DLT cache. Although this approach uses a form of fine-grained importance similar to SHADE, it does not have a rank-based relative score scheme and SHADE's *PADS* sampling approach, due to which it may suffer from a lower cache hit ratio than SHADE. Moreover, in case of a cache miss, iCACHE uses substitutability, which may impact the training accuracy convergence.

Mercury [26] enhances DL training efficiency by prioritizing important samples. However, unlike SHADE, it is not an I/O cache and does not manage data replacement or eviction. CoorDL [12] examines PyTorch's data retrieval process and identifies I/O as a bottleneck in DL training. To address this, it introduces a MinIO cache that randomly selects data items in the first epoch and retains them throughout training without eviction. However, as shown in Figure 3.12, simply caching random samples alone fails to deliver the expected performance gains.

A body of work is focused on optimizing the I/O components of DL applications. NoPFS [8] adopts a prefetching approach that uses hardware level configurations to take caching decisions based on a sample access pattern obtained from trying to approximate Belady's MIN. However, in common online training like hyperparameter tuning experiments [28] with different random seeds, such sample access patterns change constantly and hence are not readily available. We address this constant change in sample access pattern through our dynamic cache management policy without depending on hardware configurations for boosting our performance.

Hoard [29], Quiver [13], and FanStore [30] explore the idea of adding a global caching layer to the GPU cluster for improving the training performance of DL workloads. DeepIO [31] proposes an entropy-aware mechanism for determining next minibatches but it does

not offer any cache eviction policies and suffers from lack of dataset randomization. DIESEL [32] is a comprehensive storage solution that supports key-value-based metadata service, task-level caching, and chunk-based shuffling. However, these works do not focus on how to enable fundamental data locality for DLT jobs. SHADE, on the other hand, exploits importance sampling to enable data locality for DLT jobs.

3.7 Summary

The I/O pipeline is a major bottleneck in distributed DLT when data is read from a remote storage. To address this bottleneck, ad hoc solutions such as using faster local storage devices (e.g., SSDs) had been employed. However, those ad hoc solutions cannot fundamentally address the I/O efficiency of DLT workloads. Although caching is possible for DLT, naively caching redundant samples does not provide any benefits. SHADE realizes a DLT-aware caching policy, which takes advantage of the fine-grained importance scores of data samples in order to enable a high level of data locality, and therefore, fundamental cacheability for DLT jobs. Our evaluation demonstrates that SHADE improves the read hit ratio of a small memory cache (of only 10% of the WSS of the dataset) by up to $4.5\times$ compared to traditional, non-DLT-aware caching policies, thus significantly improving the DLT performance.

Chapter 4

Workload-aware Client Scheduling in Machine Learning Training

4.1 Introduction

Modern machine learning (ML) is no longer constrained to running on well-endowed data center clusters alone, rather is being increasingly done closer to the sources of data such as edge devices [36]. As mobile devices become more prominent, the potential to train powerful models with end-user data has garnered a lot of interest [41]. This new paradigm of ML in a distributed collaborative nature, known as federated learning (FL), had a reported global market value of $\$110.8$ million in 2021, and is expected to grow at 10.7% annually [128]. While promising, FL poses numerous challenges, which make it different from traditional ML. First, many unreliable devices that can drop out any moment can end up participating in training. Second, the end-user devices are extremely heterogeneous in terms of training data, computing power, communication capabilities, etc., making selection and efficient cooperation between them challenging.

To address the challenges associated with data heterogeneity and system heterogeneity, researchers and practitioners have put forward a number of techniques [129, 130, 48, 131, 49, 132, 133, 26, 134] for client scheduling and client data sampling to improve the accuracy of the global model and reduce the training time. However, one crucial aspect of system heterogeneity–the memory capacity of client devices, has remained mostly unexplored. While a recent paper [135] examined the impact of limited on-device storage in FL and proposed policies for on-device data selection, it does not fully consider the case of efficiently handling heterogeneous memory cache and flash storage available across devices when I/O is a bottleneck.

To understand the impact that memory and flash device can have on FL training time, we perform two sets of experiments using ResNet-18 [126] model on the FEMNIST [126]

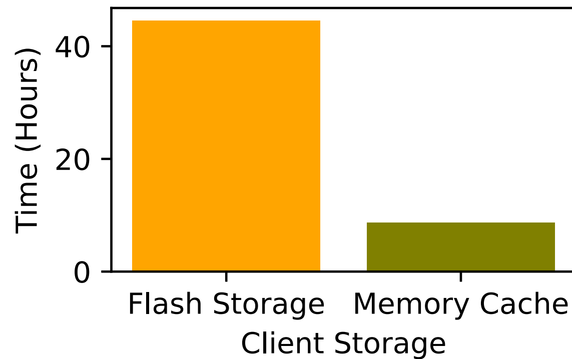


Figure 4.1: Training time comparison between clients using memory cache and flash storage.

dataset. In one, we allow multiple clients to train by fetching all samples from the memory cache. In the other, we allow the clients to fetch samples only from the flash device. Figure 4.1 shows that clients fetching from the flash storage took 5.1× longer to complete the training consisting of 150 rounds. This observation underscores that efficiently exploiting the limited memory cache on client devices can improve performance when training on I/O-intensive workloads.

One way to exploit the limited device memory is through caching client data samples. However, adapting the caching policies across a large number of distributed, heterogeneous client devices is non-trivial as it requires rethinking existing FL client scheduling and data sampling methods, especially with I/O performance as the focus. In particular, there is a need to orchestrate three connected components—(1) client scheduling, (2) client data sampling, and (3) client data sample caching—in such a way that can improve the training time and accuracy convergence of FL workloads.

In this paper, we design FedCaSe—a novel solution aimed at improving FL accuracy and training time in scenarios involving a large number of diverse client devices with heterogeneous memory and data. At the core, FedCaSe builds atop three strategies. First, FedCaSe uses a new sample selection strategy for samples that have demonstrated the most significant potential to improve model accuracy, i.e., experienced samples, and caching them within client memory for efficient, repeated training. Second, FedCaSe identifies experienced clients based on their record of improving model accuracy and reducing wall clock training time, which is gathered through training on the selected influential samples in the previous step. Third, FedCaSe schedules experienced clients with experienced samples in their cache for multiple rounds to ensure robust model training, which reduces the overall training time.

Specifically, this chapter makes the following contributions.

- We characterize the potential of clients (i.e., client experience) having heterogeneous memory to improve accuracy and reduce wall clock time for the FL model.
- We introduce an experience-based score (i.e., sample experience) for data sampling of clients having heterogeneous memory to identify the sample's potential in increasing accuracy and improving read hit ratio (RHR) in limited memory cache of clients.
- We design a novel reverse-optimization technique (RO) to adaptively drive the scheduling of experienced clients in future rounds to improve FL performance.
- We present the design and implementation of FedCaSe– a new FL framework that leverages the experience of millions of clients and their samples to improve overall performance atop heterogeneous memory devices.
- We incorporate FedCaSe in a widely-used FL framework [136] and compare against a series of baselines and advanced scheduling, sampling, and caching methods. Results on a testbed of up to 2800 clients show that compared to the state of the art, FedCaSe improves the experienced client participation up to $29.1\times$, improves the global read hit ratio (RHR) by up to $81.7\times$ (locally up to 318.58×) given the heterogeneous limited memory cache of clients, and thus ensures accuracy improvement rate up to 2.06× faster based on wall clock time, up to 1.4× faster based on number of rounds while keeping the round duration up to 2.4× less.
  FedCaSe is open-source and publicly available at

  https://github.com/rkhan055/FedCaSe/.

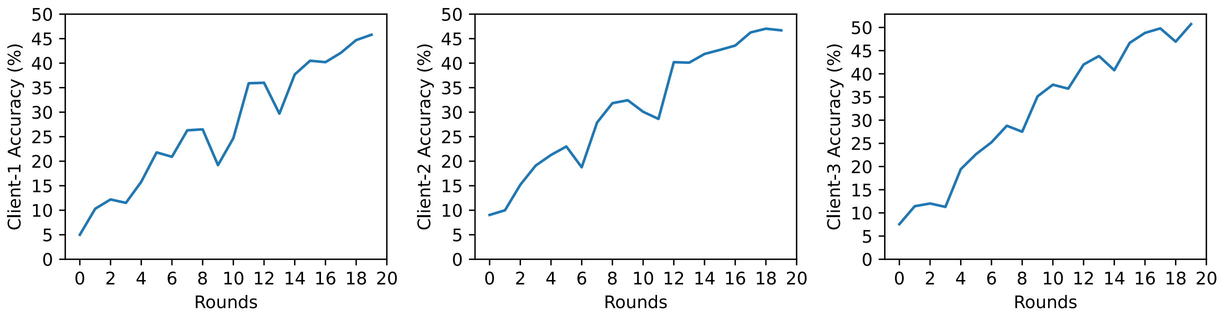


Figure 4.2: Improvement of test accuracy for three different clients using different samples for training from the EMNIST dataset.

4.2 Background & Motivation

This section explores the key concepts of Distributed Deep Learning and Federated Learning, providing essential context for the entire section. Furthermore, we perform an exploratory analysis on the potential of harnessing both client and sample experiences to improve the training performance by effectively mitigating I/O bottlenecks.

4.2.1 Distributed Machine Learning

With the growing adoption of ML, its execution has expanded beyond single-node or single-process settings. While research experiments may still use a single node, real-world deployments often leverage multiple accelerators. This approach, known as distributed ML, enables workloads to meet computational demands efficiently. The two primary forms of distributed ML are data-parallel and model-parallel training.

The entire dataset is partitioned across all participating nodes in equal chunks in data-parallel settings. In contrast, the model is partitioned across multiple accelerators in model-parallel settings, and different accelerators compute different parts of the entire model. These kinds of distributed training are often conducted in large in-house clusters of homogeneous computational nodes. Moreover, the dataset is partitioned into equal chunks across the participating nodes. Models are trained on these datasets in batches through iterative forward and backward passes over the model to minimize a loss function—a mathematical function for quantifying the difference between the predicted values generated by a model and the actual observed target values in a dataset. Minimizing the loss function means finding the model parameters that best fit the available data. The notable concern to us is that there often is no heterogeneity in data quantity and quality along with system resources in distributed ML.

4.2.2 Federated Learning

Federated Learning (FL) [33] follows the distributed ML paradigm, extending computation across multiple clients. Each client trains a global model on its local dataset over multiple rounds, then sends its trained model to a central server. The server aggregates these local model parameters to produce a more robust global model, which is later shared with all clients. This collaboration benefits clients by allowing them to retain their data while accessing a powerful model trained on otherwise unavailable data. FL deployments have a constraint on selecting clients to participate in each round from the entire global set of clients in order to reduce communication and computational overhead: updates are collected from N (configurable) participants that completed training the earliest each round [41]. In FL, client local models are trained while the master model only aggregates the model weights from all clients. Aggregation (tensor calculation) and selection are high-speed compared to the local client training, causing no bottlenecks at the aggregator and scheduler level. The main difference between FL and traditional distributed ML is in data heterogeneity, system heterogeneity, and network heterogeneity among the participating clients, making FL significantly more challenging than conventional ML.

To tackle the heterogeneity, efforts have been made towards guided client selection [48], clustering [49], or improving upon existing ML algorithms [50, 51]. Although these works focus on improving client communication, computation, or client selection, none

fully consider the I/O impact of client samples in a scaled-up federated setting. Some works [1, 13] look at client data sampling, but these are limited to in-house homogeneous cluster settings. While limited storage in client devices has been brought to attention [135], proper methods of exploiting limited memory cache due to I/O bottleneck has not been fully investigated.

In this work, we examine the I/O bottleneck from thousands of diverse client devices with limited memory cache during large-scale FL. We prototype our solution for image-based workloads, i.e., computer vision, as these workloads are known to be I/O-intensive [7, 137, 8, 12, 1] in scaled cross-device FL [35, 36]. However, our approach is general and works with other FL modes and I/O-intensive workloads.

4.2.3 Exploiting Client Experience

In typical cross-device FL, clients are randomly scheduled for a particular round. The random nature of client selection is suboptimal, creating scenarios where clients are scheduled only once throughout training. Selecting a client once means that a particular client has been trained for only one round. To better understand the distribution of the number of times clients get trained during random selection and the impact of rounds, we analyze random-scheduling-based client training with benchmarking datasets.

Impact of Rounds on Training. First, we look at the impact of rounds on improving the accuracy of an ML model. We track an increase in accuracy improvement while training a ResNet-18 [126] model using three clients on the EMNIST [138] dataset. We divide the entire EMNIST dataset into 1000 random chunks and assign three random chunks to three clients, respectively. Each client's data is heterogeneous and each client has a different set of samples from the EMNIST dataset. Figure 4.2 looks at accuracy improvement rate (not the global accuracy) across rounds for each client (x-axis) when clients train on their individual datasets. We observe that after round one, all clients achieve a local accuracy less than 10%. The accuracy increases as the number of rounds increases. This phenomenon indicates that selecting a client only once for training in FL is not ideal and that training a client for multiple rounds is more impactful.

Impact of Random Client Scheduling. Second, we analyze the number of times clients get selected when client scheduling occurs randomly. We train a ResNet-18 model on the FEMNIST [139] dataset, a federated version of the EMNIST dataset. In each round, 100 out of 2800 total clients are randomly selected each round for training over 150 rounds. Figure 4.3 shows that 11.9% of the total clients are scheduled for training only once and around 20.4% of the clients have been scheduled less than three times, indicating that a significant portion of clients are receiving insufficient training opportunities, which may hinder their contribution to the overall model accuracy.

As shown in Figure 4.2, clients need to be trained more than once or twice to reach a

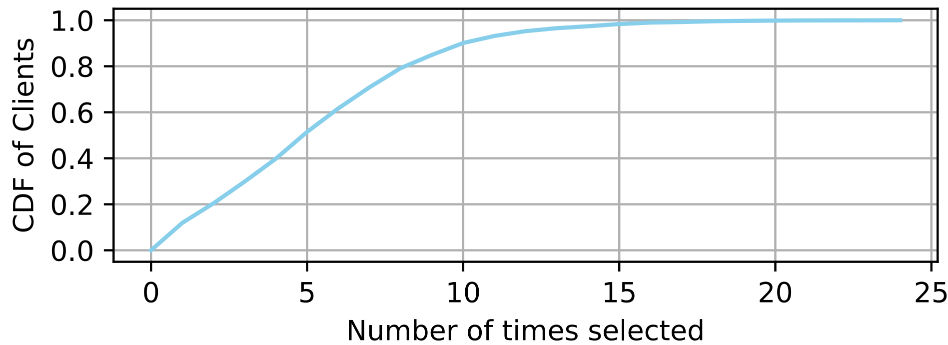


Figure 4.3: CDF of the number of clients selected during random scheduling on the FEMNIST dataset.

reasonable accuracy. However, if more time is taken for a round to complete due to I/O overhead from the flash storage, it can negatively impact the training performance. Hence, characterizing clients' experience based on their impact on decreasing round time and increasing accuracy and using that as a metric for increasing client participation in future client scheduling decisions can likely improve the performance and quality of FL training.

4.2.4 Exploiting Sample Experience

As previously noted, significant heterogeneity exists in the available data samples among clients. In this section, we demonstrate how leveraging this sample heterogeneity across multiple clients can effectively reduce I/O bottlenecks.

Impact of Client Data Sampling. It is known from prior research [110, 111, 140, 112] that some samples in a dataset are more important than others as they contribute more to improving the accuracy of an ML model. We verify this claim through our accuracy improvement experiment in section 4.2.3 on the three clients shown in Figure 4.2. Table 4.1 shows the starting and end accuracy of the three clients on a test dataset. We observe that different clients having different samples have different starting and end accuracies, indicating that certain samples can indeed influence the accuracy of the model in different ways. Moreover, the accuracies of different clients vary across the training. For example, although client 2 has the highest starting accuracy, client 3 ends training with the highest accuracy. In this case, the experience of the clients as well as the samples belonging to it towards improving model accuracy varies throughout the training. We plan to exploit this variance to our advantage.

In FL, client models are aggregated to improve the global model. Hence, we can prioritize client participation along with their samples at certain intervals for improving the global model. For example, Figure 4.2 shows that during round 10, both clients 2 and 3 has a

Table 4.1: Contribution towards accuracy improvement is different in each round for different client samples.

Client ID	Round-1 Accuracy (%)	Round-20 Accuracy (%)
1	5.05	45.81
2	9.04	46.69
3	7.56	50.71

local accuracy over 30% but client 1 has around 20%. In this case, we can improve the global model's accuracy by prioritizing clients 2 and 3 during round 10 and training more on the samples belonging to them. Hence, to improve global model accuracy by merging client contributions, it's essential to carefully consider the impact of each client when selecting them and their data samples for future training rounds. By monitoring how client performance and sample characteristics vary, prioritizing clients with higher-quality or more experienced samples can accelerate model improvements. Training more intensively on these experienced clients with valuable data will allow the model to converge faster and achieve better accuracy in fewer rounds, enhancing overall system efficiency.

I/O Bottleneck in Training. As I/O bottleneck is a known problem [7, 5] in training models, recent works [13, 1] have tried to introduce caching solutions to tackle it. While promising, these solutions are specially catered for homogeneous node settings found in traditional, centralized, distributed training and do not fundamentally solve the severe heterogeneity posed by numerous client devices in FL. In FL, client devices are equipped with fast-processing accelerators [141, 142], which are more data-hungry compared with devices equipped with only CPUs, thus I/O might become a bottleneck in some FL clients. We can measure the I/O performance through looking at the global RHR and local RHR of the clients. Global RHR is the ratio of the total number of hits of all clients over the sum of their total hits and misses. Local RHR is the ratio of the total hits of a particular client over the sum of its total hits and misses.

This I/O bottleneck is exacerbated by the fact that client devices can have heterogeneity in memory cache capacity and the number and size of the data samples. Figure 4.4 shows that a traditional policy, LRU (least recently used), has a poor global RHR of less than 1% in an FL setup of 1000 clients having 10% working set size (WSS). Although employing new optimizations like no evictions (MinIO [12]) and sample substitution (Quiver [13]) can increase the RHR, it is still less than 10%. A lack of a client selection technique that looks at previous patterns means that a client that warmed up its cache in the current round might not be used in the future, resulting in low global RHR. Our target policy should achieve at least a 10% hit ratio when clients cache 10% WSS, with potential to improve beyond 10% (not bounded).

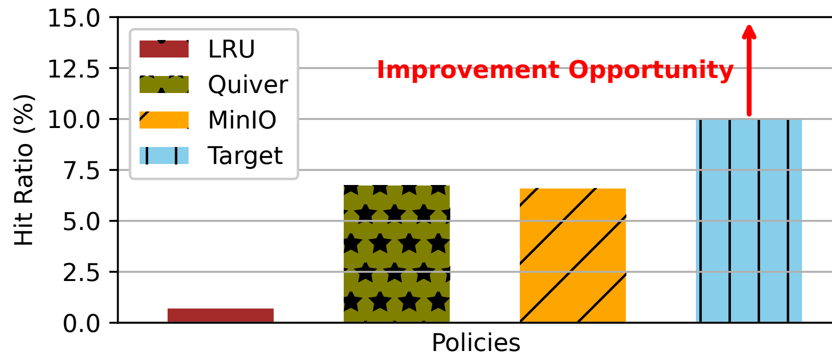


Figure 4.4: Global read hit ratio (RHR) of different caching policies in a heterogenous FL setup of 1000 clients.

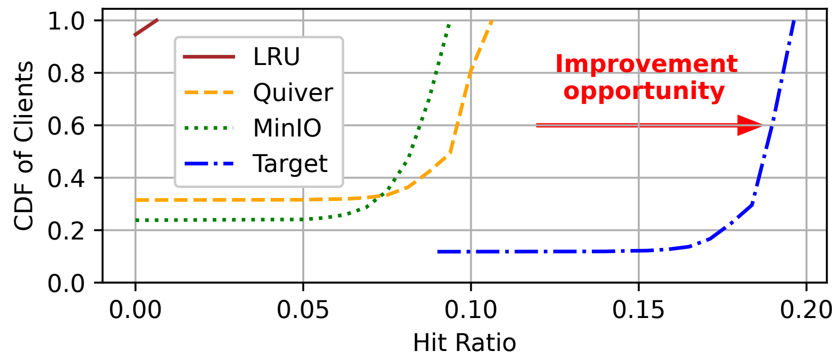


Figure 4.5: Heterogeneous memory creates problems in efficient utilization of memory cache of FL clients.

We further investigate the clients' local RHR. Figure 4.5 shows no client can reach an RHR of 0.1 (i.e., 10%) for LRU and MinIO. Only 19% of the clients have an RHR over 10% for Quiver. As a portion of the samples of each client are randomly selected for each round to reduce time [136], LRU evicts samples from the cache every time a new sample arrives. Moreover, a random selection of samples entails that samples in the cache might not even be used later. Hence, even after employing policies like Quiver or MinIO, only a small portion of the clients get to use the samples inside the cache. If clients cache 10% WSS, then an ideal policy (our target) should be able to exploit previous patterns in training to utilize all of the samples from their limited cache so that the local RHR becomes at least equal (but can be more) to the WSS. At the same time the ideal policy should prioritize selecting experienced and warmed-up clients so that the global RHR also increases above 10%. In this case, intelligently driving the client and sample selection can provide more opportunities for improvement.

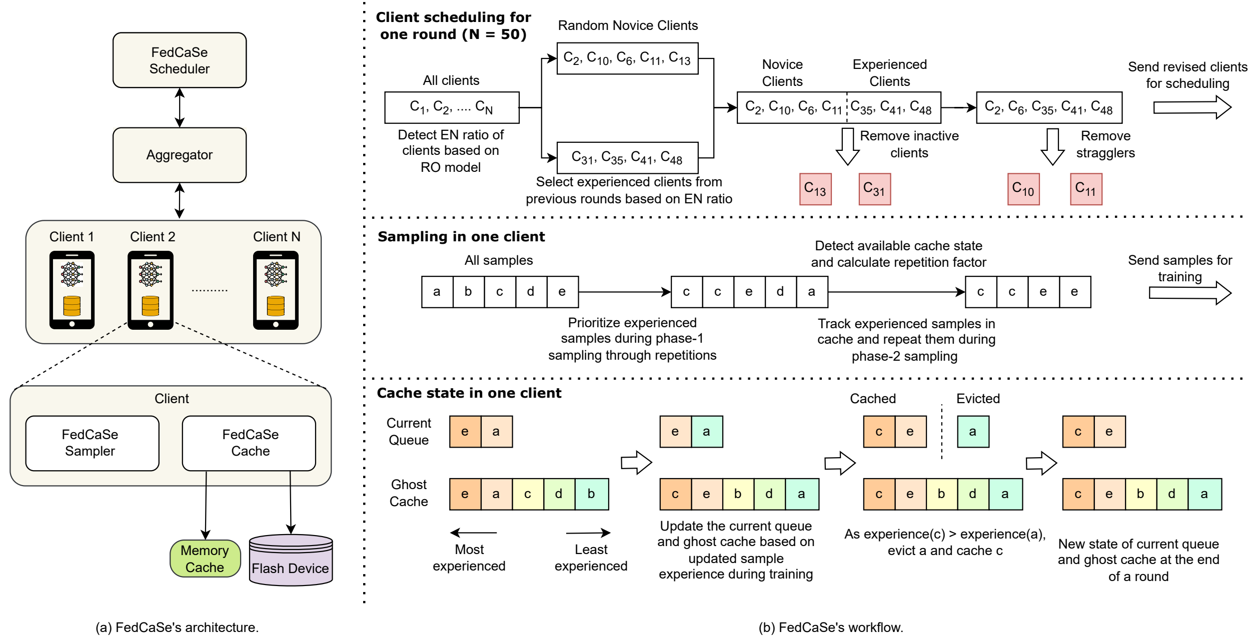


Figure 4.6: (a) FedCaSe architecture overview. (b) An illustration of how FedCaSe's components interact in a single FL round.

4.3 FedCaSe **Design**

Our study in section 4.2 sheds light on the potential to address the I/O bottleneck posed by heterogeneous client devices in an FL setting and motivates a new cooperative mechanism between client scheduling, sampling, and caching co-designed with the FL framework. This section presents the challenges and design principles of FedCaSe, followed by the design detail.

4.3.1 Challenges

We aim to utilize the limited memory available in client devices to enhance the training performance at client granularity and use the obtained performance benefit at the client-level training to drive future clients' scheduling decisions to improve the FL performance. This is challenging as there exists heterogeneity in terms of available memory space, the number and size of client data samples, and the computational and communication abilities of the clients. Although existing policies propose certain techniques, they are limited to addressing only the computational and communication heterogeneity across clients. They overlook one key aspect, i.e., storage heterogeneity across clients. Our main task is realizing a framework that takes clients' caching and scheduling decisions in FL while considering the heterogeneity in all areas surrounding clients to provide enhanced

performance.

The key insight of FedCaSe is steering I/Os, and more specifically, manipulating a cache-oriented sampling selection policy through a lens of FL scheduling, has a (positive) impact on FL training. The reason behind this is that different clients, along with their associated data samples, vary in their ability to perform in a heterogeneous setting, therefore revealing interesting exploitation opportunities to enhance the training performance by intelligently prioritizing clients and samples having a higher experience in improving FL performance.

Leveraging this insight, FedCaSe models a large FL deployment as a geo-distributed, "virtual" object cache; the objective of this newly identified cache optimization problem is to minimize training time and maximize training quality through caching the most experienced samples. To achieve the objective, we present a codesign of FL scheduling algorithm and per-client sample caching policies. However, translating potential exploit opportunities into FL training improvements poses non-trivial challenges.

First, different clients have heterogeneity regarding available memory storage, computation, and communication abilities. Hence, it becomes challenging to factor in these variables and prioritize clients for training. An ideal priority scheme would characterize clients' experience to navigate client heterogeneity and enhance the FL performance.

Second, in each round of FL, numerous clients need to be scheduled. Scheduling only experienced clients might be detrimental to the overall training as there might be other clients which have not been trained. Hence, properly fixing the $experienced : novice$ (EN) ratio becomes non-trivial.

Third, clients have heterogeneity in the size and quantity of samples residing within them. Again, some samples might carry more utility toward improving the model accuracy. Since memory cache is limited within the clients, designing caching policies for each client individually for the samples residing within them becomes challenging.

4.3.2 FedCaSe **Overview**

FedCaSe consists of three main components—the client scheduler, the client data sampler, and the client data cache. The scheduler works in tandem with the aggregator. It calculates the experience of clients based on their ability to increase the model accuracy (model utility) and their ability to decrease the wall clock time (I/O utility). Then, it uses the client experience as a metric to decide the appropriate experienced:novice (EN) ratio of scheduling for the next round using an optimization model. Once the clients are scheduled for training, the FedCaSe client data sampler works at the client level to calculate sample experience for all of the client's samples based on their ability to improve the model's accuracy (model utility) and prospect towards further training (train utility). Based on the sample experience, it decides the best combination of samples for training. During training, the FedCaSe cache keeps the client memory cache updated with the most experienced

samples for faster retrieval to minimize training time and maximize training quality. Figure 4.6 shows the architecture overview of FedCaSe.

Client Scheduler

The client scheduler performs two main functions: (1) calculating the client experience from the client model and I/O utility; and (2) fixing the experienced:novice (EN) client ratio for scheduling clients.

To calculate the client experience from various heterogeneous variables, FedCaSe uses three techniques. The first technique captures a client's relative accuracy improvement performance (model utility) compared to others. The second technique captures the relative improvement in decreasing wall clock time (I/O utility) by a client compared to others. The third technique calculates a composite experience score using the model and I/O utility that takes into account the impact of both utilities. FedCaSe client scheduler then uses the computed experience score as a metric to determine how many and which experienced clients will take part in training in the following rounds. To determine the ratio of experienced:novice (EN) clients, FedCaSe uses a novel reverse optimization (RO) technique grounded in ML.

All these techniques combined help in conducting training with the clients, which will help increase the accuracy of the global model and decrease the round training time. Algorithm 2 shows the steps for client scheduling. At the global server level, scheduling involves tracking client experience scores and running RO policy, with no additional computations beyond standard FL procedures.

Client Model Utility (CM_u). The model utility of a client denotes the utility it carries towards improving the FL model accuracy. To understand model utility, first, we observe the loss (l_c) of each round for a particular client. Loss quantifies how far off the predicted values of a ML model are from the actual values in the training dataset. Using loss for determining importance has been explored in literature [111, 110, 140, 112]. Instead of using raw losses, FedCaSe client scheduler collects the loss of every client at the end of each round as soon as the client finishes training and uses min-max scaling to assign a relative model utility for each client in real time (Alg. 2, lines (7-13)). This standardizes the losses to a uniform scale and makes it amenable to being used with other utilities.

Algorithm 2: FedCaSe Client Scheduling

```
1  Input: C: set of all clients
2  F: {} # sorted dict(client:experience), curr_clients: {}
3  Δ_l: {}, Δ_d: {} # set of loss and round diff. of clients
4  P: {} # set of EN ratios of rounds
5  l_min, l_max: ∞, −∞,    d_min, d_max: ∞, −∞
6  Function collect_client_exp (clients):
7      D: {},    L: {}# duration and loss sets of clients
8      for c in clients do
9          d_c, l_c = train(client) # duration and loss
10         D = D ∪ {d_c},    L = L ∪ {l_c}
11         l_min, l_max = min(l_min, l_c), max(l_max, l_c)
12         d_min, d_max = min(d_min, d_c), max(d_max, d_c)
13         CM_u = (l_c - l_min) / (l_max - l_min)
14         CI_u = (d_max - d_c) / (d_max - d_min)
15         client_experience = α*CI_u + β*CM_u
16         F[c] = client_experience # record experience
17     cur_t, cur_l = find_cur_dur_loss(D,L)
18     return cur_t, cur_l
19 Function get_round_diff (prev_l,prev_t):
20     cur_t, cur_l = collect_client_exp(curr_clients)
21     δ_l = cur_l - prev_l, Δ_l = Δ_l ∪ {δ_l}
22     δ_d = cur_t - prev_t, Δ_d = Δ_d ∪ {δ_d}
23     return cur_l, cur_t
24 Function find_EN (Δ_l, Δ_d):
25     Tδ_l = max(Δ_l) + std(Δ_l), Tδ_d = min(Δ_d) -std(Δ_d)
26     ρ_r = regression(Δ_l, Δ_d, P, Tδ_l, Tδ_d)
27     return ρ_r
28 Function schedule_clients (T_clients,len_rc):
29     prev_l, prev_t = 0, 0
30     for r in rounds do
31         l_r, t_r = get_round_diff (prev_l, prev_t)
32         ρ_r =find_EN(Δ_l, Δ_d)
33         P = P ∪ {ρ_r} # append ρ_r for next round
34         E_clients = ρ_r * F[:len_rc] # experienced clients
35         len_nc = len_rc - len(E_clients)
36         # select novice & new clients randomly
37         N_clients = random_permutation(C)[:len_nc]
38         selected_clients = E_clients + N_clients
39         prev_l, prev_t = l_r, t_r
40         yield selected_clients
```

Client I/O Utility (CI_u). The I/O utility of a client denotes the utility that it carries towards decreasing the round duration. To measure the I/O utility, FedCaSe client scheduler observes the time required to complete a round (d_c) by a client. Each client takes different times to complete a round due to the heterogeneity in computational, communications, and sample size differences. To understand the I/O impact of each client on a standardized scale, it uses min-max scaling to calculate each client's relative I/O utility (Alg. 2, line 14).

Client Experience. The client experience is a composite score of the model and the I/O utility for each client. Since the model utility and I/O utility are normalized to a standardized scale, the FedCaSe scheduler manages to convert the sensitivity of both metrics to a common scale. For example, the loss can be minimal (in fractions), while the time required might be very large. As a result, it becomes challenging to unify both utilities, as a minor change in one might drastically impact a client's overall utility. A relative scale for both metrics helps FedCaSe capture each metric's impact appropriately compared to others. Both I/O utility and the model utility have associated weight factors α and β respectively for placing emphasis on the particular utility required for a job. A sensitivity analysis on the impact of these weight factors has been provided in section 4.5.3. The scheduler maintains a sorted dictionary of client experiences, updating them in real-time after each round (Alg. 2, lines 15-16).

Reverse Optimization (*RO*) Model. In FL, the global model gets impacted by heterogeneous datasets of different clients. Hence, new or novice clients need to take part in training to improve the global model accuracy. A major dilemma arises when we try to balance the experienced:novice (EN) ratio, ρ_r of clients. To address this challenge, FedCaSe scheduler uses the RO model to adaptively change ρ_r to reduce the round duration and improve the accuracy.

The improvement in accuracy can be understood through a stable decrease in the model loss. After every round, FedCaSe proactively checks the decrease in loss of the global model (δ_l) and the round duration (δ_d) using the current round clients ($curr_client$) due to the ρ_r of the previous round (Alg. 2, lines 19-23). It keeps a list, Δ_d for keeping (δ_d)s and another list, Δ_l for keeping (δ_l)s of consecutive rounds. To improve the wall clock time and accuracy, we need to decrease the δ_d and increase the δ_l between rounds. Deciding the increase or decrease of EN ratio to get a decrease in δ_d and an increase in the δ_l requires solving many complicated auxiliary problems that are often computationally expensive [143]. Hence, FedCaSe looks at the problem in reverse and introduces a new technique called reverse optimization (RO).

Since we aim to reduce δ_d and increase δ_l, assume that we have already decreased the δ_d and increased the δ_l. The current value of δ_d is δ_d - ∂t and the current value of δ_l is δ_l + ∂l. Now, the problem that FedCaSe aims to solve is—given that it knows the target round duration ($T\delta_d$), i.e., δ_d - ∂t and target round loss ($T\delta_l$), i.e., δ_l + ∂l, what would be the ρ_r? Hence, the complex problem now gets mapped to a classical regression analysis problem. This regression problem is solved using a regression model by the FedCaSe scheduler.

δ_d and δ_l act as the features (i.e., attributes used to train) of the regression model. In the initial warm-up rounds ($\sim$10), EN ratios are decided randomly to observe the relationship between changing (ρ_r)s with δ_d and δ_l. Since the features are normalized, it improves the stability of the regression model and prevents overfitting.

As the goal is to decrease the δ_d and increase the δ_l, FedCaSe sets the desired base value of δ_d and δ_l as the min and the max of Δ_d and Δ_l respectively. The ∂t and the ∂l are obtained from the standard deviation of the Δ_d and Δ_l (Alg. 2 lines 24-27) to maintain a steady increase in δ_l and decrease in δ_d. In this way, it continuously learns to select the best EN ratio. FedCaSe always chooses the most experienced clients based on this EN ratio, which when determined intelligently, can be anything between 0-1. In each round, if the EN ratio is minimum, then less experienced clients and more new clients are selected, which tackles the overfitting and bias. Additionally, choosing the same client more times reduces the model utility, and hence, biased clients are automatically avoided by RO policy. Once the EN ratio has been predicted, it selects the most experienced clients from the dictionary, $\mathcal{F}$ $\langle$K:client_id, V:client_experience$\rangle$ sorted based on experience that it keeps (Alg. 2 lines 28-40). The impact of RO in FL has been evaluated in section 4.5.3. The *RO* model can be trained online or offline periodically for taking scheduling decisions with negligible overhead ($\sim$0.02% of entire round).

Tackling Bias in Client Selection. Client utility is measured by both model and I/O utility. The I/O utility considers whether a device is high-end by checking how fast it can complete a round for the client. While high-end devices might be favored, this occurs only if high-end devices consist of valuable data samples too. However, this is temporary as their utility decreases through repetitive training. Moreover, to prevent bias the RO policy ensures random client selection according to EN ratio.

Client Data Sampler

The client data sampler works with each client to perform two main functions: (1) calculating the experience score of each data sample belonging to each client; and (2) determining the memory:flash (MF) device ratio, i.e., how many and which samples would be received from memory and flash for training in each round.

To calculate the client sample experience, it uses three techniques. The first technique captures the prospect of a client sample towards improving a model (model utility, M_u) through relative loss. The second technique captures a sample's potential and usefulness towards future participation in training (train utility, T_u) through relative frequency. The third technique calculates a composite sample experience score using the model and train utility that takes into account the impact of both utilities.

FedCaSe client data sampler then uses the sample experience as a metric to determine the number and the identity of the samples that will be used for training from the memory

Algorithm 3: FedCaSe Client Data Sampling

```
1  Input and Initialization:
2  S_L: a set containing loss of each sample of a client
3  S_F: a set containing the frequency of sample usage
4  S_E: a set containing the experience of all samples
5  for s in client_samples do
6      # find model utility, M_u and train utility, T_u
7      M_u = (s_l - min(S_L)) / (max(S_L) - min(S_L))
8      T_u = (s_f - min(S_F))/ (max(S_F) - min(S_F))
9      s_e = M_u + T_u # client sample experience
10     S_E = S_E ∪ s_e # add updated sample experience
11 phase1_samples = prob_dist(S_E)
12 # find current samples in memory and flash cache
13 M_S, F_S = find_experienced_cache(phase1_samples)
14 # R_S: required samples for training
15 r = α * (R_S / len(M_S)) # α = 0.5 by default
16 rep_samples = r * M_S # repeat samples in memory
17 len_flash = R_S - len(rep_samples) # quantity from flash
18 phase2_samples = rep_samples + F_S[:len_flash]
19 return phase2_samples
```

cache and the flash storage in a particular round. Algorithm 3 shows the steps for data sampling inside each client.

Sample Model Utility (M_u). The sample model utility denotes the ability of a sample to contribute towards improving the training model accuracy. It is calculated using training loss like the client model utility. However, to quantify the impact of each sample, FedCaSe uses loss of individual data samples of a client rather than the combined batch-based training loss used in the client model utility.

Sample Train Utility (T_u). The train utility is calculated through the number of times a sample has been trained, i.e., the access frequency of the samples. It captures a sample's potential and usefulness towards future participation in training. Each client keeps a set of the access frequency of each sample and updates that set as training moves on.

Sample Experience. Once the sample model and train utility has been calculated and their relative impact determined, the sample experience, s_e, is obtained from the summation of both utilities (Alg. 3, line 7-9). By incorporating both model and training utility in experience evaluation, FedCaSe adapts dynamically and avoids convergence on a limited data subset.

In contrast to importance-sampling-based techniques [110, 1, 112] that focus on loss only,

this experience-based approach takes into account the impact of both utilities when performing sampling from a probability distribution. As a result, it manages to prioritize samples that have been trained on more and are likely to be in the cache during training, therefore decreasing the wall clock time of the client.

At the same time, when similar samples are selected too often, their model utility decreases. Hence, experience scores give them less priority while choosing for the next round, thus avoiding sample bias. In this way, FedCaSe prioritizes important frequently used samples, but excludes them once their utility decreases. Such a bias-tackling mechanism, combined with insights from important sampling techniques, enables FedCaSe to perform repetitions in sample selection without making the model biased and sacrificing accuracy.

Sampling Procedure. FedCaSe performs sampling in two phases. In the first phase, the sampler collects a list of repetitive samples with the most experience out of all the samples belonging to the client. A probability distribution determines how the experienced and novice samples would be merged for training that will likely produce the best accuracy and round duration improvement (Alg. 3, line 11).

In the second phase, FedCaSe checks the cache for experienced samples, M_s already in the memory cache (Alg. 3, line 13). Then, according to the number of samples required for training (R_s) and the number of samples inside the client's memory cache (M_s), we find a repetition factor (r) that denotes the number of times the experienced samples would be trained more (Alg. 3, line 15). r is reduced by a depreciation factor, α, to maintain a good balance of variation in the samples used to be used for training. Experienced samples undergo further repetitions, ensuring a higher read hit ratio (RHR) for the client's memory cache (Alg. 3, lines 16-19).

Client Data Sample Cache

The client data cache works inside each client and performs two main functions: (1) tracking the experience of each sample in the memory cache and flash cache while dynamically updating the experience scores; and (2) making caching and eviction decisions from the memory cache during training.

Since our goal is to train more on experienced samples from a client's limited memory cache to reduce I/O time, we must always keep the memory cache occupied with the most experienced samples so that repetitions of those samples during sampling (Alg. 3) can be beneficial for training. For tracking the experience of each sample belonging to a client, FedCaSe maintains two sorted metadata queues - the current queue (CPQ) and the ghost cache (GPQ) based on the loss, l_s, and the number of times a sample has been trained, f_s. These queues store l_s and f_s as a metadata tuple $\langle l_s, f_s \rangle$. The sample experience, e_s, is calculated from these queues during taking caching and eviction decisions (Alg. 4, lines 10-14). The CPQ tracks the experience associated with every client sample inside

Algorithm 4: FedCaSe Client Data Sample Caching

```
1  Input and Initialization:
2  # Initialize priority queue of (loss, frequency) of samples in memory and trained
     samples, respectively
3  CPQ: {},    GPQ: {}
4  for s in client_samples do
5      if s in memory_cache then
6          fetch_from_memory(s)
7          l_s = get_loss_from_training() , f_s = CPQ[s][1]
8          CPQ[s] = (l_s, f_s+1)
9      else if s in GPQ then
10         l_s, f_s = GPQ[s]
11         # Calculate using Alg. 3 (line 6-9)
12         e_s = get_sample_experience(l_s, f_s)
13         s_min, (l_min, f_min) = arg min(CPQ), min(CPQ)
14         e_min = get_sample_experience(l_min, f_min)
15         if e_s > e_min then
16             evict(s_min) & CPQ.remove(s_min)
17             l_s = get_loss_from_training()
18             cache(s) & CPQ.insert(s, (l_s, f_s + 1))
19         else
20             fetch_from_flash(s)
21     else
22         fetch_from_flash(s)
23     l_s = get_loss_from_training(), f_s = GPQ[s][1]
24     GPQ[s] = (l_s, f_s + 1)
```

the cache, and the GPQ tracks every client sample that has already participated in the training. During the training process, if a certain sample's experience is more than the sample having the lowest experience inside the current memory cache, then the sample having the lowest experience is evicted from the cache, and the new sample having the higher experience score is cached (Alg. 4, lines 15-22). This way, the FedCaSe client cache dynamically updates the memory cache.

Memory Overhead. Each client maintains two priority queues having $\langle$float, float$\rangle$ tuples. Assuming that each client has enough space to store 100B samples (each 1MB), the total space available to the client would be $97.7 * 10^6$GB. Assuming storing each tuple will need 8 bytes, the memory overhead for maintaining the queues is very negligible ($\sim 0.001\%$ of the entire space required to keep the samples).

4.4 Implementation

FedCaSe is implemented as three separate Python libraries for client scheduling, client data sampling, and client data caching built to work with a popular FL engine, FedScale [136]. The client scheduler determines the EN ratio by using the RandomForestRegressor model from scikit-learn [144]. The client data sampling library and the caching library work with the clients of an FL engine. FedCaSe uses PyTorch [123] multinomial and numpy [145] library to build the logic behind client data sampling. Each client builds its dataset by inheriting PyTorch's Dataset class. FedCaSe has APIs for communicating with the memory cache to conduct caching and eviction decisions inside the Dataset class.

4.5 Evaluation

4.5.1 Experimental Setup

FedCaSe has been designed to work in extensive deployments with millions of edge devices. Nevertheless, such deployment is both cost-prohibitive and challenging to guarantee the reproducibility of experiments. Hence, we resort to training with 2800 emulated clients with the selection of up to 100 clients per round using NVIDIA P100 GPUs on Chameleon Cloud testbed [119]. We simulate real-world FL training using two representative computer vision datasets—FEMNIST [139] and CIFAR-10 [114] and two models, ResNet-18 [126] and MobileNetV2 [146] that are widely used in FL evaluations. Train and test datasets are partitioned following LEAF [139] benchmark.

Clients in our study have sample sizes ranging from 1 MB to 100 MB and capture the inherent data sample size heterogeneity in cross-device FL. Figure 4.7 shows the CDF of the average sample sizes of the different clients used for running our experiments. Moreover, the clients vary in the amount of available memory cache, i.e., different clients can cache different numbers of samples. Figure 4.8 shows the heterogeneity in memory cache space across the clients normalized based on amount of samples that can be fit in memory, i.e., data sample quantity heterogeneity. The heterogeneity related to computing and communication is simulated using data provided by FedScale [136] which uses data from AI Benchmark [147] and MobiPerf [148] to replicate real-world FL deployments.

To show the effectiveness of the FedCaSe's caching policy, we need to allow clients to use a particular portion of their memory for caching samples. Although memory cache for samples can be fixed at any percentage as necessary by the client, to ensure consistency and reproducibility, in our evaluations, we allow each client to cache 10% of its WSS in its memory cache. Note that since each client has a different number of samples as their WSS, the cache sizes become heterogeneous for the clients. Assume three clients have WSS sizes $\langle 50,100,150 \rangle$; then caching 10% of WSS would mean the number of samples they can cache

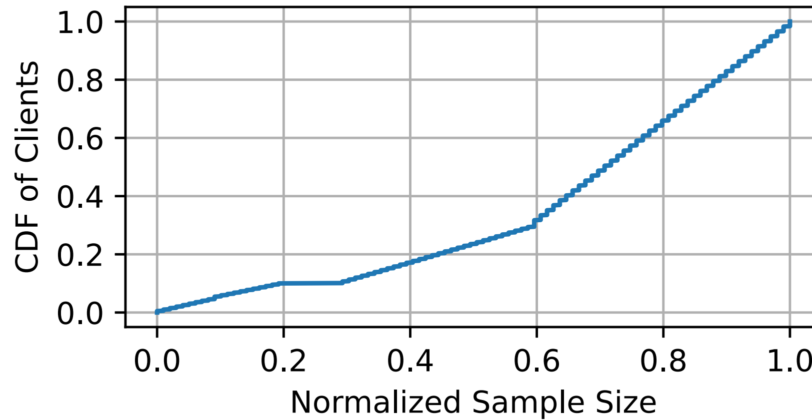


Figure 4.7: Sample sizes of clients differ greatly. Sample sizes normalized with respect to their sizes across x-axis.

would be 5, 10, and 15, respectively. We report the simulated wall clock time and rounds in our evaluations based on the capabilities of the devices. While the operation time for individual devices is estimated, the actual training is performed on real data samples using real GPUs to enhance model accuracy.

Our evaluation aims to address the following questions:

- How much can FedCaSe increase the global and local RHR in large-scale heterogeneous FL? What is the impact of each of its component on RHR? (section 4.5.2)
- How can the improved RHR translate into improved model accuracy during FL training through exploiting client and sample experience? How much adaptive is the Reverse Optimization (RO) policy throughout the training process? (section 4.5.3)
- How much can FedCaSe decrease the round duration in FL? How much does each of its utility functions contribute to reducing the number of rounds? (section 4.5.4)

4.5.2 Impact on Read Hit Ratio (RHR)

In this section, we evaluate FedCaSe's client data sample caching policy against other state-of-the-art caching policies used in traditional homogeneous settings. We evaluate the RHR of FedCaSe against three state-of-the-art policies—SHADE [1], Quiver [13], and MinIO [12] along with traditional policies like LRU and LFU (least frequently used) to show how much FedCaSe can improve the global and local RHR of thousands of clients having heterogeneous memory cache. We also perform an ablation study on FedCaSe's different policies (C: Caching, S: Sampling, Sched: Scheduling) to understand how much each policy impacts the RHR.

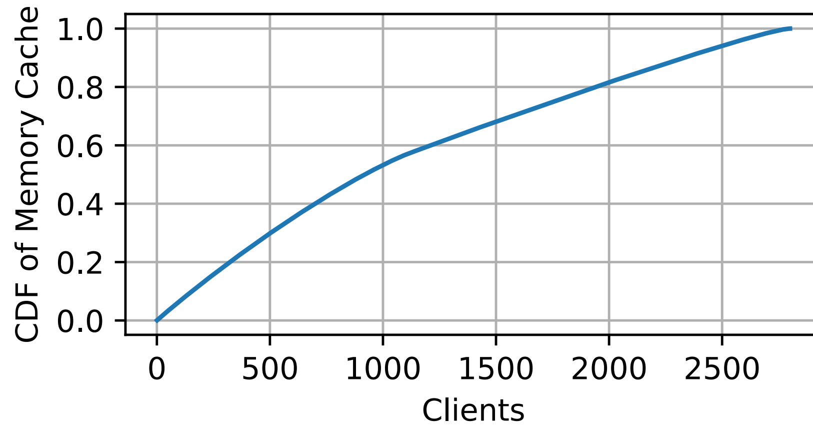


Figure 4.8: Memory cache space of the clients differ greatly. X-axis denotes the Client IDs.

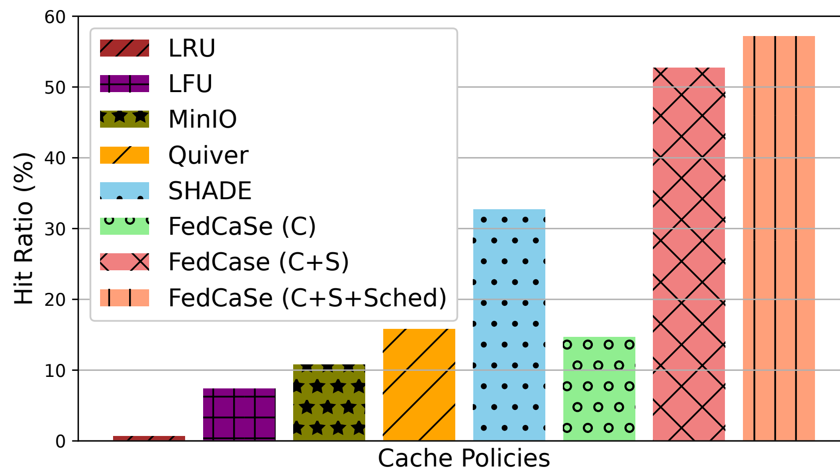


Figure 4.9: Global read hit ratio (RHR) of different caching policies in a heterogenous FL setup of 2800 clients. Ablation study on three components of FedCaSe (C: Caching, S: Sampling, and Sched: Scheduling).

SHADE clients (i.e., clients using SHADE) use a sample priority-aware caching technique for caching samples in their memory. Quiver clients use a substitutability technique that prioritizes samples already in the client's cache for training. MinIO clients do not evict samples once they are cached in memory. FedCaSe differs from these policies in two aspects—it samples based on clients' heterogeneous memory cache space and caches based on sample experience, which improves its RHR and accuracy. The clients are scheduled using both the default scheduling technique (i.e., random) and a state-of-the-art scheduling technique, Oort [48].

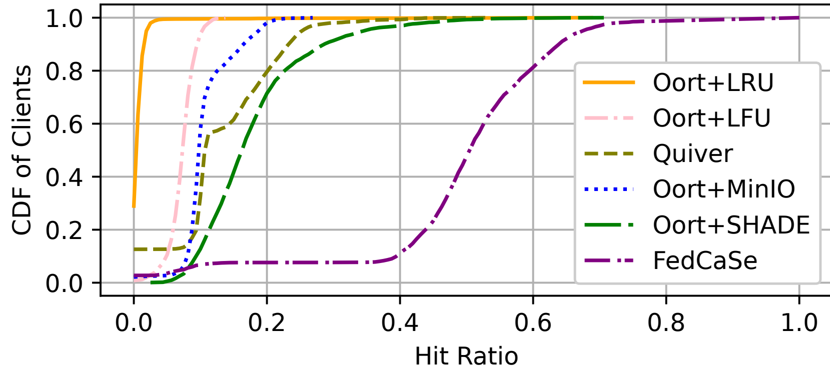


Figure 4.10: FedCaSe outperforms other state-of-the-art policies in local RHR at the scale of thousands of heterogeneous clients.

Ablation Study. Figure 4.9 shows that FedCaSe (C) performs similarly to Quiver. However, when we consider memory heterogeneity in the sampling procedure and adjust the experienced samples based on each client's limited memory cache size, FedCaSe (C+S) improves performance by 3.59×. Including experience-based scheduling, FedCaSe (C+S+Sched) further enhances the performance, ensuring a 1.75×, 3.62×, 5.29×, 7.73×, and 81.72× higher RHR compared to SHADE, Quiver, MinIO, LFU, and LRU cache policies respectively when trained for 150 rounds. LRU and LFU policies perform the worst in RHR as these policies cannot exploit random data sampling patterns of thousands of clients in the limited heterogeneous cache. Since SHADE uses loss-based sampling to enhance RHR, it performs better than the other policies. However, like other policies discussed in section 4.2, SHADE cannot adjust the sampling and scheduling to match the heterogeneity of the client devices. FedCaSe's experience-based scheduling policy prioritizes the usage of warmed-up clients in the future, thus outperforming SHADE in global RHR.

Next, we examine the individual contributions of each client, i.e., local RHR, compared to the collective gain in global RHR across the entire set of 2800 clients. Figure 4.10 illustrates that in FedCaSe, 92.39% clients have a RHR greater than 0.2, which is 318.58×, 48.37×, 3.65×, and 3.24× more compared to LRU, MinIO, Quiver, and SHADE respectively. Lack of a sample retention policy like FedCaSe means new samples displace existing ones in the cache, resulting in a high eviction rate. Since FedCaSe learns from previous patterns in training through sample experience, it can properly guide the sampling procedure of clients with limited heterogeneous memory to increase the local RHR.

4.5.3 Impact on Accuracy Improvement

To investigate whether the gains in RHR shown in the previous section translate well into accuracy improvement, in this section, we evaluate the accuracy improvement of

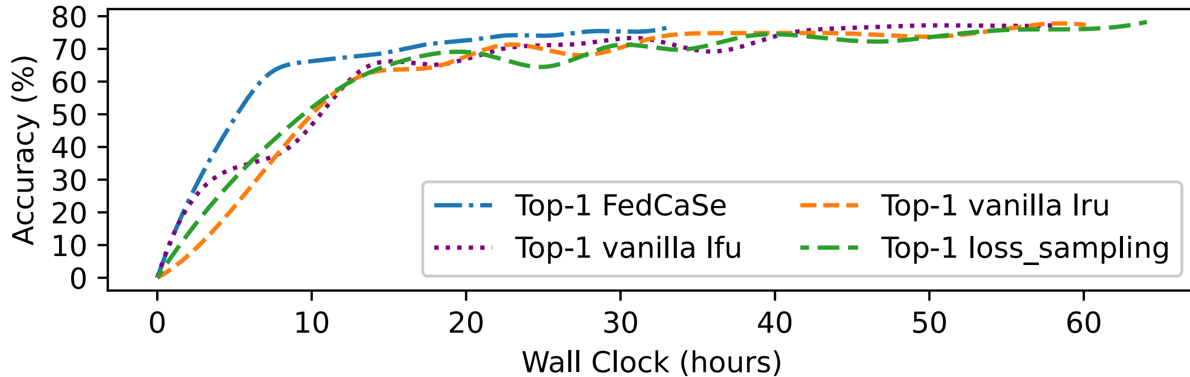


Figure 4.11: Accuracy improvement of FedCaSe vs. `vanilla` configurations using ResNet-18 on FEMNIST.

FedCaSe against random and advanced sampling and scheduling techniques. We use top-1 accuracy for evaluation, which denotes the percentage of predictions where the model's top prediction matches the true label of the input data. We train up to the point when the accuracy curve starts to plateau, indicating convergence.

In the first set of experiments, we evaluate FedCaSe against three baselines based on random client scheduling, data sampling, and traditional caching policies:

1. `vanilla lfu`, which performs client scheduling and client data sampling randomly. However, clients can put 10% WSS in the cache and evict the data samples based on the LFU policy when updating the cache.
2. `vanilla lru`, which performs client scheduling and client data sampling randomly. However, clients can put 10% WSS in the cache and evict the data samples based on the LRU policy when updating the cache.
3. `loss_sampling`, which uses a loss-based advanced sampling policy like SHADE [1] and Mercury [26], and it is equipped with a caching policy that caches samples based on loss importance and evicts samples randomly when looking to cache important samples.

We observe the accuracy improvement rate of FedCaSe and the baselines during training. Figure 4.11 shows that FedCaSe, caching 10% WSS, is up to $1.85\times$ faster than `vanilla` and `loss_sampling`. These baselines are unable to choose samples and adapt the sampling conveniently to improve accuracy when clients have a limited heterogeneous memory cache. Although `loss_sampling` chooses samples based on importance, it cannot fully exploit the opportunities that important samples present at client granularity. While FedCaSe decides the experience based on model utility and train utility of a sample, `loss_sampling` decides the samples for the next round based only on loss and disregards

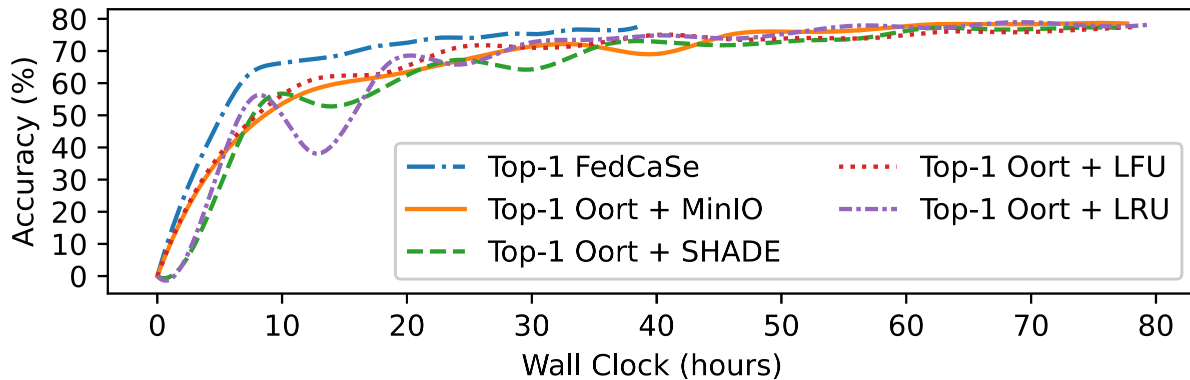


Figure 4.12: Accuracy improvement rate of FedCaSe vs. Oort. Training ResNet-18 on FEMNIST for 150 rounds.

a crucial metric, i.e., train utility to filter samples. As train utility is not considered, `loss_sampling` cannot reward experienced samples based on I/O benefits and hence suffers from increased round durations.

Next, we compare FedCaSe's client scheduling technique against four additional baselines using Oort [48], a state-of-the-art client scheduling technique. Oort uses loss generated by clients to detect client importance, thereby using that as a driving metric for selecting clients in the next rounds. Oort penalizes clients which become stragglers but does not reward clients which complete rounds faster, like FedCaSe. Hence, Oort does not fully capture a client's ability to reduce round duration. Additionally, Oort prioritizes exploring more clients rather than training more on already experienced clients like FedCaSe. This approach makes Oort leave out on gaining the most benefits from a single client due to lack of training as previously discussed in section 4.2.3.

For evaluation, we equip Oort with four caching policies to show the impact FedCaSe's experienced clients have on accuracy improvement against advanced policies:

1. `Oort + MinIO`, which uses a caching policy similar to MinIO [12] where cached samples are never evicted.
2. `Oort + SHADE`, which leverages SHADE [1], a policy using importance of samples to take eviction decisions.
3. `Oort + LFU`, which evicts the data samples based on the LFU policy when updating the cache.
4. `Oort + LRU` evicts the samples based on the LRU policy while updating the cache.

Figure 4.12 shows that FedCaSe takes 2.02×, 2×, 2.04×, and 2.06× less time than `Oort + MinIO`, `Oort + SHADE`, `Oort + LFU`, and `Oort + LRU` respectively to reach accuracy convergence.

Table 4.2: The improvement of FedCaSe in number of rounds clients get scheduled in different ranges over vanilla and Oort. $\geq$ X means a single client gets called $\geq$ X times throughout training.

Client Calls	Improvement (vanilla)	Improvement (Oort)
≥ 10	1.51$\times$	3.01$\times$
≥ 15	3.28$\times$	6.66$\times$
≥ 20	19.61$\times$	29.1$\times$

To understand why accuracy improved faster, we further analyze client participation compared to `vanilla` and Oort configurations across different ranges. Table 4.2 shows that FedCaSe can call experienced clients over 20 times (i.e., the same client takes part in entire training for over 20 rounds) throughout training 19.6$\times$ and 29.1$\times$ more compared to `vanilla` and Oort respectively. As Oort prioritizes clients with a higher impact on accuracy improvement, it explores more clients which are inexperienced. In contrast, FedCaSe uses RO policy to find experienced clients across rounds. Figure 4.13 shows that the RO policy adapts itself (40% to 90% of total client selection can be experienced) to minimize the training time and maximize the accuracy improvement while exploiting the limited memory cache of clients.

Figure 4.14 shows that the same clients are scheduled more frequently (some over 100 times) in FedCaSe compared to Oort. As Oort places less emphasis on the I/O utility of clients and more on exploration, it ends up with a suboptimal list of clients for decreasing the I/O time in each round. On the other hand, as FedCaSe's policy is to train more on experienced clients which already have experienced samples stored in the cache, it ensures faster accuracy improvement.

Sensitivity Analysis. Two parameters that affect the client experience, and thus the client scheduling, are the tunable weight parameters α and β of the client's I/O utility and model utility, respectively. We change the values of α between [0,1], where β = 1 - α, and train for 150 rounds on the FEMNIST dataset using ResNet-18. Figure 4.15 illustrates that as α increases, both training time and top-1 accuracy decline. This trend highlights that clients that minimize training time are prioritized.

4.5.4 Impact on Rounds

In this section, we will evaluate how FedCaSe's policies impact the number and duration of rounds in FL. Given that Oort prioritizes exploring clients, which can increase accuracy, it is expected to reach accuracy convergence in fewer rounds. However, each round takes

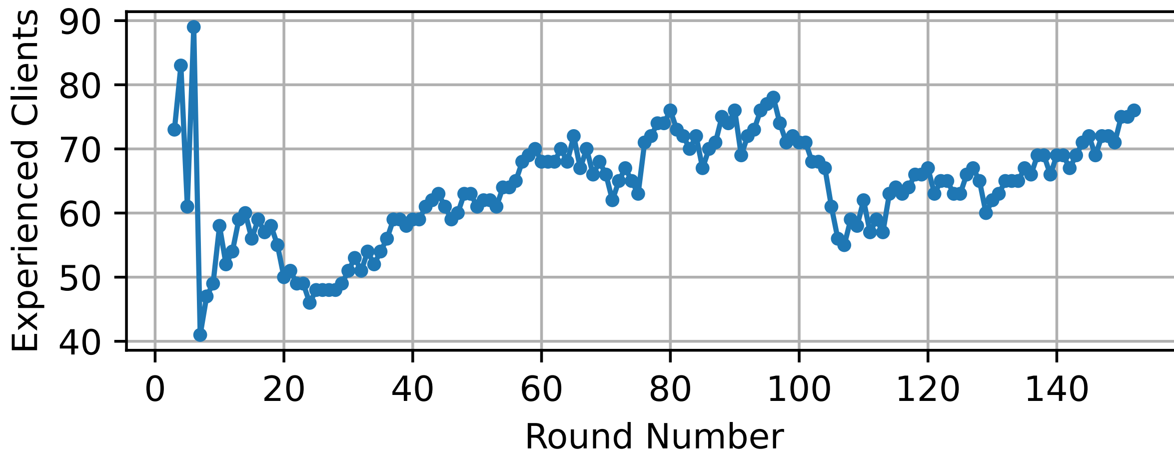


Figure 4.13: Adaptivity of RO policy in scheduling experienced clients.

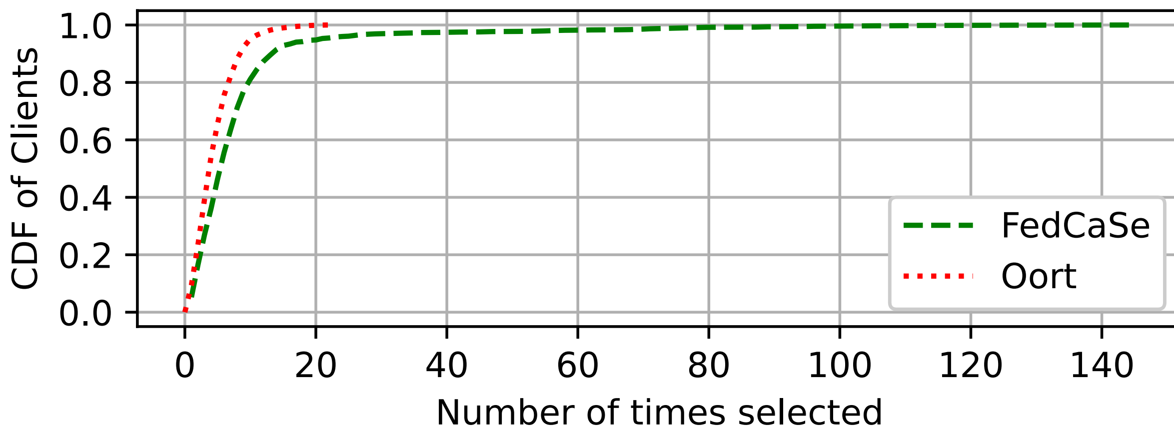


Figure 4.14: Experienced clients get scheduled for participation in future training more frequently in FedCaSe.

significantly longer as I/O utility is only partially considered in client scheduling.

When training ResNet-18 on FEMNIST shown in Figure 4.12, we observe that although FedCaSe requires 1.25× more rounds to reach Oort's accuracy, FedCaSe's round duration is 2.04× less. Figure 4.16 shows that although FedCaSe takes $1.16\times$ more rounds than Oort when training ResNet-18 on CIFAR-10, its round duration is $2.4\times$ less as it strikes a good balance between model and I/O utility of clients. Hence, FedCaSe can quickly increase its accuracy improvement rate (1.6×) compared to Oort based on wall clock time.

We perform another round of experiments after equipping Oort with three cache policies:

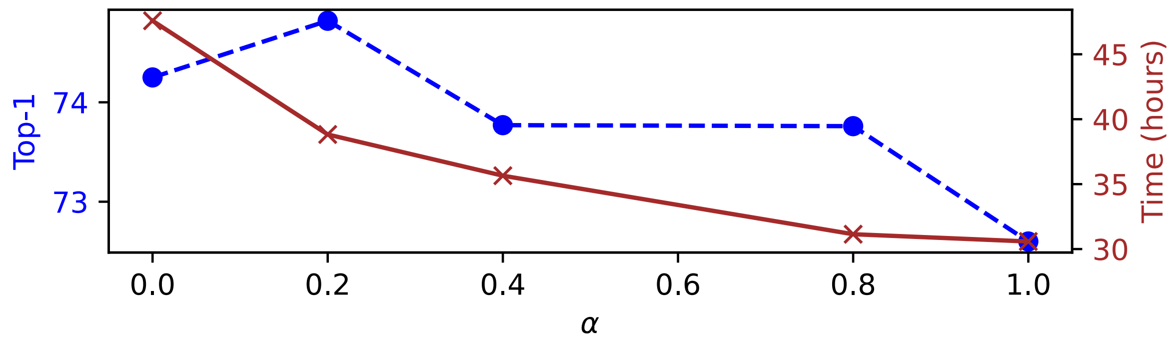


Figure 4.15: Sensitivity analysis on the impact (Top-1 accuracy and training time) of placing weight, α on I/O utility when determining client experience.

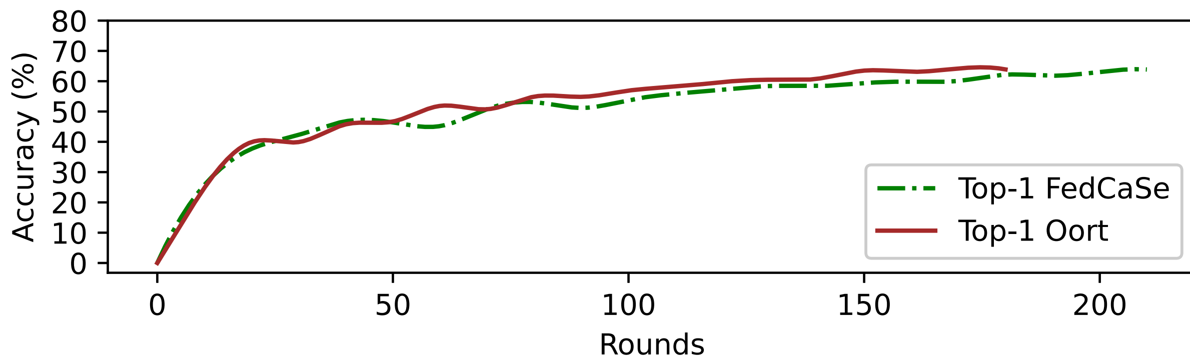


Figure 4.16: Accuracy improvement rate of FedCaSe vs. Oort with respect to rounds. Training ResNet-18 on CIFAR-10.

LRU, LFU, and SHADE, to visualize the average round time taken by these baselines using MobileNetV2 model on FEMNIST. Figure 4.17 shows that FedCaSe round time is 2.06×, 1.96×, and 1.87× less compared to `Oort + LFU`, `Oort + LRU`, and `Oort + SHADE` respectively. As discussed in section 4.5.2, FedCaSe's ability to increase the global RHR significantly reduces the round duration.

Ablation Study. By default, FedCaSe prioritizes both the model and I/O utility during data sampling and client scheduling. Hence, although it can take a few more rounds to reach accuracy convergence compared to Oort, the wall clock time taken decreases significantly. However, users can also empirically prioritize the model utility during data sampling and client selection if accuracy convergence needs to be achieved in fewer rounds. Moreover, if users do not want to use a cache, or does not have a sufficient one, they can prioritize model utility to get the maximum performance. We try to understand the impact of model and I/O utility in decreasing the number of rounds during training through

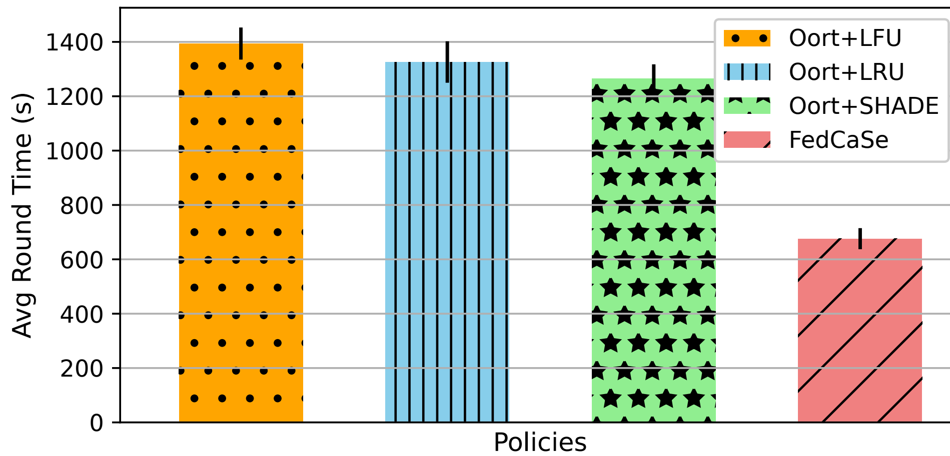


Figure 4.17: FedCaSe vs. Oort round time improvement using different cache policies during training MobileNetV2 on FEMNIST.

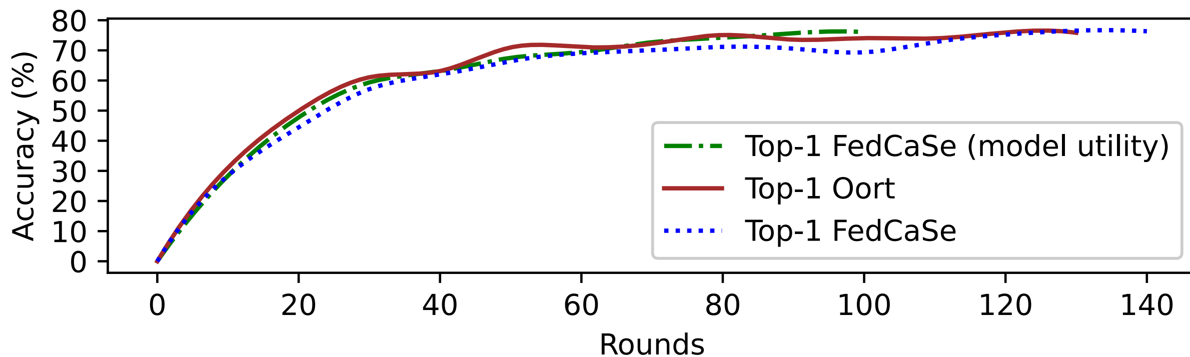


Figure 4.18: Accuracy improvement rate of FedCaSe vs. Oort with respect to rounds. Ablation study on FedCaSe's model and I/O utility fucntions. Training ResNet-18 on FEMNIST.

an ablation study where FedCaSe uses both model and I/O utility and FedCaSe (model utility) uses only model utility.

Figure 4.18 shows that prioritizing model utility can enable FedCaSe to reach accuracy convergence up to 1.4× faster compared to Oort with respect to rounds. This result also shows that even without a cache (i.e., 0% WSS in cache), FedCaSe's client utility measuring mechanisms can boost the performance of training. While Oort prioritizes selecting clients with a higher potential to increase accuracy, it performs random data sampling within clients. Instead of random sampling, FedCaSe emphasizes model utility during both data sampling and client scheduling, which results in faster convergence with respect to rounds.

4.6 Assumptions and Limitations

In this section we discuss the assumptions and limitations that may influence the scope and applicability of our findings to provide a comprehensive understanding of our approach.

Client Incentivization. Client incentivization is a separate branch of FL research that is orthogonal to our work, as we assume voluntary client participation, similar to related studies [48, 49, 149]. In future work, we plan to extend FedCaSe with incentive mechanisms to encourage client contributions based on observed utilities.

Broader Applicability of the Proposed Solution. Our approach is applicable to NLP tasks, as NLP models can use loss during training and large text documents can cause I/O bottlenecks. However, for NLP workloads, we need to make trade-offs between the importance of tokens, sentences, or sequences during calculating model utility. In tasks like translation, summarization, or question-answering, sentence pairs may influence caching decisions. We plan to explore these challenges in future work.

4.7 Summary

System resource heterogeneity across the network, memory, computing power, etc., poses a significant challenge in improving performance across millions of FL client devices. Furthermore, heterogeneity in data sample size and quantity exacerbates the issue. While several policies address specific aspects of heterogeneity, none address how to navigate a crucial heterogeneous resource—memory across millions of clients. FedCaSe is the first to realize a unified intelligent client scheduling, data sampling, and caching solution for millions of client devices having heterogeneous limited memory sizes. FedCaSe leverages client experience and sample experience, calculated based on the ability to increase model accuracy and decrease round time to intelligently and adaptively drive the client scheduling, data sampling, and caching decisions. Thus, it improves the accuracy improvement rate by up to 2.06× and increases the read-hit ratio by up to 81.72× globally compared to state-of-the-art client scheduling and caching policies.

Chapter 5

Workload-aware Job Scheduling & Fairness in LLM Inference

5.1 Introduction

In recent years, multi-tenant personalized large language model (LLM) serving platforms, which support a wide range of applications like interactive question answering (QA), summarization, coding assistance, etc., have surged in popularity, with notable examples including Copilot [150], Punica [151], and S-LoRA [14]. On these serving platforms, some users may flood the system with excessive requests, causing service disruptions for others and leading to considerable unfairness. For instance, on February 13, 2024, OpenAI reported a partial outage lasting over five hours for its API and ChatGPT services due to DDoS attacks, which recurred with varying durations in the following days [11]. Therefore, to maintain service availability and fairness, it is crucial to prevent abusive usage and ensure fair distribution of resources among users on LLM platforms.

Existing LLM platforms enforce limits on the number of requests each user can submit within a given period, such as requests per minute (RPM) to address the challenge of abuse prevention and ensuring fairness. For example, Google Gemini for Workspace limits usage to 500 times per month [9], and OAI Chat GPT Plus imposes limits of 80 messages every 3 hours on GPT-4o and up to 40 messages every 3 hours on GPT-4 [61]. However, such rate-limiting solutions have drawbacks. First, modern LLM applications are built from multiple LLM agents, each operating with an LLM model, working together to produce a single response for the user. The graph of interconnected LLM call requests is termed an LLM interaction. Throttling at the LLM request-level without understanding the application's needs can result in incomplete user responses and resource wastage. Second, requests can still be throttled even if the system is underloaded, causing resource underutilization and user frustration.

The Virtual Token Counter (VTC) scheduler prioritizes requests of users who receive the minimal service when processing batches, aiming to prevent unfairness [6]. However, unlike RPM, VTC does not throttle or block users, thus making it unable to fully prevent abusive behaviors. Based on the principle of fair-queueing [152], VTC gives both benign and abusive users equal service, leading to resource wastage, longer request queues, increased latency, and user frustration.

Table 5.1: Comparison of FAIRSERVE and current methods.

Method	**Maintains user experience**	**Avoid resource under-utilizations**	**Block abusive behaviors**	**User & App character-istic-aware service calculation**	**Reduce resource wastage**	**Consider multi-agent LLM apps**
RPM [9, 61]	×	×	✓	×	×	×
VTC [6]	×	✓	×	×	×	×
FAIRSERVE	✓	✓	✓	✓	✓	✓

Additionally, modern LLM platforms receive requests from users having varying needs as they belong to different applications. Different applications have distinct typical ranges for input and output token lengths. For instance, article summarization applications often involve long inputs and short outputs, while code generation applications typically feature short inputs and long outputs. Hence, when scheduling requests for users across diverse apps, an ideal policy would be application-characteristic aware to ensure fairness and at the same time curb abusive behavior.

To gain a deep understanding of these problems, we conduct a comprehensive analysis on MS CoPilot, a leading real-world multi-tenant personalized LLM serving platform. Our investigation (§ 5.2) confirms the limitations of existing methods and highlights the need for a more sophisticated approach to ensure fairness amid diverse applications. Based on these findings, we designed a novel system, FairServe, which is tailored to ensure fair access to LLM resources amid diverse applications. FairServe consists of two core components: (1) Overload and Interaction-driven Throttling (OIT); and (2) Weighted Service Counter (WSC) scheduler which collaborate together to serve user requests.

Unlike conventional throttling solutions, OIT throttles requests only when the system is overloaded, maximizing resource utilization. Additionally, it implements throttling at the LLM interaction-level (a single LLM interaction may consist of multiple simpler LLM calls), leveraging application characteristics to prevent token wastage and curbs abusive behavior of user across apps by setting application-specific limits. The WSC component complements OIT by selecting requests from the users who have received the least service, defined by a weighted resource slice. This weight is calculated based on the token ratio to ensure fairness across users. Note that, Unlike the equality-centric approach proposed by

VTC, our method prioritizes serving users who have received the least service, building on an equity-centric approach [153] that addresses the diverse needs of users across different applications. Table 5.1 compares the effectiveness of the existing methods against FairServe to prevent abusive behaviors and achieve fairness.

Specifically, this chapter makes the following contributions.

- We conduct large-scale analysis on millions of requests across 34 applications on a leading multi-tenant LLM platform, revealing intricate characteristics of modern LLM applications and guiding future research.
- We introduce a novel approach, Overload and Interaction-driven Throttling (OIT), for throttling LLM requests to curb abusive behavior.
- We design a novel mechanism, Weighted Service Counter (WSC) for scheduling LLM requests to ensure fairness amid diverse applications.
- We present the design and implementation of FairServe—a new LLM serving system that integrates WSC and OIT for fairness and abuse prevention.
- We integrate FairServe in a open-source LLM serving platform [14] using state-of-the-art iteration-level continuous batching [15] and compare against a series of policies for LLM serving fairness. Our results on a testbed of over a thousand users show that FairServe reduces waiting queue delays by $10.67 - 93\times$, decrease latency (time-to-first-token) by $1.03 - 1.06\times$, increase throughput by $1.03 - 1.75\times$ (with 0% token wastage) across apps while curbing abusive behaviour and enabling better service to users by $99.45 - 100\%$.

Efforts are underway for deploying FairServe in production and making the source code of our prototype available to the research community to foster further research and development in this critical area.

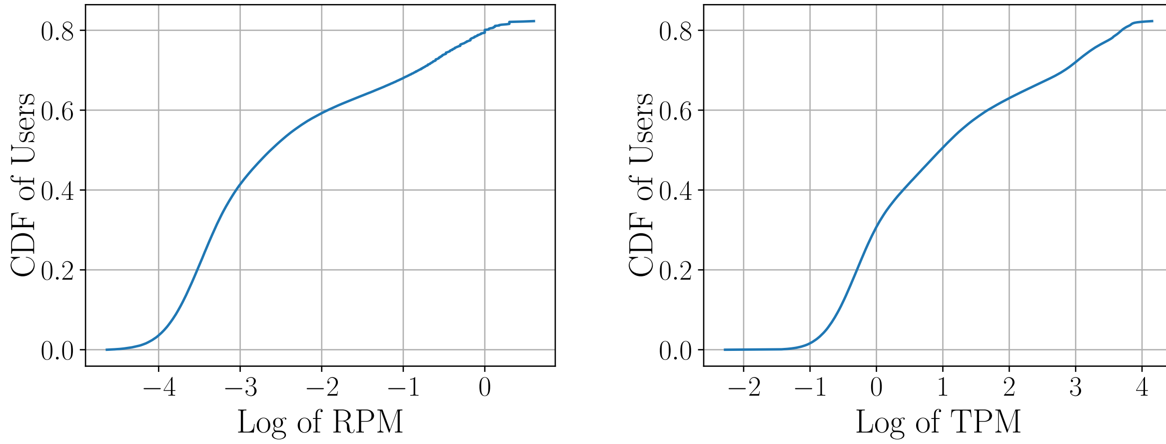
5.2 Production Workload Analysis

We analyze a full day of trace data from 34 distinct applications, covering millions of requests[1]. We investigate the impact of user behavior, application behavior, LLM agents, and throttling techniques to gain insights into designing a fair LLM serving system.

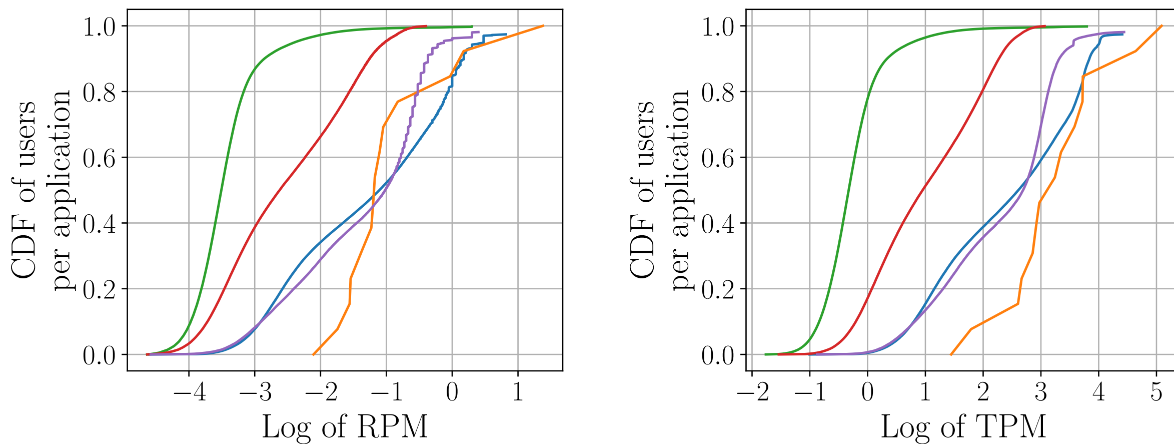
5.2.1 Impact of User Behavior

We first investigate user request patterns, specifically assessing if requests from users and applications follow any particular trends with the goal to curb abusive behaviour and ensure fairness across users.

[1]We have anonymized the exact volume and application names due to confidentiality reasons.



(a) Overall distribution of RPM across different users. (b) Overall distribution of TPM across different users.

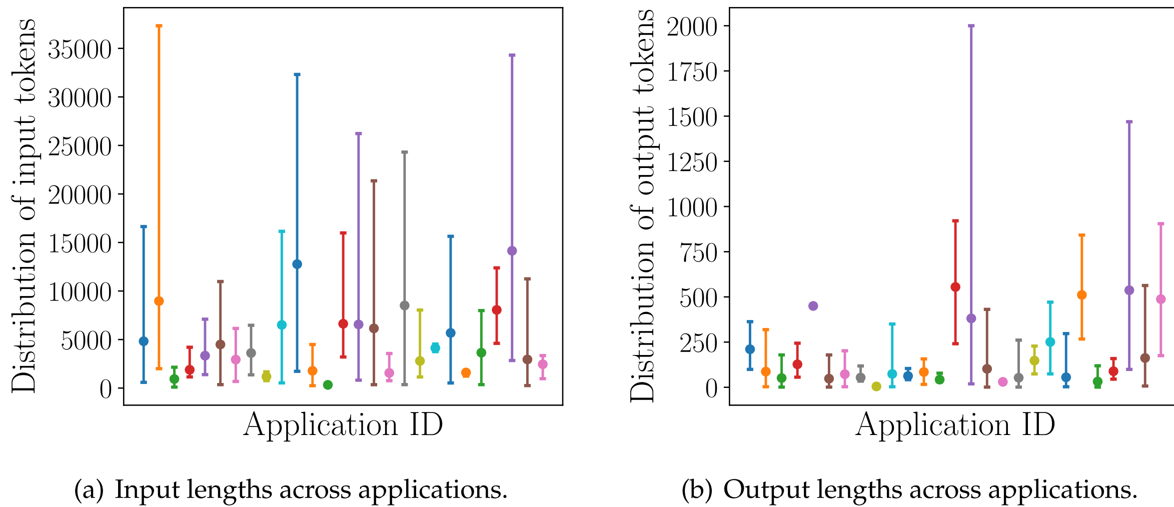


(c) RPM across different users for different applications. (d) TPM across different users for different applications.

Figure 5.1: Users' varying RPM and TPM across apps make prior policies ineffective for ensuring fairness and deterring abusive behavior.

First, we profile the CDF of users by requests per minute (RPM) and tokens per minute (TPM) to identify potential abusive behaviors. Figures 5.1(a) and 5.1(b) show that the overall request frequency and number of tokens sent by a user tends to fall in a certain range (99th percentile for RPM is below 10^2). However, the maximum RPM and TPM observed is upto 10^4 and 10^6 respectively, showing signs that different users have different trends. In this case, a rate-limiting policy may mark a lot of users as abusive where as in reality this could be due to difference in their trends.

Next, fine-grained analysis into the users of different applications reveal that users behave differently across applications in terms of RPM and TPM. For example, Figure 5.1(c) (each



(a) Input lengths across applications.

(b) Output lengths across applications.

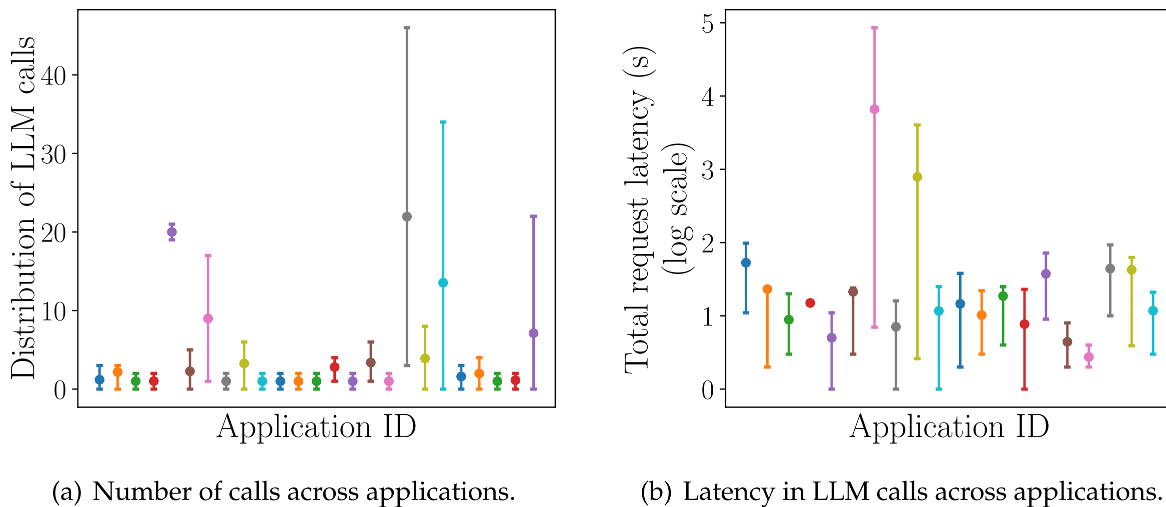
Figure 5.2: Token counts differ across applications, suggesting that LLM scheduling must consider variations in user applications.

curve is for an app and each point corresponds to a user) shows that although one app has only around 20% of its users' RPM to be less than or equal to 10^{-1}, for others it could be well above 80%. Again, Figure 5.1(d) shows that although around 80% of the users of some app have a TPM of 10^2 or less, for other apps it could be just 20%. Both of these observations mark a stark dissimilarity in the RPM and TPM limits across applications, prompting us to treat users belonging to different applications differently. Hence, trying to maintain equal TPM across all users like VTC [6] in ensuring fairness might not be ideal and will not be sufficiently effective to deter abusive behavior.

Takeaway 1. *A user's overall RPM and RPM for each application falls in a certain range and these are different from one another. As benign users have different tokens per minute (TPMs), simply targeting equal TPMs across users is not sufficiently effective to deter the abusive behavior or achieve fairness.*

5.2.2 Impact of Application Behavior

In this section, we closely examine the input, output, and total token lengths across various applications. Our aim is to detect token patterns that will aid in designing a fair weighting system for users of different apps. Figure 5.2(a) illustrates that the average input lengths differ significantly across applications, with each having a specific range within which all its requests fall. Although Figure 5.2(b) shows that the output lengths across applications are more similar than the input lengths, there are still some variations. This insight pushes us to reevaluate how we assess user service across different applications. We propose



(a) Number of calls across applications. (b) Latency in LLM calls across applications.

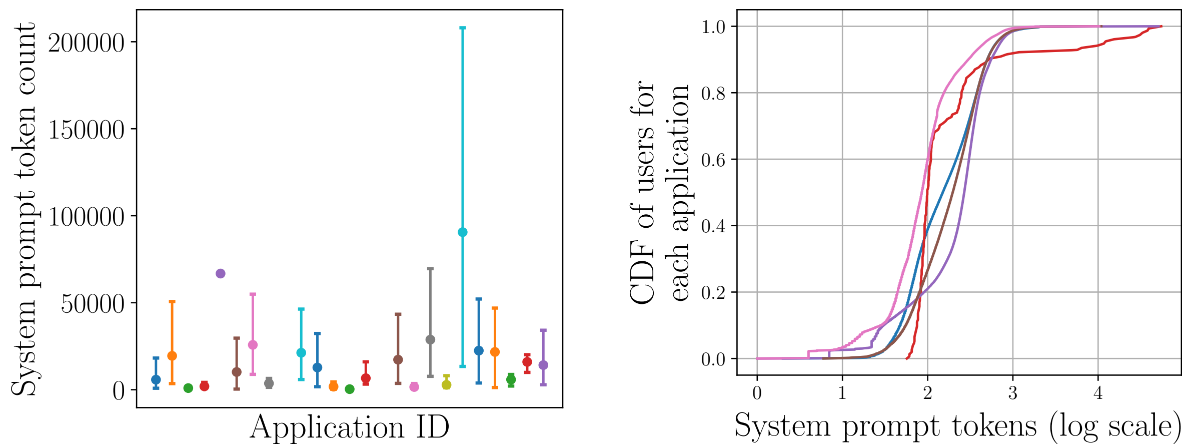
Figure 5.3: The variability of LLM calls across apps must be considered to reduce latencies and queueing delays in multi-agent apps.

shifting from a uniform token weighting to a strategy that assigns distinct weights based on application behaviors. For instance, consider two applications, A and B, where the average total request lengths are 10 and 50 tokens, respectively. If 5 tokens are processed for each application at a given moment, we calculate that users from app A and B have received 5/10 (i.e., 50%) and 5/50 (i.e., 10%) of their respective service as that is the portion of the request that gets completed for these users. This method differs from the prior fairness policy [6], which would have allowed both users to receive an equal amount of service at that stage, and hence would treat them equally when scheduling requests.

Takeaway 2. *Each application has its own normal range of input and output token lengths for requests, which vary across different applications. This insight indicates that service calculations during LLM request scheduling must account for the application-level differences between user requests.*

5.2.3 Impact of LLM Agents and System Prompts

Next, we investigate the impact of LLM agents and system prompts on generating the final response for a user request. Figure 5.3(a) shows that across applications the number of LLM calls vary, i.e., the same application can generate different number of LLM calls (can be over 20) using LLM agents, depending on the context of the user request. Furthermore, Figure 5.4(a) shows that: (1) across applications the system prompt token count varies, and (2) even within the same application, the same LLM agents can generate different number of system tokens. Moreover, Figure 5.4(b) shows that these system prompts coming from



(a) System prompt token in LLM calls across apps. (b) Output tokens in system prompts across users.

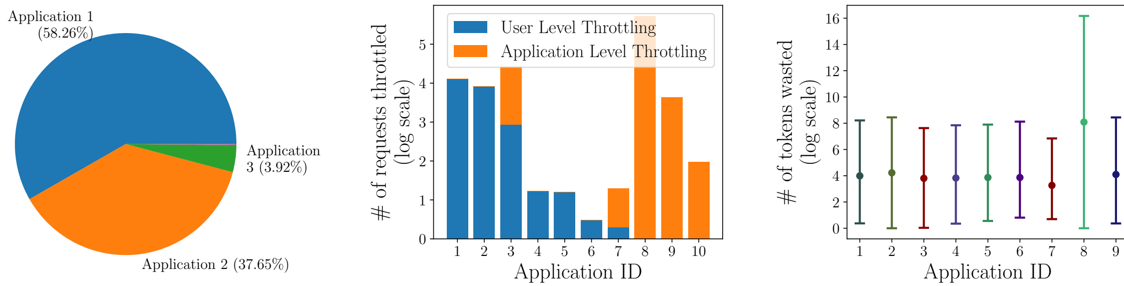
Figure 5.4: System prompts lead to varying characteristics in total tokens of each interaction and the number of output tokens.

LLM agents can lead to the generation of different number of output tokens. In particular, we observe that 20% of the users of four applications are generating over 500 output tokens. Additionally, Figure 5.3(b) shows that these LLM calls have varied latencies (can be over 6 seconds), depending on the position of the LLM call request in the waiting queue for the interaction. Long latencies are undesirable for applications with tight SLOs, stressing the need for proper placement of LLM call requests during scheduling in order to reduce latencies and queueing delays for multi-agent LLM requests.

Takeaway 3. *The number of LLM calls and their latencies vary across different applications and within the same application. Inefficient scheduling can cause these calls to get queued up, leading to delays in the execution of these LLM calls and hence results in high latencies. This observation indicates that scheduling must account for LLM call patterns to avoid delaying users.*

5.2.4 Impact of RPM Throttling

In industries, the de-facto standard of throttling to curb abusive behavior has been to use rate-limiting solutions, i.e., if the number of requests belonging to a user goes above a certain limit, then the request gets throttled [154, 155]. While simple, this basic throttling approach has several issues. First, as shown in Table 5.2, around 26.8% of all requests have a graph size larger than one (i.e., each interaction consists of multiple LLM calls), meaning that if the user goes over the limit within the interaction, the request will get throttled and the user will not get a response. Figure 5.5(c) shows that throttling unawareness across applications can waste over 30K tokens, diverting resources that could have otherwise



(a) Distribution of throttled requests across different applications.

(b) Ratio of requests throttled at application and user level for each application.

(c) Distribution of wasted tokens due to throttling across applications.

Figure 5.5: A throttling system without user and application-awareness can lead to resource wastage.

Table 5.2: Distribution of graph sizes and tokens in graphs

Graph size	Requests	Avg Input	Avg Output	Avg Sum
1	73.22%	4994.65	136.86	5131.52
2-10	26.09%	21352.13	241.19	21593.33
11-20	0.50%	51200.13	3258.75	54458.90
21-30	0.11%	65924.80	9217.13	75141.94
31-40	<0.01%	94096.51	13582.78	107679.30
41-50	<0.01%	161158.58	12874.49	174033.07
>50	<0.01%	543559.74	22102.99	565662.74

reduced latency, improved throughput, and enhanced user satisfaction. Second, the needs of users may vary if they are receiving service from different applications. Hence, a fixed limit applied to all users might prevent certain users from receiving service if the number of LLM calls made by those users exceeds the predefined user-level request limit.

Another variation of rate-limiting solution is to place limits on different applications [10], i.e., if the requests belonging to an application goes over a limit, all further user requests belonging to that application will get throttled. However, such a rate limiting approach will also have unintended consequences. First, a request limit for a specific application can overflow, causing unnecessary throttling of user requests despite unfilled quotas or low system load, resulting in inefficient resource utilization and user dissatisfaction. For example, Figure 5.5(a) shows that three apps account for around 99.93% of all throttled requests, implying that their request limits are exceeded far more frequently than those

of the other apps. Second, without enforcing user-specific limits, setting application-only limits allows malicious users to gain unfair access to the compute resources by flooding the system with requests across multiple applications.

The nature of throttling can also vary across different applications. For example, Figure 5.5(b) reveals that a large number of requests are frequently throttled at the application level (8, 9, and 10), at the user level (1, 2, 4, 5, and 6), and a combination of both (3 and 7) although not all users from this throttled set was abusive. This result indicates that rate-limiting solutions are vulnerable to abuse.

Takeaway 4. *With the RPM policy, a certain number of users and applications tend to be throttled. Throttling in the middle of execution leads to the wastage of resources and users' resource quota. This observation suggests the necessity of a user- and application-aware throttling policy to prevent abuse and minimize resource waste by throttling only during overloads and outside active LLM interactions.*

5.3 FairServe Design

Our study in § 5.2 sheds light on the potential to address the problem of throttling in the middle of interactions and token wastage, and motivates the design of an application and user request characteristic-aware system for ensuring fairness. The main argument in VTC is to serve clients as per the principle of fair queueing [152], where each client is guaranteed at least an equal amount of resources, i.e., $1/n$ of the server's resources. FairServe argues for weighted fair queueing [153], where resources are allocated based on pre-determined weights, allowing for an *equitable* allocation tailored to the specific needs or behavior of each user or application. *This approach recognizes that fairness is not always achieved through strict equality, as proposed by VTC [6]; instead, equitable allocation can better address heterogeneous demands of users and applications.*

Designing a weighted fair LLM serving system presents non-trivial challenges.

First, in multi-tenant LLM service providers, resource wastage primarily arises from two areas: (1) allowing abusive users a portion of valuable resources, and (2) mid-interaction throttling even if system is underutilized. Increasing resource utilization while ensuring quality service for benign users and detecting abuse thus becomes challenging, requiring fine-grained tracking within LLM interactions.

Second, users in multi-tenant environments belong to diverse applications, each with unique characteristics. Weighted fair allocation requires understanding each application's behaviors at different stages of an LLM interaction.

This section presents the design principles of FairServe to address these challenges, followed by the design detail. Figure 5.6 shows the overview of FairServe and Figure 5.7 illustrates how FairServe ensures equitable fairness across users.

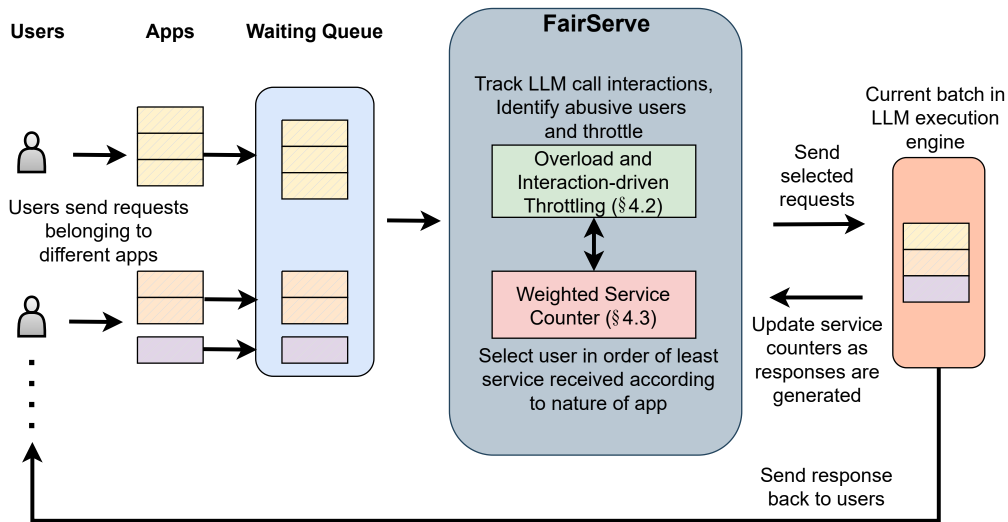


Figure 5.6: The design overview of FairServe.

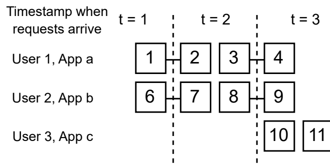
5.3.1 FairServe Overview

FairServe comprises of two core components: (1) Overload and Interaction-driven throttling (OIT), and (2) Weighted Service Counter (WSC) scheduling (see Figure 5.6). In contrast to traditional RPM-based throttling, OIT throttles user requests only when the KV cache is overloaded, thus making maximum utilization of available resources. FairServe uses a combination of user- and application-level limits to perform throttling at the LLM interaction level instead of at the request level, reducing token wastage by accounting for user and application behaviors. WSC crafts a user service weighing mechanism determined by the ratio of tokens processed to the expected token count (e.g., the average based on historical statistics) for the application associated with each user's requests at each level of the LLM interaction. WSC selects requests from the users who have received the least service, defined by a weighted resource slice. This weight is calculated based on the token ratio to ensure fairness across users.

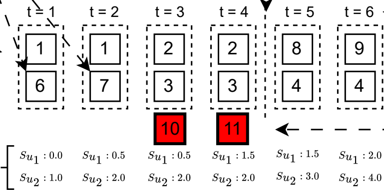
Algorithm 47 shows the details regarding how the two components interact with one another while processing requests. The entire system is integrated with the state-of-the-art continuous batching mechanism and operates in two parallel streams— the monitoring and the execution stream. The monitoring stream continuously listens for incoming requests and the execution stream is in charge of the prefill and the decode phases. OIT is a part of the monitoring stream (Alg. 47, lines 19-25) and the larger chunk of weight and service allocation mechanisms in WSC take place in the execution stream (Alg. 47, lines 27-48).

User+App	Input len	System prompt len	Output len	Avg app input len	Avg app sys. len	Avg app output len	Request token ratio
User 1, App a	100	100	100	200	200	200	0.5
User 2, App b	50	50	50	50	50	50	1
User 3, App c	50	50	50	25	25	25	2

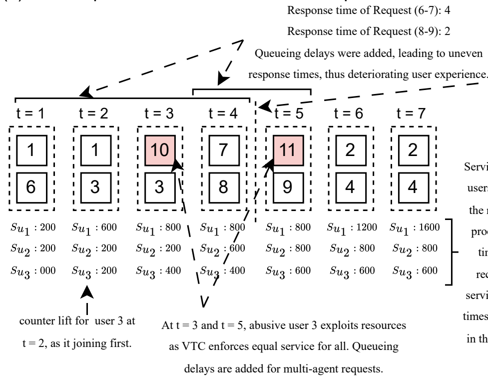
For VTC, the weighted service of each request for users 1, 2, and 3 is 400, 200, and 200, respectively, while for FairServe, it is 0.5, 1, and 2. Users 1 and 2 have multi-agent requests (requests connected with line), with user 1 receiving responses 2× later due to 2× longer prompts. User 1 has a higher need for more resources as its prompts are larger. User 3 is abusive, sending two requests at $t = 3$ despite app c's limit of one.



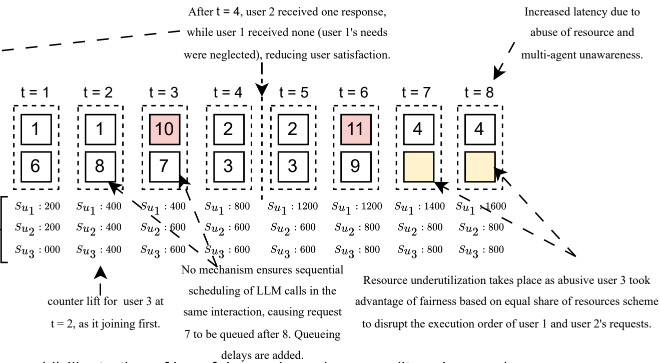
(a) User request characteristics and arrival patterns.



(b) Illustration of how FairServe processes user requests while maintaining equitable fairness.



(c) Illustration of how fairness based on equality schemes (e.g., VTC) processes requests in one scenario where request 3 is queued for processing at t = 2.



(d) Illustration of how fairness based on equality schemes (e.g., VTC) processes requests in one scenario where request 8 is queued for processing at t = 2.

Figure 5.7: (a) User request characteristics and their arrival patterns for the illustrations in (b)-(d). (b) Illustration of how FairServe ensures equitable fairness, reduces latency and queueing delays while curbing abusive behavior during processing user requests. (c) & (d) Illustration of two scenarios on how equality-based fairness schemes (e.g., VTC [6]) can lead to abuse, resource wastage, longer latencies, and degraded user experience.

5.3.2 Overload & Interaction-driven Throttling (OIT)

OIT performs two main functions: (1) tracking requests from users and applications against specified limits at both the user and application level; and (2) tracking the condition of the KV cache and throttling based on the specified limit for users and apps only when the KV cache is overloaded.

To track user and application requests, OIT maintains a dictionary that maps each user

Algorithm 5: FairServe System

```
1  B: Current batch
2  u_i ← 0 for all user i
3  Q: Waiting queue
4  M_new: New minibatch
5  T_g: Global request limit
6  T_a: App request limit
7  c_{g,u}: User request count
8  c_a: App request count
9  ▷ Monitoring Stream:
10 while True do
11     if new request r from user u and app a then
12         if not ∃r' ∈ Q, user(r') = u then
13             if Q = ∅ then
14                 let e ← most recent user to exit Q
15                 u_u ← max{u_u, u_e}
16             else
17                 X ← {i | ∃r' ∈ Q, user(r') = i}
18                 u_u ← max{u_u, min{u_i | i ∈ X}}
19         c_{g,u} ← c_{g,u} + 1; c_a ← c_a + 1
20         if system overloaded & r not in interaction then
21             if c_{g,u} > T_g then
22                 block r; c_{g,u} ← c_{g,u} − 1; break
23             else if c_a > T_a then
24                 block r; c_a ← c_a − 1; break
25         Q ← Q + r
26 ▷ Execution Stream:
27 while True do
28     if can_add_new_request() then
29         M_new ← ∅
30         while True do
31             Y = find_incomplete_interactions(B)
32             if any interaction is incomplete in B then
33                 k ← arg min_{i∈{user(r)|r∈Y}} u_i
34                 x ← incomplete interaction from k in B
35                 r ← next request in x
36             else
37                 let k ← arg min_{i∈{user(r)|r∈Q}} u_i
38                 let r be the earliest request in Q from k
39             M_new ← M_new + r; Q ← Q − r
40         forward_prefill(M_new)
41         B ← B + M_new
42     forward_decode(B)
43     F ← find_finished_requests(B)
44     for each f in F do
45         i ← user(f) ; a ← application of request f ; j ← current stage of interaction of f
46         w_aj ← weight of m in stage j
47         u_i ← u_i + E_i((αL_I^{ai} + βL_S^{ai} + γL_O^{ai}) / w_aj)
```

and application to the arrival times of their respective requests. OIT takes a combined rate-limiting approach that merges both user and application limits, based on the analysis of historical data. A combined user and application-aware approach helps address the challenge of resource wastage caused by abusive behaviors. When a new request arrives, its arrival time is appended to the list associated with the corresponding user and app. Tracking arrival times enables OIT to detect if any user or application exceeds their request limit. Upon request arrival, the internal request counter for the corresponding user and application is incremented (Alg. 47, line 19).

OIT continuously monitors the state of the KV cache and assesses whether new requests can be added to the waiting queue. When the KV cache becomes overloaded, OIT checks whether a request is in the middle of an interaction. If so, this request is not throttled and instead stalled to (1) guarantee resource utilization and (2) minimize unnecessary token wastage. Otherwise, at first, the user's RPM, and later the application's RPM limit, is checked to see if the user or app has exceeded the limit before the incoming request is served. If either limit is exceeded, the next request is throttled (Alg. 47, lines 21-24). The requests in the waiting queue are added to a new, pending batch of requests and then merged into the active batch currently being served (Alg. 47, line 25).

5.3.3 Weighted Service Counter (WSC)

WSC performs two main functions: (1) calculating the service received by users based on the ratios of input, system, and output tokens processed for their requests to the maximum token limits of the corresponding apps; and (2) identifying the user with the least service received and forwarding the user's request for processing.

To align user service with the characteristics of their corresponding applications, WSC updates the service received by users based on the ratio of processed tokens to the normal range (average) of request token length associated with the app. Before a user receives a response, the request goes through multiple stages of an interaction depending on the application. We model the weight of app 'a' in stage 'j' of interaction in Eq. 5.1.

$$w_{aj} = \alpha \bar{N}_I^{aj} + \beta \bar{N}_S^{aj} + \gamma \bar{N}_O^{aj} \tag{5.1}$$

Each of the different types of tokens also has a weight associated with it. Since processing input tokens can be parallelized and hence is much fast compared to processing the output tokens, weights enable WSC to ensure that service is counted according to the number of different categories of tokens that are getting processed. Following OpenAI conventions [156], we fix α, β, and γ to be the weights for input, system and output tokens respectively. $\bar{N}_I^{aj}$, $\bar{N}_S^{aj}$, and $\bar{N}_O^{aj}$ represent the expectation of the number of input, system, and output tokens respectively for the app a at stage j of an interaction. We calculate these expectation values based on analysis on historical data.

The input, system prompt, and output lengths for each user i corresponding to app a are denoted by L_I^{ai}, L_O^{ai}, and L_S^{ai} respectively. The total service received by a user at the end of an interaction is computed using Eq. 5.2 where app a requires m_i LLM calls to complete an interaction and E is a user priority factor that can be adjusted to give preference to certain users based on system policies or usage trends.

$$S_i^a = E_i * (\sum_{j=1}^{m_i} \frac{\alpha L_I^{aij} + \beta L_S^{aij} + \gamma L_O^{aij}}{w_{aj}}) \tag{5.2}$$

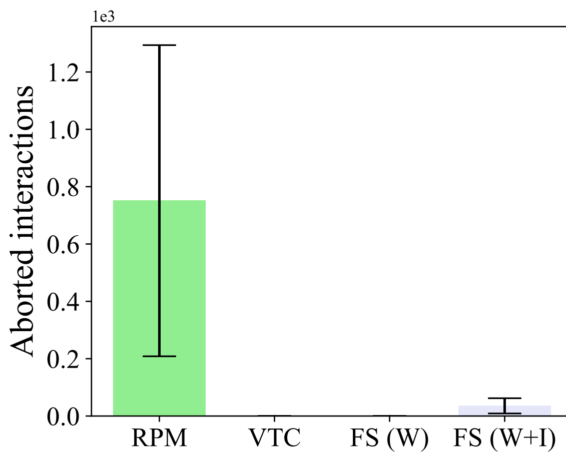
WSC maintains a service counter u_i for all users which is initialized to 0. Waiting queue, Q immediately adds a new incoming request upon its arrival after checking for abusive behaviour using overload-driven throttling § 5.3.2 (Alg. 47, lines 19-24). Q is a dict that maps users to their corresponding incoming requests. If this is the only request from the sender user, then a counter adjustment takes place (Alg. 47, lines 12-18) similar to prior research [6]. This sort of adjustment is done to create balance, ensuring that underloaded periods of certain users do not create unfairness against other active users.

In the execution stream, WSC evaluates whether new minibatches, M_{new}, composed of user requests, can be merged with the current batch, B. Initially, it identifies which requests within the current batch are part of an ongoing interaction, requiring additional processing to complete the interaction. Among those requests, the system identifies the user who has received the least service so far and selects the next request from that user's interaction (Alg. 47, lines 31-35). This technique enables FairServe to keep lower queueing lengths and ensure better response times.

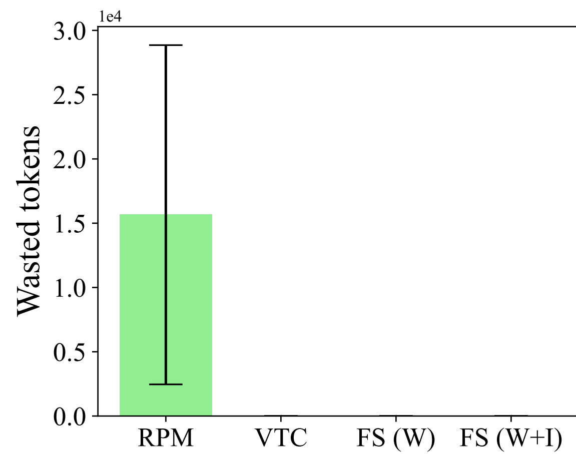
Otherwise if no requests in the current batch are part of an ongoing interaction, the earliest request from the user who has received the least service, as determined by Eq. 5.2, is selected for inclusion in M_{new} (Alg. 47, lines 36-39). Once the prefill and decode stages for a request are completed, the service counters for the users are updated, proportional to the app's assigned weights (Alg. 47, lines 40-48).

5.4 Evaluation

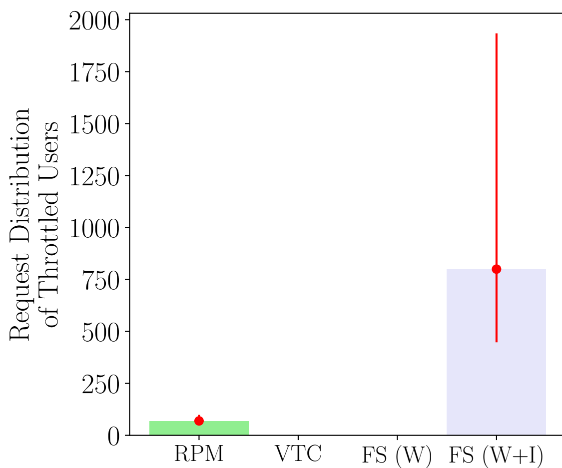
In this section we evaluate the effectiveness of FairServe against other baselines on the real-world workload trace that we analyzed in § 5.2. We deployed a Llama-7B model on an A100 GPU (similar to VTC) that was used to serve user requests of the real trace. We run all of the policies for different durations of time and observe how each perform on curbing abusive behaviour and achieving fairness. FS (W+I) or FS indicates FairServe with all components and FS (W) indicates FairServe is using only the WSC component. We denote all of the applications in our experimentation with Ids to protect anonymity.



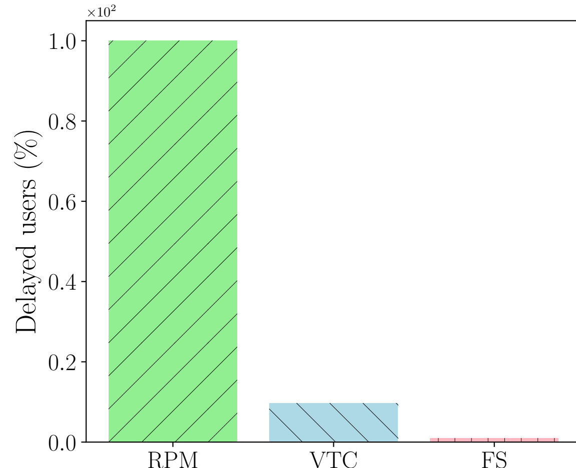
(a) Aborted interactions during LLM serving.



(b) Tokens wasted from interaction throttling.



(c) Request distribution within throttled users.



(d) Delayed users due to longer waiting queue lengths.

Figure 5.8: FairServe does not incur resource wastage as it is multi-agent LLM interaction-aware.

Our baselines are two state-of-the-art solutions used widely in industry and academia for curbing abusive behavior and achieving fairness—(1) RPM (Requests per minute), i.e., a rate-limiting solution; and (2) VTC [6]. Our evaluation aims to answer the following questions:

- Can FairServe ensure equity? (§ 5.4.1)
- To what extent can FairServe curb abuse and token wastage? (§ 5.4.2)
- How effectively does FairServe reduce queuing delays for users in multi-agent LLM applications? (§ 5.4.3)

- What is the impact of FairServe on the prompt and decode throughput and time-to-first-token (TTFT) latency? (§ 5.4.4)
- Can FairServe improve the user serving experience across requests and interactions? (§ 5.4.5)

Table 5.3: Synthetic benchmark setup

User & App	**Input Length**	**System Prompt Length**	**Output Length**	**App Average Lengths**			**Weighted Service**	
				Input Length	**System Prompt Length**	**Output Length**	**Tokens**	**Token ratio**
User 1, App A	100	100	100	200	200	200	400	0.5
User 2, App B	50	50	50	50	50	50	200	1

Table 5.4: Number of responses received and TPOT latency.

Policy	**Responses**		**TPOT Latency (s)**	
	User 1	**User 2**	**User 1**	**User 2**
FairServe	720	362	0.035	0.075
VTC	644	456	0.105	0.043

5.4.1 Demonstration of Fairness Objective: Equity

We demonstrate FS's equitable fairness in multi-agent scenarios through a synthetic trace experiment while comparing against VTC. Table 5.3 presents the input, system prompt, and output lengths for requests from two users across different applications, along with their respective average values. The weighted service for each request is measured in two formats: token count and token ratio. The token-count-based weighted service is computed as the sum of each token type's length multiplied by its corresponding weight, while the token ratio follows Eq. 5.2.

Setup. User 1 uses App A with 4 LLM calls/request and double the token weight of User 2's App B (1 call/request). User 1's RPS is 2× that of User 2. User 1 has a greater need than User 2 because it's prompts are longer and requests are more frequent.

Response Rate. Table 5.4 shows that with FairServe, User 1 receives 1.99× more LLM responses than User 2, aligning with the higher demand of App A. In contrast, this ratio is just 1.41× using VTC.

Avg. time per output token (TPOT). With FairServe, TPOT of User 2 is 2.14× of User 1, while for VTC, User 2's TPOT is lower than User 1 at 0.4×. FairServe processes each token

of user 1 faster than user 2 as user 1 has a higher need. FairServe also had a better TPOT than VTC by 3× and completed 12% more multi-agent requests.

Thus, FairServe ensures greater equity than VTC by distributing service based on application needs (e.g., User 1 with fewer requests but more LLM calls), while improving TPOT for all users.

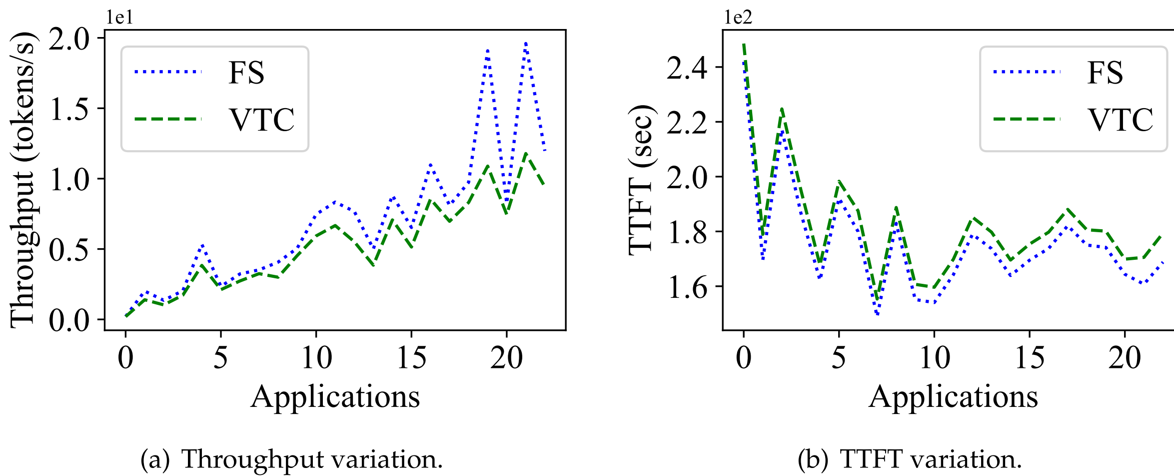
5.4.2 Curbing Abusive Behaviour

As discussed in § 3.2, in multi-agent LLM apps, user queries initiate interactions having multiple LLM calls. Throttling in the middle of interaction wastes resources on tokens that provide no benefit, as the user receives no final response. Figure 5.8(a) shows that as RPM based strategy is completely oblivious of interactions, it throttles users blindly based on a specified limit, and hence throttles $21.15\times$ more interactions compared to interaction-aware FS (W+I). FS (W+I) throttles only those interactions that are not in the middle of execution. Figure 5.8(b) shows that RPM's unwareness regarding multi-agent apps wastes a lot of tokens (mean: $15.66 * 10^3$). On the other hand, FS (W+I), with multi-agent awareness and knowledge about the KV cache, only throttles during system overloads when users are not in mid-interaction, thus not incurring any token wastage while maintaining maximum KV cache utilization.

Moreover, FS (W+I) manages to throttle only users who fall outside the normal range, i.e., abusive. Figure 5.8(c) shows that while FS (W+I) throttles users whose request distribution is well above the normal range, RPM policy throttles users with very low request distribution. VTC and FS (W) does not have any throttling policy and hence serves all users without throttling. However, our evaluation shows that VTC wastes approximately 77.2×10^3 tokens by serving both abusers and benign users equally.

5.4.3 Reducing Queueing Delays

One of the aspects that adversely affects user experience is if requests are waiting for longer durations before processing. Since, FairServe is multi-agent aware, it prioritizes serving users who are already in the middle of an interaction using the WSC component (§ 5.3.3). Thus users do not wait a long time before receiving the final response. Figure 5.8(d) shows that only a mere 0.93% users experience queueing delays in FairServe, making it $10.67\times$ and $93\times$ less compared to VTC and RPM respectively.



(a) Throughput variation.

(b) TTFT variation.

Figure 5.9: FairServe (FS) maintains a higher throughput and a lower TTFT (time-to-first-token) latency across apps.

5.4.4 Improvement in Throughput and Latency

Since FairServe manages to curb abusive behavior, we now look at how such curbing impacts the throughput of the user requests across the different applications.

Figure 5.9(a) shows that FairServe maintains $1.03\times$-$1.75\times$ higher throughput across all of the applications. Moreover, Figure 5.9(b) shows that FairServe manages to attain $1.03\times$-$1.06\times$ lower TTFT latency, i.e., the time required to generate the first token in a request across all of the apps. FairServe obtains this improvement obtained in throughput and TTFT latency as less users experience queueing delays due to WSC component and abusive users are not given opportunity to waste valuable resources and time due to the OIT component.

We further dig deeper into the throughput of user requests to understand the impact of FairServe's policies at individual stages of the LLM inference process. Table 5.5 shows that FS (W+I) maintains a $1.05\times$ and $1.1\times$ higher throughput compared to VTC and RPM respectively in the prompt processing phase. In the decode phase, FS (W+I) maintains a $1.02\times$ and $1.47\times$ higher throughput compared to VTC and RPM. Although TTFT is the same for FairServe and VTC, we observe that it is lesser for RPM. As RPM throttles lots of requests blindly, the requests that do get processed enjoys a slightly better TTFT.

5.4.5 Improvement in Served Experience.

In a real-time LLM serving system, requests are processed continuously. When incoming requests exceed the server's capacity, an RPM policy may throttle users or requests but

Table 5.5: Throughput and latency during real-time processing.

Scheduler	Throughput (TPS)		TTFT	Service provided %		Served users %	
	Prompt	Decode		Requests	Interactions	Requests	Interactions
RPM	11155.15	182.85	40.20	51.52	85.36	92.03	87.64
VTC	12190.10	262.97	42.57	-	-	-	-
FS (W)	12217.11	267.05	42.82	-	-	-	-
FS (W+I)	12247.98	267.99	42.53	96.67	100	99.45	100

still handles a high volume due to the abundance of incoming requests. Here, we analyze the proportion of users receiving a final response among all users who received feedback. We define *served users* as those who received a final response and *service provided* as the requests or interactions that led to a final response without being throttled.

Table 5.5 shows that FairServe consistently has a higher percentage of served users and provided service compared to RPM. In particular, for multi-agent apps involving interactions, it manages to provide 100% service and served user indicating that FairServe could improve the user experience for all users in multi-agent scenarios. Note that, we do not show this comparison against VTC as it does not throttle, and hence all users receive feedback even if it is at the expense of increasing the latency of another user.

5.4.6 A Case Study: Comparing Diverse Apps

Table 5.6 shows three diverse applications sending requests with a timeout. Application 14 has request with moderate prompt size and heavy decode size, application 7 has prompt heavy requests with moderate sized decodes while application 12 has both, light prompts and decodes. We see that a naive RPM strategy with FCFS scheduling disproportionately benefit applications 14 and 7 due to their heavy nature, as observed in their prompt and decode throughputs. FS (W) and VTC are able to balance the service each of these applications is getting, improving the prompt throughput for application 12 by $24\times$. However, they still waste a lot of tokens while scheduling as they ignore multi-agent requests in these applications. Finally, we observed that while VTC is wasting 3224 tokens by ignoring multi-agent calls, FS (W+I) is able to provide service of comparable levels to each applications, while wasting almost $8\times$ (461) less tokens.

Table 5.6: Case study: Comparison of service provided to specific applications by each strategy

-		Application 14	Application 7	Application 12
Requests		269	139	448
Avg. Prompt		6370	14854	999
Avg. Decode		102	74	32
Requests Served	RPM	56	4	2
	VTC	37	29	73
	FS (W)	37	29	73
	FS (W+I)	37	27	80
Prompt Throughput	RPM	360.99	859.70	5.19
	VTC	256.14	374.31	124.19
	FS (W)	253.41	390.47	124.69
	FS (W+I)	258.20	359.02	127.58

5.5 Related Work

Fairness & Characteristic-awareness in ML Training. Fair scheduling mechanisms like Quincy [157], DRF [158], and CARBYNE [159] have been proposed for long-running jobs in clusters. However, ML jobs differ from traditional workloads in their nature, with specific requirements for scheduling and placement. As a result traditional policies fail to ensure fairness [160].

To ensure fairness in ML job scheduling, Gandiva [161] introduces a profiling and resource-trading approach, Themis [160] employs an auction-based method to balance fairness and efficiency, Gavel [162] combines optimization techniques with preemptive, round-based scheduling to fairly allocate resources among users, and Pollux [163] dynamically adjusts resources to optimize both cluster-wide performance and fairness. Sia [164] leverages integer linear programming to ensure fairness while enabling elastic scaling for hybrid parallel ML jobs. Additionally, SHADE [**?**] exploits sample characteristics and FedCaSe [2] exploits client and sample characteristics to speed up ML training.

While these works address scheduling, fairness, and efficiency in ML training employing a wide variety of ML characteristic-aware techniques, they are not designed to handle the specific requirements of LLM serving jobs in multi-tenant environments. In such settings, LLM serving requires addressing distinct issues, such as prioritizing users and applications during request batching and throttling to mitigate abusive behavior.

Fairness & Characteristic-awareness in LLM Serving. Recently, there has been growing momentum in both industry and academia focused on enhancing LLM serving systems. Techniques like batching [165], memory optimizations [166], scheduling [167], exploitation of other LLM-specific characteristics like chunking prefills [168], disaggregation of execution stages [169, 170] or a combination of these techniques [59, 15, 171] have been proposed to improve the throughput and latency of serving models across users.

Although these systems aim to enhance LLM inference performance for users, the critical issue of ensuring fairness in serving LLMs has remained largely unexplored. VTC [6] introduces a scheduling policy designed to ensure that users receive service in a fair and equal manner. However, in multi-tenant environments, where users request services from diverse applications, enforcing equal service across users is suboptimal. Different applications have varying resource demands that must be met for optimal performance. FairServe bridges this gap with an application-aware fair scheduling policy that balances fairness with each application's specific needs.

5.6 Summary

While there have been significant strides in lowering latency and enhancing throughput for LLMs, ensuring fairness across diverse LLM application users is a newer challenge that has garnered significant attention. While several fairness methods have been suggested, they fall short when applied to LLM users in diverse applications. In this work, we conduct a large-scale analysis of LLM user and application characteristics at a leading LLM service provider. Using our gathered insights, we propose FairServe—a user and LLM application characteristic-aware LLM serving system that ensures fairness in scheduling user requests in a multi-tenant setting.

Chapter 6

Conclusion

Artificial intelligence (AI) systems are undeniably transforming industries, from precision medicine and environmental sustainability to finance, education, and industrial automation. These technologies have unlocked unprecedented opportunities to address previously insurmountable challenges and drive innovation. However, as AI adoption accelerates, critical challenges in scalability, efficiency, and resource optimization have emerged. The growing complexity of AI workloads places immense strain on distributed systems, exacerbating issues such as GPU underutilization, I/O bottlenecks, and inequitable resource allocation. These inefficiencies not only hinder AI's scalability but also risk deepening resource disparities, limiting the transformative potential of AI across diverse domains. At the heart of these challenges lies the need to align AI workloads with the capabilities and constraints of heterogeneous systems, from high-performance GPU clusters to resource-constrained edge devices.

A key goal of this dissertation is to transform how AI workloads are managed, ensuring they are effcient, scalable, and equitable. Hence, this dissertation focuses on designing adaptive, resource-aware AI systems that bridge the gap between application needs and system constraints. In this dissertation, we have effectively applied the principle of workload-awareness to identify key challenges in distributed machine learning and data-intensive systems. Our investigations have resulted in novel, impactful solutions that address these challenges with precision and effectiveness. For example, to the best of our knowledge, SHADE [1] is the first to develop an intelligent caching system for deep learning applications that outperforms even the optimal Belady's MIN offline caching [115] algorithm in the context of deep learning. Moreover, FedCaSe [2] is the first to realize a unified intelligent client scheduling, data sampling, and caching solution for millions of client devices having heterogeneous limited memory sizes. Additionally, FairServe [3] is the first to propose multi-agent-aware equitable approach to scheduling user requests in LLM inference.

This dissertation tackles the increasing complexities of modern computing and data-

intensive systems, highlighting the critical need for efficient and adaptable resource management. It focuses on three key workload domains—deep learning, federated learning, and LLM inference—across three real-world deployment scenarios: (1) high-performance computing clusters, (2) edge computing in the cloud, and (3) multi-tenant cloud platforms. Through extensive analysis of workload behaviors, the development of rigorous models, and the design of efficient systems [1, 2, 3], this work enhances system-level efficiency while prioritizing practical, user-centric metrics.

6.1 Summary & Impact

Deep learning (DL) training is inherently both compute- and data-intensive, with resource demands varying significantly across different phases. As large models are trained on GBs to TBs of data stored in SSDs or HDDs, they rely on high-performance GPUs to perform computations. However, as GPUs continue to grow more powerful, their performance increasingly depends on the effciency of data delivery from storage systems. In many cases, GPUs are left idle, waiting for data to arrive, resulting in significant underutilization. This imbalance, caused by I/O systems unable to keep up with GPU compute power, has emerged as a critical bottleneck in DL training pipelines. Addressing this challenge—effciently bridging the gap between data access and computational needs— is paramount for unlocking the full potential of modern DL systems.

First, to address this challenge, we target deep learning workloads that are seen in high-performance computing clusters. Data retrieval from storage devices is a bottleneck in data-parallel distributed DL training on high-performance GPU clusters. In this regard, SHADE [1] is an innovative caching system that significantly boosts training effciency while addressing this I/O bottleneck from external storage devices. SHADE introduces a novel rank-based importance calculation framework to dynamically evaluate the relative importance of data samples during every stage of training. Leveraging this insight, SHADE employs a priority-based sampling policy and an importance-aware caching technique to exploit data locality, ensuring that high-impact samples are prioritized for faster access. This carefully engineered combination of sampling and caching has led to transformative improvements: Cache hit ratio enhanced by up to 4.5×, training throughput increased by up to 2.7×, and accuracy improvement rate increased by 3.3× compared to traditional caching methods allowing better GPU utilization and reduced idle time. SHADE's contributions extend beyond technical innovation, influencing both academia and industry. Its methodologies have been integrated into project-based coursework, including Advanced Topics in Systems and Big Data Systems at Virginia Tech and the University of Virginia, providing students with hands-on experience in addressing real-world DL system challenges. In the industrial domain, SHADE's practical relevance has been recognized in technical talks at leading organizations, such as IBM Research and ByteDance. These engagements have fostered collaborations across diverse academic and

industrial communities, demonstrating SHADE's versatility and broad applicability. By bridging theoretical advancements with practical deployments, SHADE exemplifies how research can simultaneously advance the SOTA and address pressing real-world needs.

Second, we look at this challenge from the perspective edge devices in the cloud (i.e., Federated Learning). As with high performance clusters, modern edge devices suffer from an I/O bottleneck where storage systems struggle to match the computational speed of advanced processors. In Federated Learning (FL) systems, this issue is exacerbated by the inherent heterogeneity of edge devices, including variability in compute power, storage capacity, and network connectivity. Without intelligent policies for effcient on-device data placement and client scheduling, FL training suffers from inefficiencies, leading to longer training times, slower accuracy convergence, and reduced scalability. Motivated by these challenges, we proposed FedCaSe [2], a novel framework for Federated Learning that unifies client scheduling, data sampling, and caching to optimize performance for massively heterogeneous edge devices in the cloud. FedCaSe introduces a key innovation: leveraging the experience of both data samples and clients— their relative impact on overall performance metrics—to guide caching and scheduling decisions. By exploiting this experience, FedCaSe dynamically optimizes the placement of high-impact data within the limited memory of edge devices and orchestrates the selection of clients to maximize training effciency. Our experiments with representative workloads and policies show that compared to the state-of-the-art, FedCaSe improves the experienced client participation up to 29.1×, improves the global read hit ratio (RHR) across all clients by up to 81.7× (locally up to 318.58×), and thus ensures accuracy improvement rate up to 2.06× faster based on wall clock time, up to 1.4× faster based on number of rounds while keeping the round duration up to 2.4× less. These results demonstrate FedCaSe's ability to overcome the I/O bottleneck and unlock the full potential of FL systems in heterogeneous devices in the cloud. By addressing a critical pain point in FL deployments, FedCaSe paves the way for broader adoption of edge-based AI, enabling a wide range of applications, from healthcare diagnostics to personalized learning systems. FedCaSe was featured in presentations and poster sessions at ACM SoCC and highlighted on the blog of Chameleon Cloud, the NSF's premier cloud services provider.

Finally, we shift our focus from training to inference in ML and look toward addressing the challenges stemming from multi-agent AI systems during inference in large language models. The exponential rise of multi-tenant platforms serving personalized large language models (LLMs) has transformed the way users interact with applications such as interactive question answering (QA), summarization, and coding assistance. However, this surge in demand has introduced critical challenges: abusive user behavior can overwhelm systems, leading to unfair service allocation, resource wastage, and degraded user experiences. Existing solutions, such as request-per-minute (RPM) limits, fall short of addressing these issues holistically. These rate-limiting strategies often throttle legitimate requests during low-load periods, causing resource underutilization, incomplete responses in multi-agent LLM workflows, and ineffciencies that limit platform scalability. Motivated

by the need for a fair, effcient, and user-centric approach to LLM serving, we conducted an extensive analysis of millions of LLM requests across 34 applications on Microsoft's CoPilot platform. This study provided unprecedented insights into the unique characteristics of modern LLM applications and shaped the foundation for FairServe. Notably, this dissertation introduces Overload and Interaction-driven Throttling (OIT), a novel method that curtails abusive behavior while maximizing resource utilization by considering application-specific interaction patterns. Complementing this, this dissertation builds the Weighted Service Counter (WSC) mechanism, which ensures fair scheduling of LLM requests by dynamically balancing diverse application needs. These components culminated in FairServe, a groundbreaking LLM serving system designed to achieve fairness, effciency, and abuse prevention. FairServe was rigorously evaluated on a testbed of over a thousand users, demonstrating remarkable improvements: queue delays reduced by 10.67–93×, latency (time-to-first-token) decreased by 1.03–1.06×, throughput increased by 1.03–1.75× (achieving 0% token wastage), near-perfect abuse mitigation, ensuring 99.45–100% user satisfaction across applications. FairServe is expected to benefit millions of MS CoPilot users world-wide.

6.2 Future Directions

Building on my work with systems like SHADE, FedCaSe, and FairServe, which tackle critical challenges in fairness, scheduling, and I/O efficiency in distributed ML systems, my research agenda aims to extend these foundational insights into the rapidly evolving domain of Generative AI (GenAI). This next phase of research focuses on improving the efficiency, scalability, and accessibility of GenAI services by addressing core challenges across caching, inference cost reduction, and SLA-aware performance optimization. My research agenda stems from the fact that there is a fundamental mismatch between GenAI application requirements, usage patterns, and the underlying system software stack. This mismatch leads to sub-optimal performance and effciency. By reducing the operational costs of GenAI systems, I aim to lower their carbon footprint, democratizing access to these technologies for a wider audience. I aim to address this via a careful and holistic cross-stack system design, which explores fine-grained data/model caching, inference scheduling, and selective execution opportunities to sustain modern LLM agent-based applications efficiently. Below, I outline my vision and key research thrusts.

6.2.1 Client-Side Optimizations for GenAI Systems: Context Aware Caching

Query redundancy in LLM-based applications stems from the inability to leverage syntactic matches due to semantic nuances and contextual differences in user inputs. Ensuring

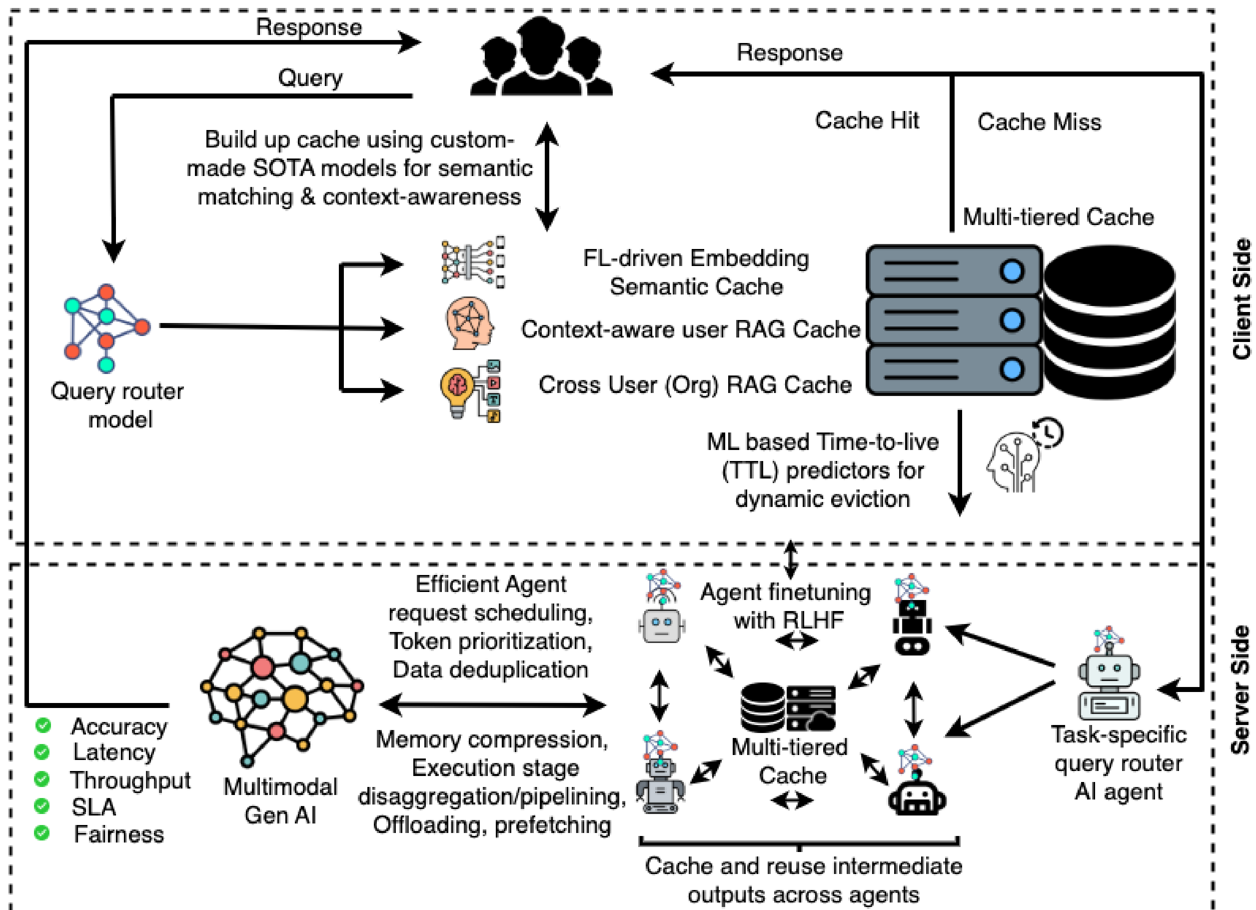


Figure 6.1: Future Directions Overview

accurate response retrieval from cache requires solutions that account for these complexities. Moreover, determining optimal caching strategies—spanning storage location (e.g., device vs. server), data format (e.g., embeddings or raw text), and multi-tier architecture (user, group, and organization levels)—remains a significant challenge. To address syntactic limitations, I propose embedding-based semantic matching for cached queries, using a federated-trained model. This method ensures contextual relevance while maintaining privacy and efficiency. By introducing the concept of context chains, I will track semantic shifts across user interactions, enabling accurate cache hits even in evolving query contexts. These chains will be integrated into a graph-based cache structure to capture interdependent queries efficiently. I propose a hierarchical caching system that categorizes and stores responses at multiple tiers—user, group, and organization—based on demographic and usage patterns. This approach enhances cache scalability and reduces inference costs across diverse user bases. My goal would be to customize cache population, eviction, and storage formats for LLM applications to maximize cache hits while minimizing latency and resource overhead.

6.2.2 Server-Side Optimizations For GenAI Systems: Model Caching and Routing.

This thrust addresses server-side inference cost reduction by leveraging white-box access and query similarity across users. Key challenges include efficiently caching intermediate inference states for advanced models like Multi-Modal Diffusion and managing the complexity of routing queries to the appropriate model (e.g., LoRA) among thousands hosted on servers. Although techniques such as reusing intermediate noise states have been explored for specific data types [172, 173], adapting these to efficiently handle multi-modal distributed data in caching systems remains unexplored. To overcome these, I propose caching intermediate outputs and contexts for re-use in similar multi-modal queries, alongside a fine-grained Retrieval-Augmented Generation (RAG)-based mechanism for optimized query routing and scheduling. This approach ensures cost savings while maintaining routing efficiency. Furthermore, the caching framework can support LoRA fine-tuning by reusing intermediate states, reducing training overhead and enabling scalable, cost-effective server-side operations.

6.2.3 Cross-Stack Optimization for Resource Management

Inspired by the resource-aware scheduling in FairServe, I aim to design holistic, SLA-aware inference frameworks that balance latency, accuracy, and cost efficiency. My goal is addressing the complexities of meeting strict Service Level Agreements (SLAs) in deploying cost-efficient inference systems. Variations in query embeddings, intermediate states, and fine-tuned large models often propagate uncertainty, challenging latency, accuracy, and performance guarantees in enterprise settings. I propose a framework to ensure SLA compliance through token reduction and embedding compression techniques for multimodal models. This includes early pruning of irrelevant query segments and context-aware compression to minimize inference loads and improve retrieval latencies. Additionally, I aim to tackle challenges such as cache size constraints and self-correcting false hits using cache compression and response summarization. By integrating SLA-driven model customization methods, this thrust will enable robust and reliable GenAI systems that balance cost efficiency with enterprise-level performance requirements.

Bibliography

[1] R. I. S. Khan, A. H. Yazdani, Y. Fu, A. K. Paul, B. Ji, X. Jian, Y. Cheng, and A. R. Butt, "SHADE: Enable fundamental cacheability for distributed deep learning training," in *21st USENIX Conference on File and Storage Technologies (FAST 23)*. Santa Clara, CA: USENIX Association, Feb. 2023, pp. 135–152. [Online]. Available: https://www.usenix.org/conference/fast23/presentation/khan

[2] R. I. S. Khan, A. K. Paul, Y. Cheng, X. S. Jian, and A. R. Butt, "FedCaSe: Enhancing federated learning with heterogeneity-aware caching and scheduling," in *Proceedings of the 2024 ACM Symposium on Cloud Computing*, 2024, pp. 52–68.

[3] R. I. S. Khan, K. Jain, H. Shen, A. Mallick, A. Parayil, A. Kulkarni, S. Kofsky, P. Choudhary, R. S. Amant, R. Wang, Y. Cheng, A. R. Butt, V. Ruhle, C. Bansal, and S. Rajmohan, "Ensuring fair llm serving amid diverse applications," in *ArXiv*, 2025. [Online]. Available: https://arxiv.org/pdf/2411.15997

[4] S. Abraham, A. K. Paul, R. I. S. Khan, and A. R. Butt, "On the use of containers in high performance computing environments," in *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. IEEE, 2020, pp. 284–293.

[5] N. Dryden, R. Böhringer, T. Ben-Nun, and T. Hoefler, "Clairvoyant prefetching for distributed machine learning i/o," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–15.

[6] Y. Sheng, S. Cao, D. Li, B. Zhu, Z. Li, D. Zhuo, J. E. Gonzalez, and I. Stoica, "Fairness in serving large language models," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 965–988.

[7] S. Pumma, M. Si, W.-C. Feng, and P. Balaji, "Scalable deep learning via i/o analysis and optimization," *ACM Transactions on Parallel Computing (TOPC)*, vol. 6, no. 2, pp. 1–34, 2019.

[8] R. Böhringer, N. Dryden, T. Ben-Nun, and T. Hoefler, "Clairvoyant prefetching for distributed machine learning i/o," *arXiv preprint arXiv:2101.08734*, 2021.

[9] Google, "Usage limits in gemini for google workspace," https://support.google.com/a/users/answer/14796585?hl=en, 2024.

[10] Microsoft, "Microsoft graph throttling guidance," https://learn.microsoft.com/en-us/graph/throttling, 2024.

[11] C. Anchia, "Navigating llm service interruptions: Strategies for seamless transitions," https://tinyurl.com/llmserviceinterruptions, 2024.

[12] J. Mohan, A. Phanishayee, A. Raniwala, and V. Chidambaram, "Analyzing and mitigating data stalls in dnn training," *arXiv preprint arXiv:2007.06775*, 2020.

[13] A. V. Kumar and M. Sivathanu, "Quiver: An informed storage cache for deep learning," in *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, 2020, pp. 283–296.

[14] Y. Sheng, S. Cao, D. Li, C. Hooper, N. Lee, S. Yang, C. Chou, B. Zhu, L. Zheng, K. Keutzer *et al.*, "Slora: Scalable serving of thousands of lora adapters," *Proceedings of Machine Learning and Systems*, vol. 6, pp. 296–311, 2024.

[15] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A distributed serving system for {Transformer-Based} generative models," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 521–538.

[16] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale deep unsupervised learning using graphics processors," in *Proceedings of the 26th annual international conference on machine learning*, 2009, pp. 873–880.

[17] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. a. Ranzato, A. Senior, P. Tucker, K. Yang, Q. Le, and A. Ng, "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., vol. 25. Curran Associates, Inc., 2012. [Online]. Available: https://proceedings.neurips.cc/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf

[18] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and z. Chen, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/093f65e080a295f8076b1c5722a46aa2-Paper.pdf

[19] 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, 2013, p. 2.

[20] Y. Bao, Y. Peng, Y. Chen, and C. Wu, "Preemptive all-reduce scheduling for expediting distributed dnn training," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2020, pp. 626–635.

[21] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," 2004.

[22] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin *et al.*, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[23] "Memcached," https://memcached.org/.

[24] "Redis," https://redis.io/.

[25] N. Megiddo and D. S. Modha, "{ARC}: A {Self-Tuning}, low overhead replacement cache," in *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, 2003.

[26] X. Zeng, M. Yan, and M. Zhang, "Mercury: Efficient on-device distributed dnn training via stochastic importance sampling," in *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, 2021, pp. 29–41.

[27] W. Chen, S. He, Y. Xu, X. Zhang, S. Yang, S. Hu, S. Xian-He, and G. Chen, "icache: An importance-sampling-informed cache for accelerating i/o-bound dnn model training," in *2023 IEEE International Symposium on High-Performance Computer Architecture*, 2023.

[28] L. Liao, H. Li, W. Shang, and L. Ma, "An empirical study of the impact of hyperparameter tuning and model optimization on the performance properties of deep neural networks," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 3, pp. 1–40, 2022.

[29] C. Pinto, Y. Gkoufas, A. Reale, S. Seelam, and S. Eliuk, "Hoard: A distributed data caching system to accelerate deep learning training on the cloud," *arXiv preprint arXiv:1812.00669*, 2018.

[30] Z. Zhang, L. Huang, J. G. Pauloski, and I. T. Foster, "Efficient i/o for neural network training with compressed data," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 409–418.

[31] Y. Zhu, F. Chowdhury, H. Fu, A. Moody, K. Mohror, K. Sato, and W. Yu, "Entropy-aware i/o pipelining for large-scale deep learning on hpc systems," in *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2018, pp. 145–156.

[32] L. Wang, S. Ye, B. Yang, Y. Lu, H. Zhang, S. Yan, and Q. Luo, "Diesel: A dataset-based distributed storage and caching system for large-scale deep learning training," in *49th International Conference on Parallel Processing-ICPP*, 2020, pp. 1–11.

[33] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Artificial intelligence and statistics*. PMLR, 2017, pp. 1273–1282.

[34] C. Huang, J. Huang, and X. Liu, "Cross-silo federated learning: Challenges and opportunities," *arXiv preprint arXiv:2206.12949*, 2022.

[35] S. P. Karimireddy, M. Jaggi, S. Kale, M. Mohri, S. Reddi, S. U. Stich, and A. T. Suresh, "Breaking the centralized barrier for cross-device federated learning," *Advances in Neural Information Processing Systems*, vol. 34, pp. 28 663–28 676, 2021.

[36] P. Kairouz, H. B. McMahan, B. Avent, A. Bellet, M. Bennis, A. N. Bhagoji, K. Bonawitz, Z. Charles, G. Cormode, R. Cummings *et al.*, "Advances and open problems in federated learning," *Foundations and Trends® in Machine Learning*, vol. 14, no. 1–2, pp. 1–210, 2021.

[37] Q. Yang, Y. Liu, T. Chen, and Y. Tong, "Federated machine learning: Concept and applications," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 10, no. 2, pp. 1–19, 2019.

[38] S. Hardy, W. Henecka, H. Ivey-Law, R. Nock, G. Patrini, G. Smith, and B. Thorne, "Private federated learning on vertically partitioned data via entity resolution and additively homomorphic encryption," *arXiv preprint arXiv:1711.10677*, 2017.

[39] K. Wei, J. Li, C. Ma, M. Ding, S. Wei, F. Wu, G. Chen, and T. Ranbaduge, "Vertical federated learning: Challenges, methodologies and experiments," *arXiv preprint arXiv:2202.04309*, 2022.

[40] S. Saha and T. Ahmad, "Federated transfer learning: Concept and applications," *Intelligenza Artificiale*, vol. 15, no. 1, pp. 35–44, 2021.

[41] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konečnỳ, S. Mazzocchi, B. McMahan *et al.*, "Towards federated learning at scale: System design," *Proceedings of machine learning and systems*, vol. 1, pp. 374–388, 2019.

[42] M. Paulik, M. Seigel, H. Mason, D. Telaar, J. Kluivers, R. van Dalen, C. W. Lau, L. Carlson, F. Granqvist, C. Vandevelde *et al.*, "Federated evaluation and tuning for on-device personalization: System design & applications," *arXiv preprint arXiv:2102.08503*, 2021.

[43] H. Ludwig, N. Baracaldo, G. Thomas, Y. Zhou, A. Anwar, S. Rajamoni, Y. Ong, J. Radhakrishnan, A. Verma, M. Sinn *et al.*, "Ibm federated learning: an enterprise framework white paper v0. 1," *arXiv preprint arXiv:2007.10987*, 2020.

[44] C. Xie, S. Koyejo, and I. Gupta, "Asynchronous federated optimization," *arXiv preprint arXiv:1903.03934*, 2019.

[45] J. Nguyen, K. Malik, H. Zhan, A. Yousefpour, M. Rabbat, M. Malek, and D. Huba, "Federated learning with buffered asynchronous aggregation," in *International Conference on Artificial Intelligence and Statistics*. PMLR, 2022, pp. 3581–3607.

[46] D. Huba, J. Nguyen, K. Malik, R. Zhu, M. Rabbat, A. Yousefpour, C.-J. Wu, H. Zhan, P. Ustinov, H. Srinivas *et al.*, "Papaya: Practical, private, and scalable federated learning," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 814–832, 2022.

[47] C. Xu, Y. Qu, Y. Xiang, and L. Gao, "Asynchronous federated learning on heterogeneous devices: A survey," *arXiv preprint arXiv:2109.04269*, 2021.

[48] F. Lai, X. Zhu, H. V. Madhyastha, and M. Chowdhury, "Oort: Efficient federated learning via guided participant selection," in *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021, pp. 19–35.

[49] Z. Chai, A. Ali, S. Zawad, S. Truex, A. Anwar, N. Baracaldo, Y. Zhou, H. Ludwig, F. Yan, and Y. Cheng, "Tifl: A tier-based federated learning system," in *Proceedings of the 29th international symposium on high-performance parallel and distributed computing*, 2020, pp. 125–136.

[50] T. Li, M. Sanjabi, A. Beirami, and V. Smith, "Fair resource allocation in federated learning," *arXiv preprint arXiv:1905.10497*, 2019.

[51] J. Wang and G. Joshi, "Adaptive communication strategies to achieve the best error-runtime trade-off in local-update sgd," *Proceedings of Machine Learning and Systems*, vol. 1, pp. 212–229, 2019.

[52] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[53] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, J. Heek, K. Xiao, S. Agrawal, and J. Dean, "Efficiently scaling transformer inference," *Proceedings of Machine Learning and Systems*, vol. 5, pp. 606–624, 2023.

[54] C. Lin, Z. Han, C. Zhang, Y. Yang, F. Yang, C. Chen, and L. Qiu, "Parrot: Efficient serving of llm-based applications with semantic variable," in *OSDI*, 2024.

[55] Q. Wu, G. Bansal, J. Zhang, Y. Wu, S. Zhang, E. Zhu, B. Li, L. Jiang, X. Zhang, and C. Wang, "Autogen: Enabling next-gen llm applications via multi-agent conversation framework," *arXiv preprint arXiv:2308.08155*, 2023.

[56] S. Hong, X. Zheng, J. Chen, Y. Cheng, J. Wang, C. Zhang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou *et al.*, "Metagpt: Meta programming for multi-agent collaborative framework," *arXiv preprint arXiv:2308.00352*, 2023.

[57] NVIDIA, "Fastertransformer," https://github.com/NVIDIA/FasterTransformer, 2023.

[58] ——, "Triton," https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/user_guide/model_configuration.html#dynamic-batcher, 2024.

[59] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 611–626.

[60] LightLLM, "LightLLM," https://github.com/ModelTC/lightllm, 2024.

[61] OpenAI, "Why's gpt 4o insanely limited to free users and even plus users? it literally barely gives you 5 messages in 5-6 hours to the free users," https://tinyurl.com/openailimits, 2024.

[62] Z. Hu, J. Tang, Z. Wang, K. Zhang, L. Zhang, and Q. Sun, "Deep learning for image-based cancer detection and diagnosis- a survey," *Pattern Recognition*, vol. 83, pp. 134–149, 2018.

[63] P. Danaee, R. Ghaeini, and D. A. Hendrix, "A deep learning approach for cancer detection and relevant gene identification," in *PACIFIC SYMPOSIUM ON BIOCOMPUTING 2017*. World Scientific, 2017, pp. 219–229.

[64] A. Cruz-Roa, H. Gilmore, A. Basavanhally, M. Feldman, S. Ganesan, N. N. Shih, J. Tomaszewski, F. A. González, and A. Madabhushi, "Accurate and reproducible invasive breast cancer detection in whole-slide images: A deep learning approach for quantifying tumor extent," *Scientific reports*, vol. 7, p. 46450, 2017.

[65] R. Miotto, F. Wang, S. Wang, X. Jiang, and J. T. Dudley, "Deep learning for healthcare: review, opportunities and challenges," *Briefings in bioinformatics*, vol. 19, no. 6, pp. 1236–1246, 2018.

[66] M. S. Hossain and G. Muhammad, "Environment classification for urban big data using deep learning," *IEEE Communications Magazine*, vol. 56, no. 11, pp. 44–50, 2018.

[67] B. Juanals and J.-L. Minel, "Categorizing air quality information flow on twitter using deep learning tools," in *International Conference on Computational Collective Intelligence*. Springer, 2018, pp. 109–118.

[68] V. I. Jurtz, A. R. Johansen, M. Nielsen, J. J. Almagro Armenteros, H. Nielsen, C. K. Sønderby, O. Winther, and S. K. Sønderby, "An introduction to deep learning on biological sequence data: examples and solutions," *Bioinformatics*, vol. 33, no. 22, pp. 3685–3690, 2017.

[69] P. Baldi, P. Sadowski, and D. Whiteson, "Searching for exotic particles in high-energy physics with deep learning," *Nature communications*, vol. 5, no. 1, pp. 1–9, 2014.

[70] J. B. Heaton, N. G. Polson, and J. H. Witte, "Deep learning for finance: deep portfolios," *Applied Stochastic Models in Business and Industry*, vol. 33, no. 1, pp. 3–12, 2017.

[71] X. Ding, Y. Zhang, T. Liu, and J. Duan, "Deep learning for event-driven stock prediction," in *Twenty-fourth international joint conference on artificial intelligence*, 2015.

[72] E. Chong, C. Han, and F. C. Park, "Deep learning networks for stock market analysis and prediction: Methodology, data representations, and case studies," *Expert Systems with Applications*, vol. 83, pp. 187–205, 2017.

[73] J. Fombellida, I. Martín-Rubio, S. Torres-Alegre, and D. Andina, "Tackling business intelligence with bioinspired deep learning," *Neural Computing and Applications*, pp. 1–8, 2018.

[74] S. Bhattacharya, S. R. K. Somayaji, T. R. Gadekallu, M. Alazab, and P. K. R. Maddikunta, "A review on deep learning for future smart cities," *Internet Technology Letters*, p. e187, 2020.

[75] M. Aqib, R. Mehmood, A. Albeshri, and A. Alzahrani, "Disaster management in smart cities by forecasting traffic plan using deep learning and gpus," in *International Conference on Smart Cities, Infrastructure, Technologies and Applications*. Springer, 2017, pp. 139–154.

[76] J. Wang, Y. Ma, L. Zhang, R. X. Gao, and D. Wu, "Deep learning for smart manufacturing: Methods and applications," *Journal of Manufacturing Systems*, vol. 48, pp. 144–156, 2018.

[77] J. F. Arinez, Q. Chang, R. X. Gao, C. Xu, and J. Zhang, "Artificial intelligence in advanced manufacturing: Current status and future outlook," *Journal of Manufacturing Science and Engineering*, vol. 142, no. 11, 2020.

[78] V. Rausch, A. Hansen, E. Solowjow, C. Liu, E. Kreuzer, and J. K. Hedrick, "Learning a deep neural net policy for end-to-end control of autonomous vehicles," in *2017 American Control Conference (ACC)*. IEEE, 2017, pp. 4914–4919.

[79] B. Huval, T. Wang, S. Tandon, J. Kiske, W. Song, J. Pazhayampallil, M. Andriluka, P. Rajpurkar, T. Migimatsu, R. Cheng-Yue *et al.*, "An empirical evaluation of deep learning on highway driving," *arXiv preprint arXiv:1504.01716*, 2015.

[80] H. Li, K. Ota, and M. Dong, "Learning iot in edge: Deep learning for the internet of things with edge computing," *IEEE network*, vol. 32, no. 1, pp. 96–101, 2018.

[81] "Global Conversational AI Market Report 2021," https://tinyurl.com/Global-AI-Market-Report-2021.

[82] X.-W. Chen and X. Lin, "Big data deep learning: challenges and perspectives," *IEEE access*, vol. 2, pp. 514–525, 2014.

[83] C. P. Chen and C.-Y. Zhang, "Data-intensive applications, challenges, techniques and technologies: A survey on big data," *Information sciences*, vol. 275, pp. 314–347, 2014.

[84] J. Howard and S. Gugger, "Fastai: a layered api for deep learning," *Information*, vol. 11, no. 2, p. 108, 2020.

[85] Q. V. Le, J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, and A. Y. Ng, "On optimization methods for deep learning," in *ICML*, 2011.

[86] M. Khan, D. Nielsen, V. Tangkaratt, W. Lin, Y. Gal, and A. Srivastava, "Fast and scalable Bayesian deep learning by weight-perturbation in Adam," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. PMLR, 10–15 Jul 2018, pp. 2611–2620. [Online]. Available: http://proceedings.mlr.press/v80/khan18a.html

[87] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 595–610.

[88] S. H. Hashemi, S. A. Jyothi, and R. H. Campbell, "Tictac: Accelerating distributed deep learning with communication scheduling," *arXiv preprint arXiv:1803.03288*, 2018.

[89] Z. Tang, S. Shi, X. Chu, W. Wang, and B. Li, "Communication-efficient distributed deep learning: A comprehensive survey," *arXiv preprint arXiv:2003.06307*, 2020.

[90] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li, "Terngrad: Ternary gradients to reduce communication in distributed deep learning," *arXiv preprint arXiv:1705.07878*, 2017.

[91] S. Shi, Q. Wang, X. Chu, and B. Li, "A dag model of synchronous stochastic gradient descent in distributed deep learning," in *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2018, pp. 425–432.

[92] S. Shi, Q. Wang, X. Chu, B. Li, Y. Qin, R. Liu, and X. Zhao, "Communication-efficient distributed deep learning with merged gradient sparsification on gpus," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2020, pp. 406–415.

[93] J. Bae, J. Lee, Y. Jin, S. Son, S. Kim, H. Jang, T. J. Ham, and J. W. Lee, "Flashneuron: Ssd-enabled large-batch training of very deep neural networks," in *19th {USENIX} Conference on File and Storage Technologies ({FAST} 21)*, 2021, pp. 387–401.

[94] P. Braam, "The lustre storage architecture," *arXiv preprint arXiv:1903.01955*, 2019.

[95] J. Heichler, "An introduction to beegfs," 2014.

[96] S. Microsystems, "Rfc1094: Nfs: Network file system protocol specification," USA, 1989.

[97] "Amazon S3." https://aws.amazon.com/pm/serv-s3/.

[98] "Google Cloud Storage." https://cloud.google.com/storage.

[99] Z. Zhang, L. Huang, U. Manor, L. Fang, G. Merlo, C. Michoski, J. Cazes, and N. Gaffney, "Fanstore: Enabling efficient and scalable i/o for distributed deep learning," 2018.

[100] Google, "YouTube-8M Segments Dataset," https://research.google.com/youtube8m/.

[101] Y. Oyama, N. Maruyama, N. Dryden, E. McCarthy, P. Harrington, J. Balewski, S. Matsuoka, P. Nugent, and B. Van Essen, "The case for strong scaling in deep learning: Training large 3d cnns with hybrid parallelism," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 7, pp. 1641–1652, 2020.

[102] F. Abodo, R. Rittmuller, B. Sumner, and A. Berthaume, "Detecting work zones in shrp 2 nds videos using deep learning based computer vision," in *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2018, pp. 679–686.

[103] "CSCS. 2021. Piz Daint." https://www.cscs.ch/computers/piz-daint/.

[104] "RIKEN Center for Computational Science. 2021. About Fugaku." https://www.r-ccs.riken.jp/en/fugaku/about/.

[105] S. Balaban, "Deep learning and face recognition: the state of the art," *Biometric and surveillance technology for human and activity identification XII*, vol. 9457, pp. 68–75, 2015.

[106] A. Tunick, "On benchmarking multiple gpu computing resources for faster training of deep neural networks," CCDC Army Research Laboratory Adelphi United States, Tech. Rep., 2020.

[107] J. M. Karnuta, M. P. Murphy, B. C. Luu, M. J. Ryan, H. S. Haeberle, N. M. Brown, R. Iorio, A. F. Chen, and P. N. Ramkumar, "Artificial intelligence for automated implant identification in total hip arthroplasty: A multicenter external validation study exceeding two million plain radiographs," *The Journal of Arthroplasty*, 2022.

[108] "Amazon EC2 Spot Instances. Run Fault Tolerant workloads for up to 90% off." https://aws.amazon.com/ec2/spot/.

[109] "Use Spot VMs with Batch," https://learn.microsoft.com/en-us/azure/batch/batch-spot-vms.

[110] A. Katharopoulos and F. Fleuret, "Not all samples are created equal: Deep learning with importance sampling," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. PMLR, 10–15 Jul 2018, pp. 2525–2534. [Online]. Available: http://proceedings.mlr.press/v80/katharopoulos18a.html

[111] I. Loshchilov and F. Hutter, "Online batch selection for faster training of neural networks," *arXiv preprint arXiv:1511.06343*, 2015.

[112] T. B. Johnson and C. Guestrin, "Training deep models faster with robust, approximate importance sampling," *Advances in Neural Information Processing Systems*, vol. 31, pp. 7265–7275, 2018.

[113] G. Alain, A. Lamb, C. Sankar, A. Courville, and Y. Bengio, "Variance reduction in sgd by distributed importance sampling," *arXiv preprint arXiv:1511.06481*, 2015.

[114] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[115] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems journal*, vol. 5, no. 2, pp. 78–101, 1966.

[116] K. Zhou, S. Sun, H. Wang, P. Huang, X. He, R. Lan, W. Li, W. Liu, and T. Yang, "Demystifying cache policies for photo stores at scale: A tencent case study," in *Proceedings of the 2018 International Conference on Supercomputing*, ser. ICS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 284–294. [Online]. Available: https://doi.org/10.1145/3205289.3205299

[117] NVIDIA Corporation, "nvidia-smi - NVIDIA System Management Interface program," http://manpages.ubuntu.com/manpages/precise/man1/alt-nvidia-current-smi.1.html.

[118] Sebastien Godard, "sar(1) - Linux Man Page," 2021, https://linux.die.net/man/1/sar.

[119] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbach, A. Rocha, and J. Stubbs, "Lessons learned from the chameleon testbed," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

[120] J. Chung, Z. Liu, R. Kettimuthu, and I. Foster, "Elastic data transfer infrastructure (dti) on the chameleon cloud," in *2019 IEEE 27th International Conference on Network Protocols (ICNP)*. IEEE, 2019, pp. 1–2.

[121] J. Y. Chuah, "Machine learning gpu power measurement on chameleon cloud," in *Proceedings of the10th International Conference on Utility and Cloud Computing*, 2017, pp. 181–181.

[122] K. Keahey, P. Riteau, D. Stanzione, T. Cockerill, J. Mambretti, P. Rad, and P. Ruth, "Chameleon: a scalable production testbed for computer science research," in *Contemporary High Performance Computing*. CRC Press, 2019, pp. 123–148.

[123] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in neural information processing systems*, 2019, pp. 8026–8037.

[124] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[125] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[126] 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.

[127] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[128] "POLARIS Market Research," https://tinyurl.com/polarisfedmarket, 2021.

[129] H. Wu and P. Wang, "Node selection toward faster convergence for federated learning on non-iid data," *IEEE Transactions on Network Science and Engineering*, vol. 9, no. 5, pp. 3099–3111, 2022.

[130] Y. J. Cho, J. Wang, and G. Joshi, "Towards understanding biased client selection in federated learning," in *International Conference on Artificial Intelligence and Statistics*. PMLR, 2022, pp. 10 351–10 375.

[131] T. Nishio and R. Yonetani, "Client selection for federated learning with heterogeneous resources in mobile edge," in *ICC 2019-2019 IEEE international conference on communications (ICC)*. IEEE, 2019, pp. 1–7.

[132] W. Chen, S. Horvath, and P. Richtarik, "Optimal client sampling for federated learning," *arXiv preprint arXiv:2010.13723*, 2020.

[133] S. Wang, S. Yang, H. Li, X. Zhang, C. Zhou, C. Xu, F. Qian, N. Wang, and Z. Xu, "Pyramidfl: A fine-grained client selection framework for effiicient federated learning," in *28th ACM Annual International Conference on Mobile Computing and Networking, MobiCom 2022*. Association for Computing Machinery, 2022, pp. 542–555.

[134] E. Rizk, S. Vlaski, and A. H. Sayed, "Optimal importance sampling for federated learning," in *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2021, pp. 3095–3099.

[135] C. Gong, Z. Zheng, F. Wu, Y. Shao, B. Li, and G. Chen, "To store or not? online data selection for federated learning with limited storage," in *Proceedings of the ACM Web Conference 2023*, 2023, pp. 3044–3055.

[136] F. Lai, Y. Dai, S. Singapuram, J. Liu, X. Zhu, H. Madhyastha, and M. Chowdhury, "Fedscale: Benchmarking model and system performance of federated learning at scale," in *International Conference on Machine Learning*. PMLR, 2022, pp. 11 814–11 827.

[137] D. G. Murray, J. Simsa, A. Klimovic, and I. Indyk, "tf. data: A machine learning data processing framework," *arXiv preprint arXiv:2101.12127*, 2021.

[138] G. Cohen, S. Afshar, J. Tapson, and A. Van Schaik, "Emnist: Extending mnist to handwritten letters," in *2017 international joint conference on neural networks (IJCNN)*. IEEE, 2017, pp. 2921–2926.

[139] S. Caldas, S. M. K. Duddu, P. Wu, T. Li, J. Konečnỳ, H. B. McMahan, V. Smith, and A. Talwalkar, "Leaf: A benchmark for federated settings," *arXiv preprint arXiv:1812.01097*, 2018.

[140] P. Zhao and T. Zhang, "Stochastic optimization with importance sampling for regularized loss minimization," in *international conference on machine learning*. PMLR, 2015, pp. 1–9.

[141] "Mobile GPU rankings 2021," https://www.techcenturion.com/mobile-gpu-rankings, 2021.

[142] "Smartphone GPU ranking," https://www.phoneworld.com.pk/best-gpu-ranking-list/, 2021.

[143] D. R. Jones, M. Schonlau, and W. J. Welch, "Efficient global optimization of expensive black-box functions," *Journal of Global optimization*, vol. 13, pp. 455–492, 1998.

[144] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[145] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[146] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.

[147] A. Ignatov, R. Timofte, A. Kulik, S. Yang, K. Wang, F. Baum, M. Wu, L. Xu, and L. Van Gool, "Ai benchmark: All about deep learning on smartphones in 2019," in *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*. IEEE, 2019, pp. 3617–3635.

[148] "The M-Lab MobiPerf Data Set," https://www.measurementlab.net/tests/mobiperf/, 2024.

[149] Z. Chai, Y. Chen, A. Anwar, L. Zhao, Y. Cheng, and H. Rangwala, "Fedat: A high-performance and communication-efficient federated learning system with asynchronous tiers," in *Proceedings of the international conference for high performance computing, networking, storage and analysis*, 2021, pp. 1–16.

[150] Microsoft, "Copilot for microsoft 365," https://www.microsoft.com/en-us/microsoft-365/enterprise/copilot-for-microsoft-365, 2024.

[151] L. Chen, "Potentials of multitenancy fine-tuned llm serving," https://le.qun.ch/en/blog/2023/09/11/multi-lora-potentials/, 2023.

[152] J. Nagle, "On packet switches with infinite storage," *IEEE transactions on communications*, vol. 35, no. 4, pp. 435–438, 1987.

[153] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," *ACM SIGCOMM Computer Communication Review*, vol. 19, no. 4, pp. 1–12, 1989.

[154] Anthropic, "Pricing," https://www.anthropic.com/pricing, 2024.

[155] OpenAI, "Pricing — openai," https://openai.com/api/pricing/, 2024.

[156] "OpenAI API Pricing," https://openai.com/api/pricing/, 2025.

[157] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 261–276.

[158] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *8th USENIX symposium on networked systems design and implementation (NSDI 11)*, 2011.

[159] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan, "Altruistic scheduling in {Multi-Resource} clusters," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 65–80.

[160] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla, "Themis: Fair and efficient {GPU} cluster scheduling," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 289–304.

[161] S. Chaudhary, R. Ramjee, M. Sivathanu, N. Kwatra, and S. Viswanatha, "Balancing efficiency and fairness in heterogeneous gpu clusters for deep learning," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[162] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia, "{Heterogeneity-Aware} cluster scheduling policies for deep learning workloads," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 481–498.

[163] A. Qiao, S. K. Choe, S. J. Subramanya, W. Neiswanger, Q. Ho, H. Zhang, G. R. Ganger, and E. P. Xing, "Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning," in *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021.

[164] S. Jayaram Subramanya, D. Arfeen, S. Lin, A. Qiao, Z. Jia, and G. R. Ganger, "Sia: Heterogeneity-aware, goodput-optimized ml-cluster scheduling," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 642–657.

[165] A. Agrawal, A. Panwar, J. Mohan, N. Kwatra, B. S. Gulavani, and R. Ramjee, "Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills," *arXiv preprint arXiv:2308.16369*, 2023.

[166] Y. Jin, C.-F. Wu, D. Brooks, and G.-Y. Wei, "s^3: Increasing gpu utilization during generative inference for higher throughput," *Advances in Neural Information Processing Systems*, vol. 36, pp. 18 015–18 027, 2023.

[167] B. Wu, Y. Zhong, Z. Zhang, G. Huang, X. Liu, and X. Jin, "Fast distributed inference serving for large language models," *arXiv preprint arXiv:2305.05920*, 2023.

[168] A. Agrawal, N. Kedia, A. Panwar, J. Mohan, N. Kwatra, B. S. Gulavani, A. Tumanov, and R. Ramjee, "Taming throughput-latency tradeoff in llm inference with sarathi-serve," *arXiv preprint arXiv:2403.02310*, 2024.

[169] P. Patel, E. Choukse, C. Zhang, A. Shah, Í. Goiri, S. Maleki, and R. Bianchini, "Splitwise: Efficient generative llm inference using phase splitting," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2024, pp. 118–132.

[170] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. Zhang, "Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving," *arXiv preprint arXiv:2401.09670*, 2024.

[171] C. Holmes, M. Tanaka, M. Wyatt, A. A. Awan, J. Rasley, S. Rajbhandari, R. Y. Aminabadi, H. Qin, A. Bakhtiari, L. Kurilenko *et al.*, "Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference," *arXiv preprint arXiv:2401.08671*, 2024.

[172] S. Agarwal, S. Mitra, S. Chakraborty, S. Karanam, K. Mukherjee, and S. K. Saini, "Approximate caching for efficiently serving {Text-to-Image} diffusion models," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024, pp. 1173–1189.

[173] F. Wimbauer, B. Wu, E. Schoenfeld, X. Dai, J. Hou, Z. He, A. Sanakoyeu, P. Zhang, S. Tsai, J. Kohler *et al.*, "Cache me if you can: Accelerating diffusion models through block caching," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024, pp. 6211–6220.