

Interpretability and Debugging for Distributed Privacy Preserving Machine Learning

Waris Gill

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 & Applications

Muhammad Ali Gulzar, Chair

Eli Tilevich

Bo Ji

Bimal Viswanath

Ali Anwar

December 05, 2025

Blacksburg, Virginia

Keywords: Federated Learning, Privacy, Interpretability, Debugging, LLMs, CNNs

Copyright 2025, Waris Gill

Interpretability and Debugging for Distributed Privacy Preserving Machine Learning

Waris Gill

(ABSTRACT)

Machine learning systems increasingly rely on privacy-preserving distributed training to leverage sensitive data across multiple organizations without centralization. Federated Learning (FL), a distributed privacy-preserving machine learning paradigm, enables hospitals, devices, and enterprises to collaboratively train models without accessing raw client data (e.g., Siri, Alexa, and healthcare applications). Centralized machine learning benefits from rich debugging and interpretability techniques enabled by transparent access to training data. However, FL removes this transparency, rendering traditional techniques ineffective and making debugging and interpretability a challenging open problem. This thesis addresses this challenge by asking: *How can we design automated debugging and interpretability methods for federated learning that effectively localize faults and attribute global model predictions without degrading performance or violating FL's core privacy principles?* The central insight is that effective debugging and interpretability can be achieved by analyzing model parameters, activations, and gradients-information already shared or derivable in standard FL protocols (e.g., FedAvg). We present three contributions. First, towards fault localization, we redesign traditional differential testing to operate on neuron activations produced by auto-generated inputs, exploiting the fact that faulty clients produce models with divergent activations. Second, we introduce *neuron provenance*, which decouples data-influence tracking from data access. It identifies influential neurons via gradient-based weighting and decomposes them to client-specific origins, yielding ranked lists of responsible clients across

CNNs and Transformers. Third, we extend neuron provenance to federated LLMs, where autoregressive generation and billion-parameter scale make naive tracking infeasible. It introduces token-level provenance at targeted transformer layers, achieving high attribution accuracy across multiple LLM architectures. In each case, the solution operates entirely on information available at the aggregator, requiring no client-side instrumentation. Collectively, these contributions culminate in practical tools that integrate seamlessly with existing distributed ML workflows, enabling real-time debugging and transparent model insights for both classification and LLMs in FL.

Interpretability and Debugging for Distributed Privacy Preserving Machine Learning

Waris Gill

(GENERAL AUDIENCE ABSTRACT)

When multiple organizations like hospitals, smartphone companies, or banks want to train machine learning models together, they face a dilemma: combining their data would produce better AI, but sharing sensitive patient records, personal messages, or financial information is often illegal or dangerous. Federated Learning, a privacy-preserving approach to machine learning, solves this by allowing organizations to train AI collaboratively without ever sharing their private data. This technology already powers voice assistants like Siri and Alexa. However, this privacy comes with a hidden cost. When something goes wrong, such as when an AI makes a mistake or produces harmful outputs, developers cannot investigate because they cannot see the original data. Traditional debugging and interpretability approaches, designed for centralized data access, are not directly applicable in this distributed, privacy-constrained setting. This thesis develops new methods that allow developers to identify which organization caused a problem and understand why the AI behaves as it does, all without ever accessing private data. We introduce techniques that work across different types of AI, from image recognition systems to advanced language models. These contributions, now integrated into widely-used federated learning platforms, make privacy-preserving AI more trustworthy and deployable in sensitive domains like healthcare and finance.

Dedication

To my parents, my wife, and my son

Acknowledgments

This dissertation is supported by the generous funding of the **U.S. National Science Foundation (NSF)** under Award #2452818 “Testbed for Enhancing Privacy and Robustness of Federated Learning Systems,” Award #2106420 “Reinventing Fuzz Testing for Data and Compute Intensive Systems,” and the 2024-25 **Amazon**-Virginia Tech Initiative for Efficient and Robust Machine Learning for the project “Privacy-Preserving Semantic Cache for LLM-based Services.”

I am immensely grateful to my dissertation committee: Dr. Muhammad Ali Gulzar, Dr. Eli Tilevich, Dr. Ali Anwar, Dr. Bimal Viswanath, and Dr. Bo Ji. My advisor, Dr. Gulzar, has been instrumental in helping me complete this milestone. His mentorship kept me motivated, while his invaluable feedback brought rigor to my work and opened new research directions. Always accessible, he granted me the intellectual freedom to disagree and explore diverse projects beyond my dissertation. I have learned immensely from him; his mentorship has had a profound impact on my life, and I will continue to learn from him. Dr. Anwar’s support during my PhD was unwavering. His rigor, perspectives, guidance, and encouragement were instrumental. He was always available to discuss and explore new ideas. Dr. Tilevich was supportive throughout my PhD. I frequently discussed ideas and shared my work with him. I first met him in the CRC while working late one night, and from then on, he always encouraged me and expressed great appreciation for my work. His advice and appreciation meant a lot to me. I acknowledge my MS professors at Koç University, Istanbul: Dr. Attila Gürsoy and Dr. Öznur Özkasap. I finished my MS and started my PhD during the COVID pandemic, and their support throughout my MS and during those difficult times is something I will remember forever. Dr. Fareed Zaffar, my undergraduate advisor at LUMS, made my graduate studies possible. He wrote my letters of recommendation and helped me begin my graduate journey, and I owe him a great deal. Sir Rana Sana-ur-Rehman has been a

constant source of encouragement from Kamal Education Complex to Virginia Tech, and his long-standing guidance has been invaluable to my academic journey. During my final research internship at Microsoft, I had the pleasure of working with Matthew Dressman, who supported me greatly. I am deeply grateful to my lab members and friends: Tien, Sabaat, Hadi, Humayun, and Aravind. Collaborating with them was a joy, and the pleasant environment in our lab made all the difference. Thanks to Hadi for keeping us well-fed! I often joke that Humayun is my second advisor, though he insists I only use him for rubber-ducking. In fact, the provenance idea in this thesis originated during one of our walks home from the CS Department, as I discussed my thoughts with him (rubber-ducking, according to him). Ahmad Ghafoor is the reason I went to LUMS. To this day, he, Umar, Abdullah, Uncle Ghafoor, and Aunty Munawar have done so much for me—Aunty Munawar even calls me her fourth son. My friends and family, Danyal, Abdullah, Dr. Azhar, Dr. Ahsan, Palwisha, Fatima, and Dr. Eman, have supported me from the very beginning to this day. Dr. Tugrul and Ipek, my MS friends, supported me during my time in Turkey. Misbah Aapi and Saima Aapi, Sher, Nihal, Muneeb, Alyan, and Basit are people who prayed for me and have my deepest regards. I am grateful for the “support” of my son Azlan, who contributed by pressing random buttons on my laptop while I was writing this dissertation. I apologize to my wife Maryam and to Azlan for being busy most of the time, and I deeply appreciate their patience and support. My mother always believed I would study abroad. She has given me unwavering support, faith, and courage. From my first day of school to this very day, if it were not for her, I could not have completed any of my studies. My father supported her decisions and equally supported me. I am forever indebted to them for all that I have achieved. Finally, I dedicate this dissertation to my teachers, professors, and friends, as well as my parents, my wife Maryam, and my son Azlan. *I am deeply grateful to the Virginia Tech community.*

Contents

List of Figures	xiii
List of Tables	xviii
1 Introduction	1
1.1 Problem Formulation	2
1.2 Challenges of Interpretability and Debugging in FL	3
1.3 Dissertation Key Insights	5
1.3.1 Systematic Debugging for FL Applications	8
1.3.2 Interpretability in FL via Neuron Provenance	9
1.3.3 Token-Level Attribution for Federated LLMs	10
1.3.4 FL-Assisted User-Centric Semantic Cache for LLMs	11
1.4 Contributions	12
2 Background and Literature Review	14
2.1 Federated Learning	14
2.2 Debugging and Interpretability in ML	16
2.2.1 Interpretability in Machine Learning	17
2.2.2 Provenance Approaches in Machine Learning	19

2.3	Existing Caching Approaches	20
3	Systematic Debugging for FL Applications	22
3.1	Introduction	22
3.2	Motivation	26
3.3	Design of FedDebug	28
3.3.1	Selective Telemetry	29
3.3.2	Interactive Replay Debugging	30
3.3.3	Fix and Replay	33
3.4	Faulty Client Localization	33
3.5	Evaluation	37
3.5.1	FedDebug’s Performance	40
3.5.2	Localization of Faulty Client(s)	43
3.5.3	Neuron Activation Threshold	49
3.5.4	Threats to Validity	50
3.6	Summary	50
4	Interpretability in FL via Neuron Provenance	51
4.1	Introduction	51
4.2	Motivation	56
4.3	Challenges	57

4.4	Design of TraceFL	58
4.4.1	Determining Influential Neurons	60
4.4.2	Neuron Provenance Across Fusion	62
4.4.3	Measuring Client’s Contribution	64
4.5	Evaluation	67
4.5.1	TraceFL’s Localization Accuracy in Correct Predictions	70
4.5.2	TraceFL’s Localization Accuracy in Mispredictions	71
4.5.3	TraceFL’s Robustness	73
4.5.4	TraceFL’s Scalability	76
4.5.5	TraceFL’s Localization Time	77
4.5.6	Threat to Validity and Limitations	78
4.6	Summary	79
5	Token-Level Attribution for Federated LLMs	80
5.1	Introduction	80
5.2	Motivation	84
5.3	Design of ProToken	87
5.3.1	Provenance Attribution Strategy	89
5.3.2	Attributing Autoregressive Sequences	91
5.3.3	Layer Selection for Provenance Tractability	92

5.3.4	Weighted Attribution using Token Gradients	93
5.3.5	ProToken Multi-Layer Token Aggregation	95
5.4	Evaluation	96
5.4.1	Experimental Setup	97
5.4.2	RQ1: Cross Domain and Architecture Accuracy	99
5.4.3	RQ2: Relevance Filtering via Gradient Weighting	101
5.4.4	RQ3: Computational Tractability	102
5.4.5	RQ4: Scalability Analysis	104
5.5	Summary	106
6	FL-Assisted User-Centric Semantic Cache for LLMs	107
6.1	Introduction	107
6.2	Transformer and Embeddings	111
6.3	Design of MeanCache	112
6.3.1	FL Based Embedding Model Training	113
6.3.2	Client Training	115
6.3.3	Finding the Optimal Threshold for Cosine Similarity	116
6.3.4	Aggregation	116
6.3.5	Embeddings Compression using PCA	117
6.3.6	Cache Population and MeanCache Implementation	119

6.4	Evaluation	119
6.4.1	Evaluation Settings	121
6.4.2	MeanCache Comparison with Baseline	124
6.4.3	Contextual Queries	126
6.4.4	Embedding Compression and Impact on Storage Space	128
6.4.5	Privacy Preserving Embeddings Model Training	130
6.4.6	Cosine Similarity Threshold Impact on Semantic Matching	131
6.4.7	Infeasibility of Embedding Generation with Llama 2	133
6.5	Summary	134
7	Conclusion	135
7.1	Impact	137
7.2	Future Work	137
	Bibliography	140

List of Figures

1.1	In a centralized FL architecture, an aggregator sends a global model to clients (step 1). Each client trains the model on local data (step 2) and sends the locally trained model back to the server (step 3). The server aggregates all models to form a new global model (step 4).	2
3.1	Using FedDebug, a developer can set a <i>breakpoint</i> at round 20 . When the FL application finishes round 20 , FedDebug launches a Debugging Interface, reflected on the right. <i>Step next</i> (❷) takes the developer to the next step (round or client). <i>Step-in</i> increases the granularity of computation, <i>e.g.</i> , round to client level. <i>Resume</i> (❸) rejoins the current execution status of the FL application if no intrusive actions are taken. At a given round, FedDebug can automatically localize the faulty client (❹) and then resume (❺) upon which the global model will be recomputed without the faulty client. This model will replace the corresponding round's model, and FedDebug will start retraining from that round, round 22 , in the FL interface.	29
3.2	An overview of FedDebug's fault localization approach. It first selects a random input that invokes diverse model behavior (A). It then applies differential execution on clients' models to localize a faulty client (B).	34
3.3	Global model (ResNet-34) prediction accuracy in the presence of a faulty client with different noise rates. Lower noise rates hardly degrade global model performance.	40

3.4	FedDebug’s runtime overhead as a comparison between vanilla FL framework’s aggregation time with FedDebug enabled FL aggregation.	41
3.5	FedDebug’s debugging time contains input generation time and faulty client detection time and is compared against a round’s training time.	42
3.6	FedDebug localization performance when a faulty client has varying fault strength (<i>i.e.</i> , low noise rate).	43
3.7	FedDebug finds multiple faulty clients in a linear time. Total clients are 50 in each graph.	46
3.8	FedDebug retains scalability on a large number of clients.	47
3.9	FedDebug performance at neuron activation threshold on 30 clients, including five faulty clients.	48
4.1	Illustration of training, testing, and localization phases of the real-world motivating example. The FL global model correctly classifies two colon pathology images (original labels ‘Cancer-associated Stroma’ and ‘Mucus’). During responsible client localization, TraceFL accurately identifies the client most responsible for the prediction, <i>i.e.</i> , clients trained on data points with labels Mucus (Hospital H2) and ‘Cancer-associated Stroma’ (Hospital H6).	54
4.2	TraceFL performance on multiple datasets and models both on text and image classification tasks.	67
4.3	TraceFL performance on different data distributions. The X-axis represents the values of Dirichlet alpha.	73
4.4	Impact of DP noise on FL training accuracy.	74

4.5	TraceFL’s scalability when # of rounds increase.	76
4.6	Client localization of TraceFL vs. FedDebug.	77
5.1	Motivating example showing how ProToken identifies the client responsible for anomalous output.	84
5.2	ProToken Provenance Attribution Performance. Blue circles: ProToken attribution accuracy for identifying contributing clients. Orange squares: Model accuracy on benign responses. Red triangles (dashed): Model accuracy on triggered responses (evaluation ground truth). ProToken achieves on average attribution accuracy of 98.62%.	94
5.3	ProToken Client Contribution Probability Distributions. Red boxes: Clients 0-1 (contributors) receive high probabilities. Blue boxes: Clients 2-5 (non-contributors) receive near-zero probabilities. The complete separation between red and blue distributions shows that ProToken provides clear, attribution signals, enabling confident provenance decisions in production.	97
5.4	Average <i>per-layer (i.e., individual layer) attribution accuracy of ProToken</i> across 16 configurations (4 models × 4 domains). Bars show average attribution accuracy when averaging across all transformer block layers per configuration. Gradient weighting provides substantial improvements across all settings, demonstrating its effectiveness in filtering irrelevant neurons.	99
5.5	For each model, we vary the number of monitored layers (x-axis) and measure ProToken’s average provenance computation time (left y-axis, blue) and attribution accuracy (right y-axis).	103

5.6	ProToken maintains high attribution accuracy throughout, demonstrating effective scalability from 6 to 55 clients.	104
5.7	ProToken maintains clear separation between responsible (0-24) and non-responsible (25-54) clients.	105
6.1	MeanCache’s Workflow.	111
6.2	Privacy-Preserving Embedding Model FL Training in MeanCache.	114
6.3	Embeddings Compression using PCA.	117
6.4	Response times of 100 randomly sampled user queries to the Llama 2-based LLM service in three scenarios: without any semantic cache, with GPTCache, and with MeanCache. The integration of a semantic cache does not add significant overhead to non-duplicate queries, meaning it does not impede the performance of the LLM-based service. Moreover, it significantly reduces the average response times for duplicate queries (70-99) by serving them from the local cache.	120
6.5	Comparison of MeanCache and GPTCache on a set of 100 queries, including 70 non-duplicate and 30 duplicate queries, sent to the Llama 2-based LLM service. Queries 0 to 69 are non-duplicate (<i>i.e., real label is miss</i>), and GPTCache produces significantly higher false hits on these unique queries compared to MeanCache.	120
6.6	Confusion matrices for MeanCache and GPTCache on 1000 queries comprising 700 unique and 300 duplicate queries. Among the 700 unique queries, MeanCache produces only 89 false hits, while GPTCache generates 233 false hits.	122

6.7	Performance on Contextual Queries: MeanCache vs. Baseline. (a) reports MeanCache’s fewer false hits 3 vs. 54 of GPTCache. (b) reports higher true hits by MeanCache.	126
6.8	MeanCache reports three false hits compared to 54 false hits by GPTCache.	127
6.9	(a) MeanCache’s embedding compression reduces storage by 83% compared to GPTCache. (b) MeanCache’s semantic matching speed is 11% faster with compression enabled, while still outperforming GPTCache. (c) MeanCache’s F score is slightly lower with compression enabled, but it still outperforms GPTCache.	128
6.10	<i>MPNet</i> ’s FL training helps generate high-quality embeddings.	129
6.11	FL training boosts MeanCache’s query matching precision with <i>Albert</i>	129
6.12	MeanCache automatically optimizes <i>MPNet</i> ’s threshold.	131
6.13	MeanCache identifies an optimal threshold of 0.78 for <i>Albert</i>	131
6.14	Llama 2 takes significantly longer to compute embeddings and requires substantially more storage space than <i>Albert</i> and <i>MPNet</i>	132
6.15	Llama 2 does not perform well in semantic matching, even at optimal thresholds.	132

List of Tables

3.1	FedDebug’s debugging time and accuracy when localizing a faulty client in 36 different FL settings with 100 test inputs.	44
3.2	FedDebug’s fault localization in 32 FL configurations with multiple faulty clients, ranging from two to seven.	45
4.1	Comparison of TraceFL with FedDebug on localizing clients responsible for misprediction. FedDebug is compatible with image classification only and is effective under specific data distribution (<i>i.e.</i> , $\alpha = 1$).	72
4.2	Results of TraceFL with DP in FL.	75
4.3	Scalability results of TraceFL with different number of clients with GPT.	76
6.1	MeanCache outperforms GPTCache (baseline) on both standalone and contextual queries.	121

List of Abbreviations

CNNs Convolutional Neural Networks

FL Federated Learning

LLMs Large Language Models

NLP Natural Language Processing

Chapter 1

Introduction

Machine Learning (ML) revolutionized data-driven decision-making across various applications, from medical diagnosis to financial forecasting. ML models learn patterns from datasets to predict outcomes on unseen data leading to remarkable advances in image recognition and natural language processing (NLP). Recent advancements in Large Language Models (LLMs) (e.g., Llama, ChatGPT) have further revolutionized the field with state-of-the-art conversational capabilities. As ML models evolve, they drive real-world applications like personalized healthcare and intelligent chatbots, while raising critical concerns about data privacy, interpretability, and accountability.

Despite ML's successes, traditional ML often relies on centralized data collection, which poses significant privacy risks and regulatory challenges when handling sensitive information like medical records or personal user data. Federated learning [134] is a distributed, privacy-preserving ML paradigm that overcomes these limitations by allowing multiple clients (*e.g.*, smartphones, hospitals, or edge devices) to collaboratively train a global model without sharing their raw data. Instead, each client (*e.g.*, hospital) trains a local model on its private dataset and periodically shares only model weights with a central server. The server aggregates these parameters to build a global model, which is then redistributed to clients for further training in the next FL training round. This distributed training approach preserves data privacy, reduces network bandwidth consumption, and minimizes the risks of centralized data storage. FL is widely adopted in applications like Siri [15], Alexa [49, 88], Google

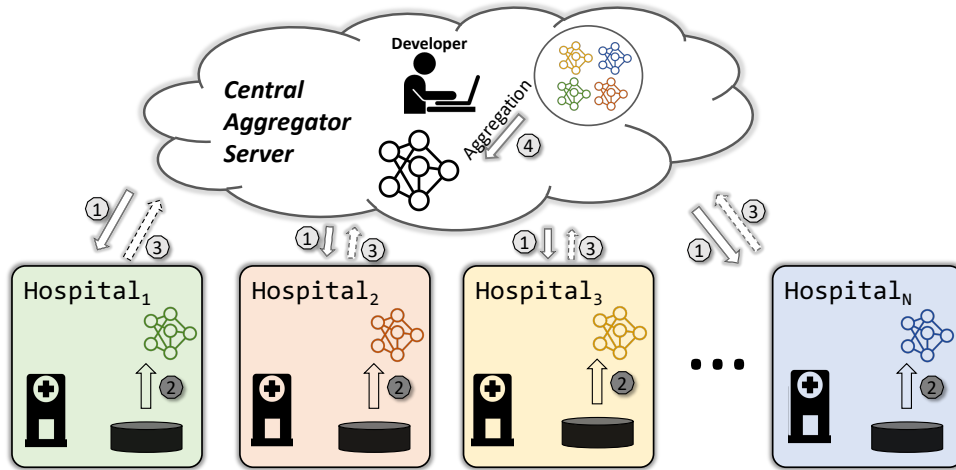


Figure 1.1: In a centralized FL architecture, an aggregator sends a global model to clients (step 1). Each client trains the model on local data (step 2) and sends the locally trained model back to the server (step 3). The server aggregates all models to form a new global model (step 4).

Keyboard [11], and healthcare domain [160]. More recently, FL has extended to Large Language Models (LLMs), enabling collaborative fine-tuning across organizations while preserving data privacy [107, 177, 198].

1.1 Problem Formulation

While FL solves critical privacy and data-centralization issues, it introduces a new open challenge of interpretability and debugging in FL deployments due to FL principles (*e.g.*, inability to access clients data) [97]. This challenge is absent or less severe in the traditional centralized ML.

Consider an FL deployment where multiple hospitals (clients) collaboratively train a global model to predict diseases from radiology images (Figure 1.1). Each hospital trains the model on its local data and shares only the model weights and performance metrics (*e.g.*, loss) with a central server. The developer at the server aims to identify faulty clients due to mislabeled or

poisoned data. However, without access to the clients' private data, pinpointing responsible or faulty clients (*e.g.*, clients with mislabeled or poisoned data) is challenging. Furthermore, even if the developer has access to some manually collected data, existing FL frameworks lack the tools to quantify each client's contribution to the global model's outputs (*e.g.*, rewarding clients based on their contributions to global model predictions). These scenarios underscore the need for specialized debugging and interpretability solutions for FL, as traditional ML interpretability methods are designed for centrally trained models and address orthogonal research problems, making them inadequate for FL's unique challenges.

Given the growing adoption of federated learning, we identify two key challenges that must be addressed to ensure effective, trustworthy FL deployments and to tackle the interpretability and debugging challenge [97]:

1. **Fault Localization:** How can developers pinpoint faulty clients when neither raw data nor test labels are accessible at the central server?
2. **Client Attribution:** How can one determine which client(s) is primarily responsible for a specific global model output? This challenge becomes particularly complex for federated LLMs, where distinguishing client contributions from pre-trained knowledge and handling autoregressive token generation present unique obstacles.

1.2 Challenges of Interpretability and Debugging in FL

In a standard ML pipeline, developers have direct access to training data, labels, and intermediate model states, making it easier to identify the source of errors or poor performance. In contrast, FL applications must operate under strict privacy constraints that prevent the central server also called aggregator (*i.e.*, application developer) from viewing the clients' lo-

cal data. Consequently, following challenges arise, which need to overcome while addressing the challenge of debugging and interpretability in FL while keeping FL principles intact.

- **Lack of Transparency:** FL setups often involve large numbers of ephemeral clients, each with unique data and unpredictable participation across rounds due to network or resource constraints [181, 207]. Without direct insight into client data or consistent client availability, pinpointing the cause of global model failures becomes exceedingly difficult.
- **Model Aggregation Complexity:** In each FL round, a subset of clients trains local models on diverse data and uploads updated parameters to a central server for aggregation. The resulting global model is thus a mixture of many local models. Determining which client (or group of clients) contributed to specific behaviors (good or bad) of the global model is a non-trivial task, especially for models with millions of parameters, such as LLMs.
- **Incompatibility with Existing Interpretability Tools:** Traditional debugging and interpretability approaches rely on transparent access to data that allows developers to isolate problems. These methods are not directly applicable in FL, since clients' data is inaccessible to the developer.
- **Heterogeneous Architecture:** Modern FL employs various models, including convolutional neural networks (CNNs) for vision and transformers (e.g., GPT, BERT, Llama) for language. These architectures, rely on specialized layers like attention mechanisms or convolutional layers. Designing a general debugging and interpretability solution that works across these heterogeneous architectures under privacy constraints is a significant technical hurdle.

- **Federated LLM Attribution Complexity:** The extension of FL to LLMs introduces unique attribution challenges absent in classification models. LLMs generate variable-length sequences through autoregressive token generation, where each token’s generation depends on previously generated tokens, creating cascading dependencies that complicate attribution. Furthermore, distinguishing contributions from federated training data versus the LLM’s pre-existing knowledge base is fundamentally ambiguous. The computational scale of LLMs, with billions of parameters, makes naive provenance tracking prohibitively expensive, requiring targeted, efficient attribution mechanisms that operate at token-level granularity while preserving privacy constraints.

1.3 Dissertation Key Insights

The central hypothesis of this dissertation is that *we can design specialized, privacy-preserving debugging and interpretability methods for FL applications to effectively localize faults and attribute predictions, without hindering FL’s performance or violating FL principles (e.g., accessing clients private data)*. We demonstrate this by leveraging several foundational insights that enable systematic debugging and interpretability in FL while preserving its privacy principles.

Debugging via Telemetry Replay. Traditional breakpoint debugging is impractical in FL because pausing distributed training disrupts all clients, risks data loss, and fundamentally conflicts with FL’s decentralized nature. We observe that instead of debugging a live FL application, we can selectively record concise runtime metrics (*e.g.*, clients’ models, training loss, hyperparameters) at the aggregator. This telemetry enables on-demand reconstruction of any prior FL state in a simulation environment, allowing developers to interactively debug,

step through rounds and clients, and perform root-cause analysis, all without disrupting the live training process or accessing client-side data.

Behavioral Divergence Detection via Neuron Activations. A central insight enabling fault localization is that all clients in an FL application share the same model architecture, making their internal behaviors directly comparable. To impact the global model, a faulty client’s model must behave differently than benign clients. We exploit this by fingerprinting model behavior through neuron activation patterns. On any given input, models’ internal neuron activations reveal behavioral similarities and divergences. This insight enables differential testing without ground truth labels. We auto-generate synthetic test inputs, select those that invoke diverse client behaviors, and identify faulty clients as those whose activation patterns deviate from the majority consensus.

Neuron Provenance via Linear Decomposition of FL Aggregation. A fundamental mathematical property underpins our attribution methods. FL aggregation algorithms (*e.g.*, FedAvg, FedProx) perform weighted linear combinations of client model parameters. This linearity means that each neuron in the global model is a weighted sum of corresponding neurons across client models, and consequently, the global model’s output on any input can be mathematically decomposed into per-client contributions. We formalize this as *neuron provenance*, a fine-grained lineage-capturing mechanism that traces how each client’s neurons contribute to the global model’s neurons, and ultimately to its predictions. This decomposition forms the theoretical foundation enabling tractable provenance in models with millions of parameters.

Gradient-Based Importance Weighting for Precise Attribution. Not all neurons contribute equally to a prediction [144, 213]. A dynamic subset activates and influences each specific output. To achieve precise attribution without noise from irrelevant neurons, we compute gradients of the global model’s prediction with respect to neuron activations.

These gradients act as importance weights, quantifying each neuron’s influence on the output and automatically filtering irrelevant contributions. By combining neuron-level provenance with gradient-based importance, we precisely rank client contributions. We map influential global model neurons to their corresponding client neurons, weight contributions by their gradient-derived importance, and aggregate across layers (with higher weight to later layers that encode task-specific knowledge) to identify the responsible client(s).

Token-Level Attribution for Autoregressive LLMs. Extending these provenance principles to federated LLMs introduces unique challenges. LLMs generate variable-length sequences through autoregressive token generation, where each token depends on previously generated tokens, creating cascading attribution dependencies. Furthermore, distinguishing client contributions from the LLM’s pre-existing knowledge and achieving computational tractability with billions of parameters require targeted solutions. We address these through three mechanisms. First, we compute per-token provenance scores that attribute each generated token to clients, then aggregate across the sequence for response-level attribution. Second, we restrict provenance to strategically selected layers, specifically the self-attention output projections and final feed-forward layers in the last transformer blocks, where task-specific signals concentrate. Third, we use activation-gradient products to automatically weight each client’s contribution by relevance for each specific token being generated.

Summary. *We materialize the aforementioned foundational insights into three novel debugging and interpretability techniques for FL. FedDebug [73] tackles fault localization, TraceFL [75] enables client attribution for classification models, and ProToken achieves token-level attribution for federated LLMs. Together, they validate the central hypothesis that privacy-preserving debugging and interpretability in FL is achievable without violating FL principles. The following sections provide an overview of each technique.*

1.3.1 Systematic Debugging for FL Applications

A faulty client can send an inaccurate model to the aggregator either due to noisy labels [85, 114, 115] in the training data or malicious intent to deteriorate the global model’s performance [27, 35, 37, 146]. Finding such a faulty client is challenging due to FL complexities (*e.g.*, numerous rounds, clients, diverse data per client). None of the existing FL frameworks provide debugging and testing support to developers when building FL applications [97]. Debugging in FL is challenging due to unpredictable client participation and the transient nature of their contributions to the global model. Traditional debugging approaches, like breakpoints, disrupt the entire process and risk data loss, as clients often lack persistent storage.

We design a systematic fault localization framework, FedDebug [73], that advances the FL debugging on two novel fronts. First, FedDebug enables interactive debugging of realtime collaborative training in FL by leveraging record and replay techniques to construct a simulation that mirrors live FL. FedDebug’s breakpoint can help inspect an FL state (round, client, and global model) and move between rounds and clients’ models seamlessly, enabling a fine-grained step-by-step inspection. Second, FedDebug automatically identifies the client(s) responsible for lowering the global model’s performance without any testing data and labels, both of which are essential for existing debugging techniques. FedDebug’s strengths come from adapting differential testing in conjunction with neuron activations to determine the client(s) deviating from normal behavior.

In our experiments, FedDebug achieves 100% accuracy in finding a single faulty client and 90.3% accuracy in finding multiple faulty clients. FedDebug’s interactive debugging incurs 1.2% overhead during training, while it localizes a faulty client in only 2.1% of a round’s training time. With FedDebug, we bring effective debugging practices to federated learning,

improving the quality and productivity of FL application developers.

1.3.2 Interpretability in FL via Neuron Provenance

FL developers face significant challenges in attributing global model predictions to specific clients. Localizing responsible clients is a crucial step towards (a) excluding clients primarily responsible for incorrect predictions and (b) encouraging clients who contributed high-quality models to continue participating in the future.

In FL, the client(s) most responsible for a global model’s prediction are the ones trained on data that contains the predicted labels. This is analogous to finding influential training samples in classical machine learning [104]. However, the two domains, single model-based centralized ML and FL, are fundamentally different. Existing influence-based debugging approaches in ML [31, 32, 161, 174] and regular software [18, 21, 148] require transparent access to data including all data manipulation operations applied on the input data. When applied to FL, these approaches will require end-to-end monitoring of clients’ training (*i.e.*, require access to clients’ data), which is prohibited in FL.

We introduce TraceFL [75], a fine-grained *neuron provenance* capturing mechanism that identifies clients responsible for a global model’s prediction by tracking the flow of information from individual clients to the global model. Since inference on different inputs activates a different set of neurons of the global model, TraceFL dynamically quantifies the significance of the global model’s neurons in a given prediction, identifying the most crucial neurons in the global model. It then maps them to the corresponding neurons in every participating client to determine each client’s contribution, ultimately localizing the responsible client.

We evaluate TraceFL on six datasets, including two real-world medical imaging datasets and four neural networks, including advanced models such as GPT. TraceFL achieves 99%

accuracy in localizing the responsible client in FL tasks spanning both image and text classification tasks.

1.3.3 Token-Level Attribution for Federated LLMs

While TraceFL addresses client attribution for classification tasks in FL, recent advances in federated learning have extended to LLMs [107, 177, 198], introducing new attribution challenges. When a federated LLM generates a response, organizations need to understand which clients influenced that specific output for attribution, debugging, and trust verification. However, unlike classification models that produce single predictions, LLMs generate variable-length sequences through autoregressive token generation, where each token depends on previously generated tokens. This creates unique provenance challenges: distinguishing client contributions from the LLM’s pre-existing knowledge, handling cascading dependencies between tokens, and maintaining computational tractability with billions of parameters.

We present ProToken, the first token-level provenance attribution method for federated LLMs. ProToken exploits several key insights: FL aggregation is linear at the parameter level, allowing decomposition of global model computations into per-client contributions; enabling focused attribution on self-attention and feed-forward layers; and per-token activation-gradient attribution automatically weights each client’s contribution by relevance for the specific token being generated. All computations operate on model updates, activations, and gradients (not raw client data) preserving FL privacy constraints. To evaluate ProToken, we develop a novel evaluation methodology using backdoor injection techniques to create verifiable ground truth, overcoming the fundamental ambiguity of distinguishing federated training contributions from pre-existing LLM knowledge.

We evaluate ProToken across four state-of-the-art LLM architectures (Gemma, SmolLM2,

Llama, and Qwen) on four domain-specific datasets spanning medical, financial, mathematical reasoning, and coding domains. ProToken achieves 98.62% average attribution accuracy across 16 configurations and maintains >92% accuracy at scale with 55 clients. These results establish ProToken as a foundational step toward interpretable, trustworthy, and accountable federated LLM systems.

1.3.4 FL-Assisted User-Centric Semantic Cache for LLMs

While developing novel debugging and interpretability techniques for FL, we discovered the potential to reduce the computational costs of Large Language Models (LLMs) by creating a user-centric semantic cache. This approach leverages FL to optimize the embedding model, enabling efficient query handling.

Large Language Models (LLMs) like ChatGPT and Llama have revolutionized natural language processing and search engine dynamics. However, these models incur exceptionally high computational costs. For instance, GPT-3 consists of 175 billion parameters, where inference demands billions of floating-point operations. Caching is a natural solution to reduce LLM inference costs on repeated queries. However, existing caching methods are incapable of finding semantic similarities among LLM queries nor do they operate on contextual queries, leading to unacceptable *false* hit-and-miss rates.

We introduce MeanCache [76], a user-centric semantic cache for LLM-based services that identifies semantically similar queries to determine cache hit or miss. Using MeanCache, the response to a user’s semantically similar query can be retrieved from a local cache rather than re-querying the LLM, thus reducing costs, service provider load, and environmental impact. MeanCache leverages federated learning to collaboratively train a query similarity model without violating user privacy. By placing a local cache in each user’s device and using

FL, MeanCache reduces the latency and costs and enhances model performance, resulting in lower false hit rates. MeanCache also encodes context chains for every cached query, offering a simple yet highly effective mechanism to discern contextual query responses from standalone.

Our experiments benchmarked against the state-of-the-art caching method, reveal that MeanCache attains an approximately 17% higher F-score and a 20% increase in precision during semantic cache hit-and-miss decisions while performing even better on contextual queries. It also reduces the storage requirement by 83% and accelerates semantic cache hit-and-miss decisions by 11%.

1.4 Contributions

Our contributions are as follows:

- To the best of our knowledge, FedDebug [73] is the first general-purpose debugging framework for FL applications. FedDebug’s novelty lies in observations about FL and the exploitation of insights on reproducibility, inference guided test generation, and differential testing that do not impede performance or violate FL privacy principles. It addresses the open debugging challenges in FL [97]. FedDebug’s artifact is available in Flower [34] (a widely used FL framework) at <https://flower.ai/docs/baselines/feddebug.html>.
- We present TraceFL [75], the first **neuron-provenance**-based interpretability approach for FL. TraceFL dynamically captures and ranks clients’ contributions to specific predictions, making it highly effective for large models like GPT with millions of parameters. It is uniquely compatible with both transformers and CNNs, advancing

debugging and interpretability in FL while supporting differential privacy (*under the given DP settings [135]*) and real-world data distributions. TraceFL’s implementation is available at <https://github.com/SEED-VT/TraceFL>.

- We introduce ProToken, the first token-level provenance attribution method for federated LLMs. ProToken addresses the unique challenges of LLMs including autoregressive generation, computational tractability with billions of parameters, and distinguishing client contributions from pre-trained knowledge. Through per-token activation-gradient analysis, ProToken achieves 98.62% average attribution accuracy across four state-of-the-art LLM architectures (Gemma, SmolLM2, Llama, Qwen) and four domain-specific datasets. ProToken establishes a foundation for interpretable and trustworthy federated LLM systems.
- We introduce, MeanCache [76], a unique user-centric semantic cache for LLM based web services, such as ChatGPT. In MeanCache, clients train the global embedding model using federated learning on their local data, while preserving user privacy. The global model formed after the aggregation generates high-quality embeddings for semantic matching. Despite compressing embeddings and saving 83% of storage space, MeanCache outperforms the existing baseline semantic cache (GPTCache). With its distribution cache design, MeanCache offers pathways to reduce one-third of the LLM query inference costs related to semantically similar queries.

Chapter 2

Background and Literature Review

2.1 Federated Learning

Federated Learning enables multiple clients (*e.g.*, mobile devices, organizations) to train a shared model without sharing their data. This allows the model to be trained using distributed data, which can be useful in cases where data is distributed across multiple devices or organizations and cannot be easily collected and centralized. Figure 1.1 shows an FL setting where multiple hospitals collaboratively train a global model on their local labeled medical imaging data. During this collaborative training, clients' training data never leaves their premises [97].

1. In the first step, the aggregator sends copies of the current global model, *i.e.*, the global model weights, and hyperparameters (*e.g.*, learning rate and epochs) to participating clients (Step 1 of Figure 1.1).
2. Using the global model as initial parameters, each client trains a model on its local data similar to traditional ML training (Step 2 of Figure 1.1).
3. Once trained, each client sends its local model, in the form of updated weights, back to the aggregator as shown in Step 3 of Figure 1.1. Additionally, clients share performance metrics such as training loss and quality/quantity of training data with the central aggregator.

4. After receiving model updates, the server aggregates the updated weights from all clients using established model aggregations (also called fusion) techniques such as FedAvg [134] to form a new global model (Step 4).

These four steps are repeated for a fixed number of *rounds* or until the global model meets some convergence criteria, for example, when training loss is close to zero. One algorithm of FL is Federated Averaging (FedAvg) [134], which uses the following equation to update the global model at each round of the training process:

$$W_{global}^{t+1} = \sum_{k=1}^K \frac{n_k}{n} W_k^{(t)} \quad (2.1)$$

where $W_k^{(t)}$ and n_k represent received weights and size of training data of client k in each round t , respectively. The variable n represents the total number of data points from all clients, and it is calculated as $n = \sum_{k=1}^K n_k$. The equation states that the global model W^{t+1} at the next round is the average of the local models from all participating clients at the current round. In each round, the clients first train their local models using their own data, then send the parameters (*e.g.*, $W_k^{(t)}$, n_k) to the central server. The central server averages the model parameters to produce a global model, which is then sent back to the participating devices. This process is repeated for multiple rounds (*e.g.*, t from 1 to 100), with each client updating its local model using the global model from the previous round. The final global model is the result of the federated averaging process.

FL has variations such as Vertical FL [122] and Personalized FL [179]. TraceFL primarily focuses on horizontal FL [134], similar to previous work on fault localization in FL [73, 74]. A typical FL setup involves a few to thousands of clients, such as mobile devices, healthcare institutions, or enterprises. FL clients exhibit diversity in data distribution and computational resources. Data is often non-independent and non-identically distributed

(Non-IID), with varying sizes across clients, such as hospitals specializing in different medical conditions (Figure 4.1). All participating clients use the same neural network architecture, ensuring compatibility during model aggregation. Additionally, clients have heterogeneous hardware and network capabilities (e.g., smartphones to powerful servers), impacting their participation and training consistency [103].

2.2 Debugging and Interpretability in ML

Debugging machine learning (ML) models has been extensively explored in the recent research [42, 78, 143, 151, 188, 195, 201]. The primary objectives of these approaches are interpretability, generating new test cases by carefully perturbing the real-world training inputs to improve performance and to find bugs and corner cases in the given model. These approaches require access to the training and testing data, and some are limited to testing a single neural network; hence, such approaches cannot be directly imported into FL. Lack of access to client data and resources in FL settings makes testing and debugging FL more challenging. If applied to FL, these testing approaches would find every client’s model defective. Clients’ models are architecturally similar but trained on local clients’ data, and thus their models are semantically different from each other. Identifying defects in an isolated model is not practical either. Every client’s model has weaknesses that will surface on carefully selected test data. FedDebug overcomes these problems by focusing on the commonality of models instead of differences.

Most relevant work to FedDebug primarily focuses on finding clients’ contributions to a global model without exposing the private data to a central server [217]. In practice, individual clients report information about training, such as dataset size and performance metrics, to the central aggregator [99, 110, 166, 212, 216]. Existing approaches use prior

information *e.g.*, previous task performance and data quality obtained via third-party services, to evaluate clients' models [186]. Other approaches recommend cross-validating clients' models on another client's local dataset [130]. Another alternate is maintaining a validation dataset at the central server to evaluate clients' models [50, 129]. A major limitation of the above FL-related approaches is that the aggregator server depends entirely on the client's reported information or test data to evaluate clients' models. The aggregator also assumes that all clients are trustworthy about their performance in these approaches, which attracts adversarial clients like the ones in targeted poisoning attacks [146]. Cross-validation is also prohibited due to the limited computing resources for edge devices such as smart home sensors. FedDebug overcomes the limitations of debugging faulty clients with interactive and automated approaches that preserve privacy.

Note that work on robust FL, including trust bootstrapping, backdoor defenses, and resilient aggregation [45, 61, 67, 141], addresses adversarial settings. Our work focuses on trustworthy (non-adversarial) settings; assessing the robustness of FedDebug, TraceFL, and ProToken under adversarial manipulation is an important future work.

2.2.1 Interpretability in Machine Learning

As the complexity of neural network models continues to increase, the need for interpretability techniques becomes more crucial and important. Interpretability techniques are used to understand the inner workings of a neural network. These techniques try to explain the decisions made by the model, and how the model makes these decisions. This is important for many reasons, including the ability to explain which input features are important to a model's output, to understand the model's behavior, and to identify potential biases and errors in a trained model. Several approaches, such as Integrated Gradients [178], Gradient

SHAP [128], DeepLIFT [169], Saliency [171], Guided GradCAM [168], Occlusion (also called sliding window method) [214], and LIME [159], exist which evaluate the contribution of each input feature to model’s output. For instance, Integrated Gradients [178] evaluates the contribution of each input feature by calculating the integral of gradients *w.r.t.* input. This is done along the path from a selected baseline to the given input. Occlusion involves replacing each contiguous rectangular region with a predetermined baseline or reference point and measuring the difference in the model’s output. This approach is based on perturbations and provides a way to evaluate the importance of input features by measuring the change in the model’s output.

Existing debugging techniques [71, 176, 182, 187, 202] are designed to identify issues and enhance the performance of a single neural network in centralized ML. These methods typically require access to training data, which is prohibited in FL. For example, NPC [202] constructs a Decision Graph using training data. Furthermore, these approaches have not been evaluated on modern neural network architectures such as Transformers.

Almost all existing debugging and interpretability approaches are inapplicable in FL, as by design, they solve an orthogonal problem, identifying the important feature in the input responsible for a prediction instead of clients. This distinction is critical because the training data or the training process is completely inaccessible in FL. Existing approaches require access to the client’s data. Furthermore, they are only designed for a single neural network, but the FL global model is a mixture of clients’ models participating in the given round. Operating these techniques on FL would require us to first identify a suspicious client’s model, a problem that TraceFL solves. Even if such techniques are applied to a client’s model, the resulting feedback is not immediately actionable and constructive. TraceFL is designed to address the limitations of the existing debugging approaches and added challenges of FL, such as distributed training, inaccessibility to clients, and the mixture of models.

FedDebug [73] introduces differential testing in FL to identify faulty clients by capturing each client’s activations for a given input and localizing the client(s) whose behavior deviates from others. Building on FedDebug, a backdoor detection technique in FL is presented in [74]. Additionally, FedGT [199] aims to identify malicious clients in FL; however, it is limited to scaling up to 15-30 clients and has not been tested on advanced architectures like GPT.

Despite their contributions, these existing methods target a narrower problem under a specialized setting. FedDebug is limited to image classification tasks using CNNs, restricting its applicability to Transformer architectures. Additionally, it is designed primarily for faulty clients and IID distributions [117], as demonstrated in Table 4.1. In contrast, TraceFL targets a broader debugging problem using a domain-agnostic, and highly accurate client localization mechanism applicable to diverse neural network architectures, data types, and distributions through its novel fine-grained **neuron provenance**.

There has been recent work on ensuring accountability in FL systems. A vast majority of solutions leverage the blockchain to ensure accountability [30, 53, 100, 138, 221]. Some of these works (BlockFLow [138], BlockFLA [53]) design an FL system that uses the Ethereum blockchain to provide accountability and monetary rewards for good client behavior. However, all these systems require utilizing the blockchain and entail significant modifications to the existing FL system, presenting a barrier to adoption. TraceFL, in contrast, can work with any existing system without modifications.

2.2.2 Provenance Approaches in Machine Learning

Provenance has been extensively studied for both ML and dataflow programs [18, 21, 91, 124, 148]. They address various issues such as reproducibility [147, 161, 163, 173, 204], provide debugging and testing granularities [91], explainability [18], and mitigating data poisoning

attacks [31, 32, 174]. In the context of machine learning, provenance tracks the history of datasets, models, and experiments. This information is used to select the interpretability of neural network predictions and reproducibility. Provenance-based approaches are important to create ML systems that generate reproducible results [147, 161, 163, 173, 204]. For instance, Ursprung [161] captures provenance and lineage by integrating with the execution environment and records information from both system and application sources of an ML pipeline. Ursprung does not require changes to the code and only adds a small overhead of up to 4%.

2.3 Existing Caching Approaches

Several caching systems are proposed to optimize the performance. Study [165] suggests a two-tier dynamic caching architecture for web search engines to enhance response times in hierarchical systems. Utilizing LRU eviction at both levels, they demonstrate how the second-tier cache can significantly lower disk traffic and boost throughput. Researchers in [25] propose a three-level index organization and [126] propose a three-tier caching. Another study [203] examined two real search engine datasets to explore query locality, aiming to develop a caching strategy based on this concept. Their analysis centered on query frequency and distribution, assessing the feasibility of caching at various levels, such as server, proxy, and client side. A novel caching technique called Probability Driven Cache (PDC) is proposed in [113] to optimize the performance of search engines by using the probability of query repetition to decide whether to cache the query. PDC uses the probability of a query to be repeated to decide whether to cache the query or not. A different approach is presented in [60], which proposes the Static Dynamic Cache (SDC) to exploit temporal and spatial locality present in the query stream, avoiding redundant processing and saving com-

putational resources. Efficient caching designs for web search engines are explored in [26], where static and dynamic caching strategies are compared, weighing the benefits of caching query results against posting lists. Challenges in large-scale search engines, which process thousands of queries per second across vast document collections, are examined in [220], focusing on index compression, caching optimizations, and evaluating various inverted list compression algorithms alongside caching policies such as LRU and LFU.

All of these studies focus on caching systems designed for traditional search engines that process keyword queries (i.e., keyword matching) and return a list of links as a response. However, when applied to LLM-based web servers or APIs, these caching systems do not provide a single concise response and may yield many false results. Moreover, such caching techniques fail to capture the semantic similarity among repeated queries, leading to a significantly low hit rate. Server-side caching for services based on LLMs is proposed in [224] and [29], aiming to reduce the massive computational cost of LLMs. In particular, the approach in [224] checks if a new query is semantically similar to any existing queries in the cache. If a match is found, the cached response is returned; otherwise, a model multiplexer selects the most suitable LLM for the query.

While these techniques can handle semantic similarity among queries and provide a single concise response, they raise privacy concerns as user queries are stored on external servers. Additionally, these techniques are static and unable to adapt to individual user behavior. Users may still be charged for the query, even if it is served from the cache. Therefore, a user-centered semantic cache that operates on the user side is needed, providing benefits in terms of privacy, cost, and latency. This cache should be able to detect semantic similarity among queries and adapt to each user’s behavior while preserving privacy. MeanCache offers these benefits without compromising user privacy.

Chapter 3

Systematic Debugging for FL Applications

3.1 Introduction

Many machine learning models today require private user information for high-quality training. However, users are naturally reluctant to share such data to minimize the risk of privacy violation. To address the above needs, Federated Learning (FL) [134] enables individual participating clients (*e.g.*, smart-home edge devices) to train a machine learning (ML) model on their local data in a privacy-preserving environment and then send the trained model (*e.g.*, the weights of the neural network) to a central aggregator to build a global model. FL trains highly accurate models without ever accessing user data, keeping clients' data privacy intact [97]. With the advent of frameworks like Flower FL [34] and IBMFL [127], FL is actively used in solving real-world problems [92, 125, 160, 223].

Problems. The support for collaborative yet privacy-preserving training in FL comes at the cost of transparency and comprehension, making debugging prohibitively complicated. For instance, a faulty client can send an inaccurate model to the aggregator either due to noisy labels [85, 114, 115] in the training data or malicious intent to deteriorate the global model's performance [27, 35, 37, 146]. Finding such a faulty client is challenging due to the large number of unpredictable clients that may not have participated in every round because

of a poor network connection or low battery power [181, 207]. The FL training process also spans numerous rounds, significantly increasing the search space for identifying the true culprit round. None of the existing FL frameworks provide debugging and testing support to developers when building FL applications [97]. These developers rely on guesswork and expensive trial-and-error debugging to find a fault-inducing client.

Challenges. FL poses two fundamental challenges when designing a debugging technique. First, in FL deployments, training and testing data are kept private and strictly reside with clients. Access to such data could allow developers to evaluate individual clients’ models sent to the aggregator and identify the lowest-performing model as the culprit, similar to traditional ML model testing. Neither test data nor labels are available to an FL application developer and, therefore, existing ML debugging approaches [143, 151, 195] are inapplicable. Second, due to the unpredictability of clients’ participation in a round and the ephemeral nature of their contributions in the global model, reproducing a fault (*i.e.*, faulty client) and then debugging it is not feasible. Traditional breakpoint debugging will pause the entire training process in FL across all clients, causing severe side effects such as data loss as clients may not have persistent storage to store data. Live postmortem or trial-error debugging may lead to a new set of clients for each round based on client availability and quorum, thus making debugging even more ineffective. Considering the above limitations and challenges, we must design a debugging approach that does not rely on clients’ data, can debug a live FL application without any interference, and can localize a faulty client precisely.

Contributions. We take inspiration from traditional debuggers, such as `gdb`, and redesign traditional debugging constructs that are tailored to the needs of an FL application developer. Our approach, FedDebug, selectively records an FL application’s telemetry data to enable realtime interactive debugging on a simulation that mirrors a live FL application. With FedDebug’s *breakpoint*, a developer can spawn a simulation of a live FL application and

inspect the current state containing information such as clients’ models and their reported metrics (*e.g.*, their training loss or hyperparameters). It also allows a seamless transition between the rounds and clients at a given breakpoint, enabling a fine-grained step-by-step inspection of the application’s state. When a developer finds a suspicious state (*e.g.*, multiple clients report high training loss), FedDebug’s automated fault localization approach precisely identifies the faulty client(s) without any test data or labels. Once a faulty client is identified, FedDebug’s *fix and replay* repairs the global training by retroactively removing the faulty client and resuming the live FL training.

Key Insights. FedDebug leverages several insights to enable systematic FL debugging while preserving clients’ privacy. We observe that instead of debugging a live FL application, we can record a set of runtime metrics essential to regenerate a given state in an FL application. Thus, FedDebug performs debugging on a regenerated simulated state equivalent to a live state. To have a measurable impact on the global model, a faulty client’s model must behave differently than the regular clients. Every client in an FL application has the same model architecture, so their internal behaviors are comparable. Based on this insight, FedDebug proposes an inference-guided test selection method to select high-quality and diverse test data from a pool of randomly generated input images using Kaiming Initialization [81]. However, an auto-generated data does not include the class label *i.e.*, an oracle. To address the *oracle* problem with such data, FedDebug adapts differential testing to FL domain. It captures differences in the models’ execution via neuron activations instead of output labels to identify *diverging* behavior of faulty clients.

Evaluations. We perform large-scale, extensive evaluation of FedDebug on popular models, two large-scale datasets, two well-established FL data distributions, and a real-world fault-injection technique in a total of 68 different FL configurations. We measure FedDebug’s fault localizability, debugging time, performance overhead over a vanilla FL framework

(IBMFL), and scalability. FedDebug shows remarkable success in identifying faulty clients. It localizes the real-world faulty client with 100% accuracy within 2.1% of a round’s training time. When faced with multiple faulty clients, FedDebug retains the high fault localization accuracy of 90.3%. FedDebug’s debugging constructs incur an overhead of 48% of the aggregation time to record telemetry data for state regeneration. Surprisingly, this time is only 1.2% of a single round’s training time in our experiments. Through our evaluation, we demonstrate that FedDebug effectively conducts interactive debugging and efficiently automates fault localization without incurring high runtime costs. FedDebug augments the IBMFL framework, but its underlying insights can be adapted for other FL frameworks.

We summarize FedDebug’s contributions below:

- **Originality:** To the best of our knowledge, FedDebug is the first general-purpose debugging framework for federated learning applications that is not limited by access to clients’ data. It addresses the open debugging challenges in FL [97].
- **Approach:** Traditional ML trains a single model, whereas FL involves distributed training across hundreds of clients over multiple rounds. Thus, existing ML debugging approaches are inapplicable on FL. FedDebug’s novelty lies in observations about FL and the exploitation of insights on reproducibility, inference guided test generation, and differential testing that do not impede performance or violate FL privacy principles.
- **Benchmark:** We evaluate FedDebug in 68 FL configurations derived from well-established datasets, models, varying clients, data distribution, and fault-injections. We package our experiment environment into a public benchmark for future research use.
- **Usefulness:** Our extensive experiments demonstrate that FedDebug successfully locates faulty client(s) without impeding the FL workflow. On a wide range of exper-

iments, FedDebug exhibits robust results against multiple faulty clients, challenging data distributions, and a large number of clients. FedDebug’s artifact and the benchmarks used in this work are publicly available at <https://doi.org/10.5281/zenodo.7578656>.

3.2 Motivation

Suppose that an FL application developer trains a global neural network model, ResNet [82], on chest X-ray images from hospitals across the country to diagnose respiratory diseases (e.g., Covid-19). The term *developer* refers to a person who writes, deploys, and monitors the FL application at the central server, as shown in Figure 1.1. Every participating hospital collects X-rays of patients labeled by radiologists and trains a local ResNet model on that data. Hospitals periodically share their locally trained models with a central server. The central server then aggregates these shared models into one global model. After aggregation, the central server sends the updated global model to each hospital to incorporate in local training in the next round, as shown in Figure 1.1.

The developer observes that multiple hospitals are reporting a high training loss from their preceding training rounds. One plausible reason is that one of the hospitals is performing training on noisy data (e.g., mislabeled by inexperienced staff [50, 114]) and continuously impacted the global model during aggregation. Thus, when the global model is shared back with the other hospitals, it influences their training.

Challenges of FL Debugging. After noticing an increase in training loss, the developer must investigate the root cause, as misdiagnosis from medical imaging can lead to ill treatment. To debug the FL application at this scale, the developer begins by manually inspecting various collected logs at the central server, including the global model weights,

shared local models from hospitals, and the response and training time of each hospital. Due to patient privacy, the hospitals refrain from sharing their labeled training data, which is critical for correctly evaluating the quality of a model and thus essential for localizing the faulty round and model. Even if the developer finds the problematic round, she cannot isolate the hospital(s) responsible for affecting the global model without test data. One option is cross-validating each client’s model by requesting that the other clients test the model on their local data. This is prohibited in practice, as it adds computational burden on clients (*e.g.*, edge devices) and can potentially cause data privacy violation. Lastly, statically inspecting hospitals’ models does not provide any meaningful information. Without any debugging techniques at her disposal, she resorts to using guesswork to identify the hospital with noisy labels.

FedDebug’s Contributions. The developer decides to use FedDebug to investigate the root cause behind high training loss. When enabled, FedDebug allows a developer to set a *breakpoint* at any round or even in the first round to capture the end-to-end training logs. This breakpoint separately invokes a debugging session, a simulation of the original FL service, without stopping the live training. In the debugging session, the developer uses FedDebug’s *step-back* and *step-next* constructs to move between rounds, inspecting the global and local models of hospitals. Upon inspecting the training rounds, she finds the specific round, *e.g.*, **round 8**, where the performance starts to deteriorate. This round can be different from the *breakpoint* enabled round, as performance issues can manifest in earlier rounds but surface later. During this inspection, FedDebug also reports the list of hospitals that participated in that round. Next, she invokes FedDebug’s fault localization algorithm to precisely identify the hospital responsible for deteriorating the global model performance. After finding the hospital with noisy labels, the developer removes it from the problematic round (*i.e.*, **round 8**) and onwards. FedDebug’s *fix and replay* starts retraining from **round**

8 to the current round and then replaces the impacted global model with the retrained global model and switches back to the original FL training.

3.3 Design of FedDebug

The goal of FedDebug is to facilitate an FL application developer in isolating a faulty client responsible for deteriorating the global model performance. Recent studies emphasize the need for debugging techniques in FL applications and the challenges associated with providing debugging support in FL frameworks [97]. To this end, we must overcome the following *major challenges* in designing FedDebug. First, the privacy concerns of FL put restrictions on any client-side interference. Second, the unpredictable and ephemeral nature of clients in FL poses a threat to reproducibility, which is critical for debugging a live system. Third, the distributed nature of FL with hundreds of participating clients makes traditional breakpoint debugging ineffective. Pausing the entire FL application at this scale will be prohibitively expensive. Therefore, traditional debugging approaches, such as `gdb`, are not suitable for the scale and architecture of FL systems.

In FedDebug, we address the above challenges and advance systematic FL application debugging. We enable realtime, interactive debugging on a simulation of the live FL application. To do so, FedDebug continuously collects and stores concise telemetry data from a live FL application. Whenever a debugging need arises, the developer can interact with FedDebug’s debugging interface, which uses the telemetry data, to regenerate an FL application’s state.

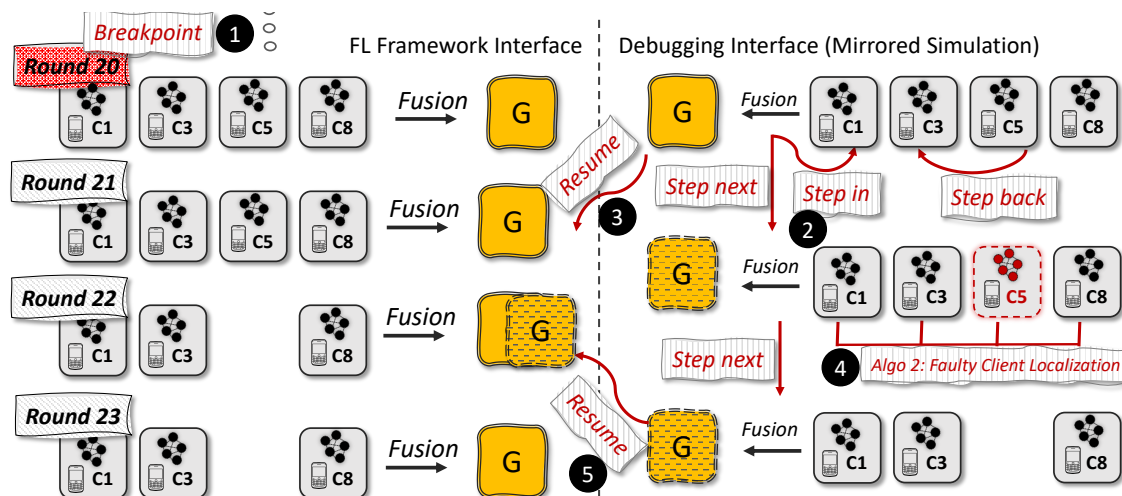


Figure 3.1: Using FedDebug, a developer can set a *breakpoint* at round 20. When the FL application finishes round 20, FedDebug launches a Debugging Interface, reflected on the right. *Step next* (2) takes the developer to the next step (round or client). *Step-in* increases the granularity of computation, *e.g.*, round to client level. *Resume* (3) rejoins the current execution status of the FL application if no intrusive actions are taken. At a given round, FedDebug can automatically localize the faulty client (4) and then resume (5) upon which the global model will be recomputed without the faulty client. This model will replace the corresponding round's model, and FedDebug will start retraining from that round, round 22, in the FL interface.

3.3.1 Selective Telemetry

FedDebug collects critical FL execution metrics to reproduce an FL application's state for the developer to interact with it while investigating the root cause of a problem. Existing FL frameworks are carefully architected to refrain from revealing private data. As a result, most debugging data is private and cannot be investigated.

FedDebug's debugging approach is inspired by replay debugging. As with any other replay debugging approach, it is essential that FedDebug stores the necessary runtime metrics to reproduce an FL application's state if requested by the developer. We design a highly selective FL event telemetry technique that records the concise execution data available at the central aggregator that is vital for generating any prior FL application state. FedDebug

is different from traditional replay debugging as it only tracks the information needed to recreate an *observable* event and does not log the information unavailable to the developer in a live application. This design reduces the size of continuously growing telemetry data and minimizes the likelihood of information leakage. FedDebug mainly stores the information available after step 3 of Figure 1.1 which is clients' models, their reported metrics such as response time, training loss, validation loss, performance metric (*e.g.*, F1 score), hyperparameters (*e.g.*, learning rates, epochs, weight decay), and round ID. Note that the FL application, including client-side training, will continue uninterrupted in the background with FedDebug's telemetry module continuously collecting execution traces.

3.3.2 Interactive Replay Debugging

To start the interactive debugging process, a developer can invoke FedDebug's debugging constructs that let the developer leverage the telemetry data to investigate the root cause. Breakpoint debugging is the de-facto method of debugging a program. It pauses the program when the execution reaches it. At that point, a developer can inspect the values assigned to different variables, both local and global, and examine the method stack. Such debugging features are not applicable in FL. The traditional breakpoint will pause the distributed training, resulting in unnecessary idling at the client side. Additionally, since the state of a round is not saved, it is currently impossible for the developer to inspect previous rounds. For instance, a developer may want to debug a latent issue that was introduced by a client five rounds ago but surfaced in the current round when the same client participated in training again.

We make the following observation about FL frameworks. An FL application only reveals aggregator's events to a developer. In contrast, events on the client's side are entirely hidden

from the developer except the ones relayed to the aggregator by the client. Building on this observation and the telemetry data captured by FedDebug, our insight is that instead of debugging a system in real-time, we can recreate its observable behavior in a simulated environment, giving an illusion of debugging an FL application in real-time. By doing so, inspections with FedDebug are side-effect free, *i.e.*, FedDebug will not interfere or interrupt the live FL application. Thus, eliminating the need to pause client-side training or halt FL aggregator execution.

Breakpoint. To this end, FedDebug offers *breakpoint* that can help a developer inspect intermediate states of an FL application. FedDebug’s breakpoint operates on computation units of *rounds*. Any abnormality in the client-reported metrics, such as training loss, validation loss, response time, and performance metrics (e.g., F1 score) can necessitate the use of breakpoints. FedDebug allows setting a breakpoint at any arbitrary round during live FL. A developer can also set a breakpoint from the start (*i.e.*, round 0) to capture end-to-end FL training traces or on a specific round (*e.g.*, round 20 in Figure 3.1-❶) to inspect FL training at that round. When the live FL application arrives at a breakpoint, FedDebug spawns a new debugging interface on the aggregator side, as shown in ❶ in Figure 3.1, while continuing the live FL training in the background.

Step in/Step out. While at a breakpoint in a debugging session, a developer can use *step-in* and *step-out* actions to switch between different granularities of computational units. Traditionally, these two actions are used to go one-level deeper in the stack (*e.g.*, inside a function call) and move one level up in the stack (*e.g.*, outside the function call), respectively. Based on this convention, we define a round as a coarse-grained unit of computation that can be decomposed into a subset of clients participating in that round. Suppose the current breakpoint is at round 20. Step-in will take the developer to the clients-level granularity (❷ in Figure 3.1) where trained models from clients are being aggregated, using a

fusion algorithm (*e.g.*, FedAvg [134]). Step-out will take the developer back to the level of rounds, allowing them to inspect the global trained model at a higher level of abstraction and understand its performance across multiple rounds. Inspecting a state at client-level granularity entails evaluating the performance of a partially-aggregated global model. For example, in Figure 3.1, step-in at ❷ will take the execution between C1 and C3, where the global model has yet to incorporate the local models of clients C5 and C8.

Step Next/Step Back. Similar to step-in/out, *step next* and *step back* help a developer transition from one state to another. For instance, if the breakpoint is at round 20, step next will take the execution to round 21 in the debugging interface, showing information corresponding to that round only. Similarly, if the breakpoint is at client C5, step back will take the execution state to a partial global model after aggregating models from clients C1 and C3 only (Step back in Figure 3.1).

Resume. Unlike resume in `gdb`, FedDebug’s *resume* does not resume any paused execution. Instead, resume gives the illusion to the developer that execution is being continued from where it left off. FedDebug creates this environment by replaying the telemetry data that was captured while the FL application was being inspected using breakpoints, in case the developer does not find any faults in the round under inspection. Once the sequence of events in telemetry catches up with the live execution of the FL application, FedDebug switches to the FL interface and shuts down the debugging interface. This three-step process is nearly indistinguishable from an FL application with FedDebug disabled, giving the impression of debugging a real-time FL application interactively. *Resume* is also illustrated in Figure 3.1

- ❸.

3.3.3 Fix and Replay

When the developer successfully identifies a faulty client in any round, FedDebug offers *Fix and Replay* to allow a developer to roll back the training and provide a retrained global model (the one without a faulty client). We describe the technique to identify a faulty client in Section 3.4. A faulty client may have a compound effect on the global model, as it may have begun to share its noisy model updates latently several rounds ago, which only later becomes noticeable. In such cases, it is important to rectify the impact of a faulty client's inclusion in prior training rounds by removing its contributions. This requires retraining over multiple rounds, which is not possible as clients may not store the data used in training in the prior rounds. Figure 3.1-④ shows the removal of a faulty client (C5) in round 21. FedDebug recomputes the global model in the debugging interface and then replaces the actual global model in round 22 with the newly recomputed global model after fix and replay (Figure 3.1-⑤). By default, FedDebug forbids the faulty client from participating in the FL training. However, it is up to the developer to weigh the benefits of including the faulty client in future rounds.

3.4 Faulty Client Localization

Faults in a client's model can arise due to measurement errors, human labeling errors, data poisoning, communication problems, or subjective biases of labellers. For a high-quality global model, it is critical to correctly identify a faulty client and potentially restrict its participation. Manually identifying faulty clients is neither scalable nor effective due to a large number of participating clients in FL and their uninterpretable models. Furthermore, the model parameters (*i.e.*, weights) do not provide any meaningful debugging information. To automate faulty client localization, we must define a feedback mechanism to guide our

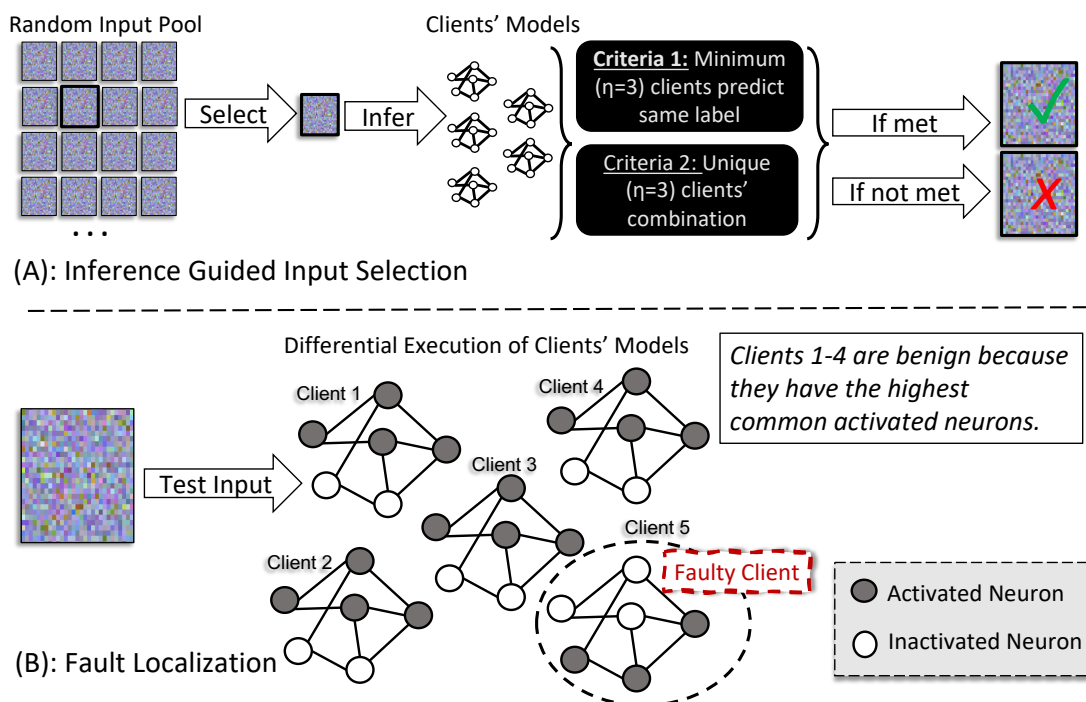


Figure 3.2: An overview of FedDebug’s fault localization approach. It first selects a random input that invokes diverse model behavior (A). It then applies differential execution on clients’ models to localize a faulty client (B).

search for faulty clients efficiently. Automated debugging tools [111, 215] for regular software address this problem by relying on multiple test *inputs* and a test *oracle*. For example, unit tests can guide the search toward concise input leading to incorrect program output [215]. In FL, the inputs and oracle translate into diverse test data and the corresponding accurate labels, both of which are unavailable to the developer at the central server.

FedDebug addresses the challenges of automated fault localization with a two-pronged approach. First, it generates a pool of random test inputs and applies a novel inference-guided test input selection to construct a suite of test inputs, as shown in Figure 3.2-A. Since the test inputs are generated autonomously and are not accompanied by ground truth labels, metrics such as F1 score or accuracy cannot be used as oracle feedback to identify a faulty client. Instead, FedDebug performs differential testing of clients’ models to measure similar-

ities and differences among models’ behaviors on selected inputs (Figure 3.2-B). FedDebug fingerprints a neural network behavior on an input by profiling the internal neurons’ contributions towards a model prediction. Subsequently, FedDebug accurately recognizes a client as faulty if its behavior deviates from the norm, which is the majority of the clients’ behavior. Our insight is that a faulty client’s model will show a noticeable difference in its internal neuron values compared to benign clients’ models, based on the principle that faulty executions are intrinsically different from correct ones. The same principle is behind popular fault localization techniques, such as spectra-based fault localization [95] and delta debugging [215].

Algorithm 1: Inference-Guided Test Input Selection

Input: *shape*: dimension of the random input to be generated.

Input: κ : number of inputs to be generated.

Input: η : minimum number of clients for the same prediction.

Output: \bar{X} : a list containing auto-generated test inputs.

```

1 rand_inputs = lazilyGenerateRandInputs(shape)
2  $\bar{X}$  = list() // a list for inference guided test inputs
3 seen_clients_sequences = list()
4 while length( $\bar{X}$ ) <  $\kappa$  do
5   r_input = pop(rand_inputs)
6   clients_preds = getPredictions(clients, r_input)
7   for label  $\in$  class_labels do
8     clients_seq = samePredClients(clients_preds, label)
9     if clients_seq  $\notin$  seen_clients_sequences and length(clients_seq)  $\geq$   $\eta$  then
10      seen_sequences.append(clients_seq)
11       $\bar{X.append}$ (r_input) // valid test input
12      break
13   if length(rand_inputs) < 1 then
14     rand_inputs = lazilyGenerateRandInputs(shape)
15 return  $\bar{X}$ 

```

Inference-Guided Test Input Selection. As shown in Figure 3.2-A, FedDebug first lazily generates a pool of random test inputs using Kaiming Initialization [81]. For example,

if the clients' models are trained on 32x32 images within the RGB scale, then FedDebug randomly creates a pool of synthetic inputs with the same size and format (*i.e.*, random images of size 32x32 in RGB scale). It then automatically selects only those inputs that lead to a consensus on predictions among a *unique* subset of clients. FedDebug selects up to κ test inputs (default is $\kappa = 10$) among the pool of 1000 random inputs. The goal is to minimize any overlapping behavior between clients while inferring unique class labels on selected test inputs. This is similar to achieving maximum code coverage in regular software with minimum tests. Algorithm 1 selects a test input (line 5) if at least ($\eta \geq 5$) clients predict the same label and that subset of clients has not been seen in a previously selected input (lines 6-11). On the next random input, if the previously observed subset of clients (*i.e.*, $clients_seq \in seen_clients_sequences$) predict the same class label, we discard this input. If a unique combination of clients predicts an unseen label, we include the input in the test suite. This process is repeated until we collect a user-defined, κ , number of test inputs.

Differential Execution of Clients Models. In the absence of correct labels of generated test inputs, FedDebug adapts differential testing to find behavioral differences and similarities among clients' models, as shown in Figure 3.2-B. FedDebug profiles the contributions of individual neurons during model inference on an input and uses these neurons activations to identify models with common behavior. Note that clients' models in FL are comparable due to having a similar architecture. Algorithm 2 describes the faulty client localization process. For a selected test input, FedDebug exhaustively iterates all possible combinations of potentially non-faulty clients (*i.e.*, $\binom{n}{1}$ combinations). For each combination, Algorithm 2 performs model inference on the test input and captures its neuron profiles. FedDebug aims to find one combination of clients that has the highest overlap in behavior, representing the true $n - 1$ benign clients and consequently isolating the precise faulty client. This is a

lightweight process due to the negligible model inference time and the iterations' linear time ($O(n)$) complexity.

Our insight is that among all possible combinations of clients, only one represents true benign clients' subset. The remaining combinations contain the faulty client with abnormal neuron activations, reducing the model behavior overlap within that set. In summary, at a given ill-performing round in FL, FedDebug takes in all participating clients' models as the only input. It automatically generates test inputs and employs differential testing on clients' models to monitor abnormal behavior to precisely identify a faulty client.

Algorithm 2: Faulty Client Localization using Differential Testing

Input: *clients*: a list of clients participated in the given FL round.

Input: *x*: a random input belongs to \bar{X} .

Input: *na_t*: a threshold to profile neuron activations.

Output: *faulty_client*: the faulty client who has abnormal behavior.

```

1 all_clients_combinations = nChooseK(clients, 1)
2 benign_clients = set()
3 max_common_activations = -1
4 for t_clients ∈ all_clients_combinations do
5   | neuron_ids = ActivatedNeurons(t_clients, x, na_t)
6   | t_clients_common_neurons = intersection(neuron_ids)
7   | temp_n = length(t_clients_common_neurons)
8   | if temp_n > max_common_activations then
9     |   | max_common_activations = temp_n
10    |   | benign_clients = t_clients
11 faulty_client = clients - benign_clients
12 return faulty_client

```

3.5 Evaluation

We evaluate FedDebug on (1) runtime performance overhead, (2) debugging time, (3) fault localizability, and (4) scalability. Our evaluation aims to answer the following research

questions:

- **RQ1.** What impact does FedDebug have on the baseline FL framework’s performance?
- **RQ2.** How accurate is FedDebug in identifying a faulty client?
- **RQ3.** Can FedDebug identify multiple faulty clients?
- **RQ4.** Can FedDebug scale to large number of clients and find a faulty client efficiently?

Datasets, Models, and FL Framework. We evaluate FedDebug on CIFAR-10 and FEMNIST datasets. Both are considered as gold standard to evaluate FL experimental settings [52, 117]. FEMNIST is a modified version of MNIST presented in the FL LEAF Benchmark [44] and the Non-IID Bench [117]. The FEMNIST dataset includes more than 340K training and 40K testing grayscale images, each with a resolution of 28x28 pixels, representing ten distinct class labels. CIFAR-10 contains 50K training 32x32 RGB images that span ten different classes and 10K instances for testing. We adopt popular CNN models, namely ResNet [82], VGG [170], and DenseNet [90]. We set the learning rate between 0.0001 and 0.001, the number of epochs between 10 and 25, the batch size from 512 to 2048, and the weight decay to 0.0001. We realize FedDebug’s design in the IBMFL library [127] due to its ease of use, open documentation, and publicly available codebase. These techniques should be equally applicable to other FL frameworks.

Evaluation Environment Specifications. We run our experiments on an AMD 16-core processor with 128 GB RAM and an NVIDIA Tesla T4 GPU. To measure the performance of FedDebug in terms of runtime and debugging overhead, we simulate IBMFL framework deployment on a MacBook Pro with a Quad-core Intel Core i5 processor and 16 GB RAM.

Federated Learning Experimental Settings. Prior FL literature [44, 117] establishes two data distribution strategies among FL clients: IID (independent and identically dis-

tributed data) and Non-IID (non-independent and identically distributed data). For Non-IID, we use the quantity base imbalance [117] where clients have an unequal quantity of data, and the class distribution is random. In IID, the clients receive the same quantity of data. None of the clients share the same data points in both settings. We simulate FL with a varying number of clients, ranging from 10 to 400 clients, in each FL training round. In practice, even with millions of clients, only a subset (in the order of hundreds) is selected in a round. Therefore, our experiment settings are representative of real-world FL deployments [24, 41, 105, 117, 134, 193].

Fault Injection. Since there is no existing FL benchmark with faulty clients, FedDebug adopts a standard noisy labels approach from prior machine learning literature to inject a faulty client in our experiments [66, 85, 93, 116, 211]. Similar to prior work [72, 115, 132], we arbitrarily add noise by changing training data labels (*e.g.*, changing label “bird” to “cat”). When such a client’s model is merged with the global model, the global model’s performance (*e.g.*, accuracy) deteriorates. We define different strengths of a fault with a *noise rate* that controls the number of labels modified in a faulty client. Noise rate is defined as the ratio between changed labels and original labels (*changed-labels/original-labels*).

Figure 3.3 shows the impact of different noise rates on the global model’s accuracy, with one faulty client and nine benign clients. Low noise rates, ranging from 0.2 to 0.7, barely affect the global model performance. With a 0.7 noise rate, the accuracy is lowered by 4.8% and 5.5% in CIFAR-10 and FEMNIST, respectively. A noise rate of 0.9 incurs a 16.2% and 9.9% reduction in the global model accuracy in both settings. Thus, to have a measurable impact on the global model’s performance, we select a noise rate of one for a faulty client. Still, we perform sensitivity analysis in Section 3.5.2 (Figure 3.6) by measuring the impact of varying noise rates on FedDebug’s fault localizability.

Neuron Activation Threshold. We adopt the method from Harel-Canada et al. [80] to

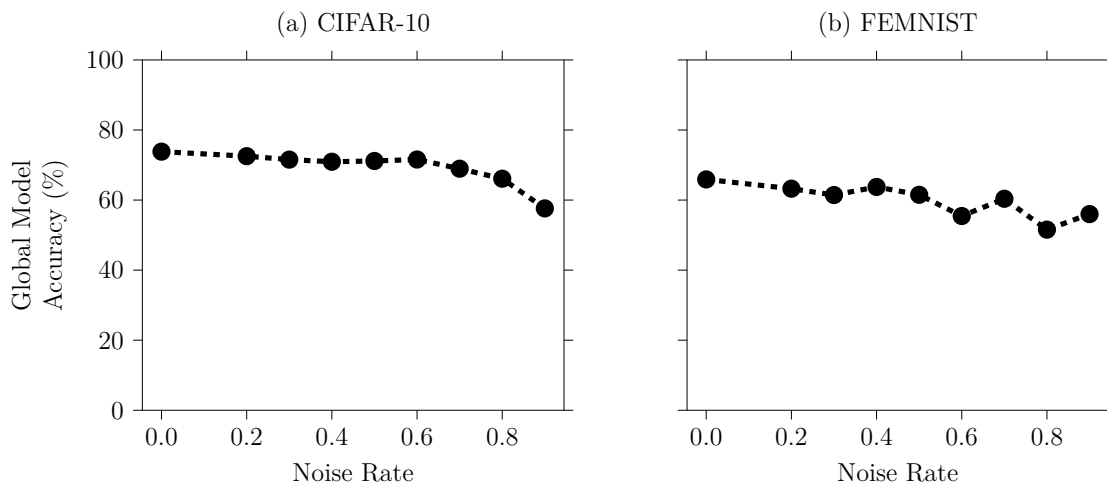


Figure 3.3: Global model (ResNet-34) prediction accuracy in the presence of a faulty client with different noise rates. Lower noise rates hardly degrade global model performance.

profile neuron activations. We empirically find 0.003 as the optimal value for the default activation threshold (see Section 3.5.3). A neuron is considered active when its value crosses this threshold.

Faulty Client Localization Accuracy. We calculate faulty client localization accuracy as the ratio between (a) the number of test inputs on which faulty clients are correctly identified and (b) the total number of test inputs. For instance, if FedDebug identifies the correct set of faulty clients on four out of ten test inputs generated by Algorithm 1, we report 40% fault localization accuracy.

3.5.1 FedDebug’s Performance

Capturing telemetry data in realtime may slow down the performance of an FL application’s aggregator. In this subsection, we present the evaluation results of FedDebug’s runtime overhead and the fault localization time. These experiment settings employ ResNet-18 with CIFAR-10.

Runtime Overhead (RQ1). To evaluate the impact on the FL application’s performance, we measure the slowdown in the running time that FedDebug incurs. We compare the cumulative processing time of the vanilla IBMFL’s aggregator (baseline) against that of the FedDebug-enabled aggregator on a variety of client combinations, ranging from 5 clients to 100 clients. The aggregation time varies with the model’s architecture and the number of clients participating in a round, but it is completely independent of the models’ quality. Therefore, we create up to 100 pre-trained ResNet-18 models and perform the aggregation.

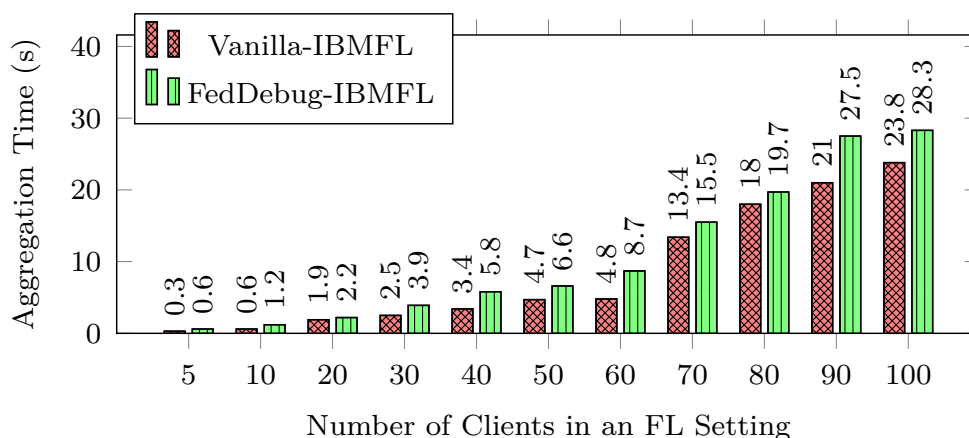


Figure 3.4: FedDebug’s runtime overhead as a comparison between vanilla FL framework’s aggregation time with FedDebug enabled FL aggregation.

Figure 3.4 compares the baseline’s aggregation time with the FedDebug enabled aggregation time. The X-axis represents the number of clients ranging from 5 to 100 clients, and the Y-axis represents the average time across two FL rounds. For instance, with 30 clients, FedDebug takes 3.9 seconds compared to the 2.5 seconds for the baseline to aggregate 30 trained models into a global model. Overall, FedDebug takes approximately 48% additional aggregation time across all experiments. However, in an end-to-end round, the training phase on the clients’ end occupies the majority (up to 97.8% in our experiments) of the round’s time. Compared to the training time of a round, the aggregation time is almost negligible, as low as 1.2% in our experiments.

Takeaway. Considering the training and aggregation time of each FL round, FedDebug’s runtime overhead is a very small fraction, 1.2%, of the training time. Hence, capturing telemetry data for replay debugging does not impede the FL application’s runtime performance.

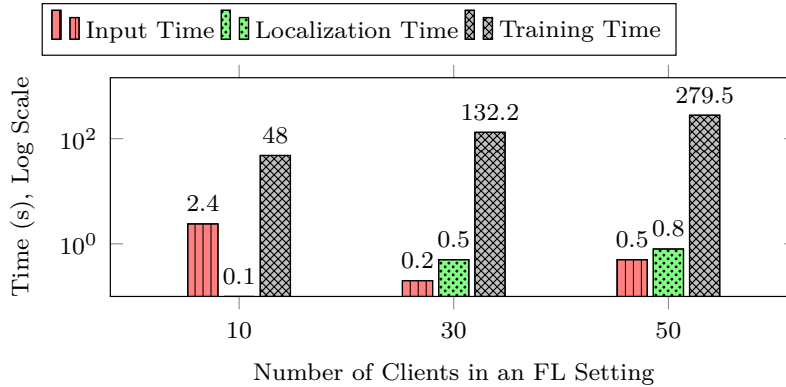


Figure 3.5: FedDebug’s debugging time contains input generation time and faulty client detection time and is compared against a round’s training time.

Debugging Time (RQ1). To assess the localizability of FedDebug, we design experiments to measure FedDebug’s *debugging time*, the time it takes to localize a faulty client. We then compare this time with the training time of that round. Since there is no comparable approach to localize a faulty client, we use training time as a baseline to provide a meaningful scale for the cost of debugging.

Figure 3.5 shows the results of these experiments. The X-axis represents the number of clients, and the Y-axis shows the debugging time in seconds on a logarithmic scale. For 30 clients, FedDebug’s input generation and selection takes 0.2 seconds to find high-quality test input, and its fault localization takes approximately 0.5 seconds to localize a faulty client. In a ten clients setting, input selection takes longer due to constraint $\eta = 4$ for criteria 1 in Figure 3.2. $\eta = 4$ means that at least four previously unseen clients should predict the same label on newly selected test input.

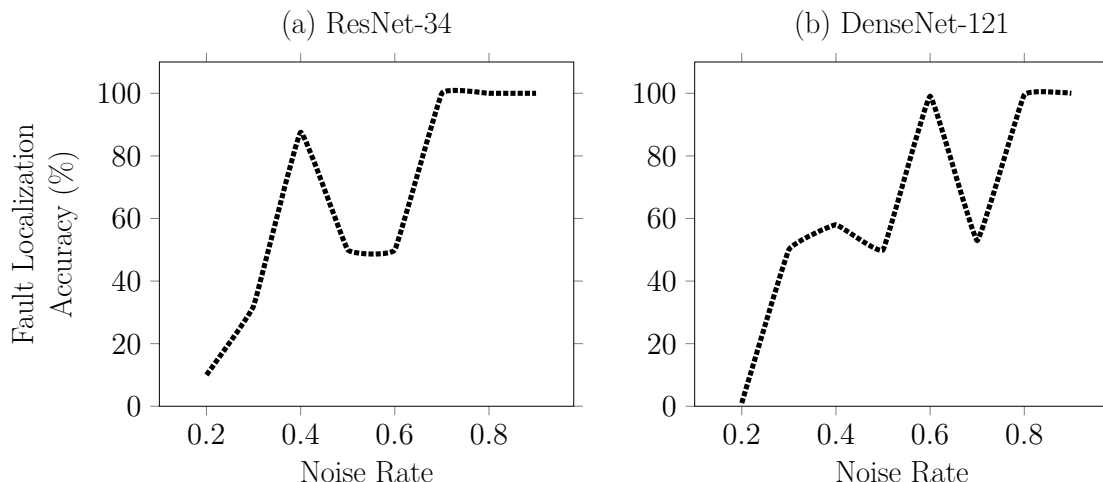


Figure 3.6: FedDebug localization performance when a faulty client has varying fault strength (*i.e.*, low noise rate).

Overall, our results show an increasing debugging time when the number of clients increases, which is expected as increasing the number of clients increases the search space. Note that the debugging time is still in the order of seconds, even for 50 clients. This is because 1) for n clients, the search space has at most n possible combinations of potentially benign $n-1$ clients, representing linear complexity, and 2) on a given input, FedDebug only profiles neuron activations once while iterating over the n combinations.

Takeaway. On average, FedDebug can efficiently identify a faulty client in 2.1% of the total training time of a round.

3.5.2 Localization of Faulty Client(s)

To answer RQ2, we measure how accurate FedDebug is in localizing a faulty client. We inject a faulty client that is representative of a real-world scenario and can cause a measurable change in the global model’s performance. By varying the number of clients, datasets, models, and data distributions (IID and Non-IID), we create 36 different FL configurations

Table 3.1: FedDebug’s debugging time and accuracy when localizing a faulty client in 36 different FL settings with 100 test inputs.

Clients	Dataset	Architecture	Accuracy % (IID)	Accuracy % (Non-IID)	Avg. Input Time (s)	Avg. Localization Time (s)
10	CIFAR10	DenseNet-121	100	100	2.41	0.44
10	CIFAR10	ResNet-50	100	100	2.40	0.22
10	CIFAR10	VGG-16	100	100	2.40	0.21
30	CIFAR10	DenseNet-121	100	100	2.42	1.29
30	CIFAR10	ResNet-50	100	100	1.18	0.70
30	CIFAR10	VGG-16	100	100	2.41	0.47
50	CIFAR10	DenseNet-121	100	100	2.42	3.26
50	CIFAR10	ResNet-50	100	100	1.37	1.24
50	CIFAR10	VGG-16	100	100	2.43	0.91
10	FEMNIST	DenseNet-121	100	100	2.40	0.47
10	FEMNIST	ResNet-50	100	100	2.40	0.25
10	FEMNIST	VGG-16	100	100	2.40	0.18
30	FEMNIST	DenseNet-121	100	100	2.41	1.37
30	FEMNIST	ResNet-50	100	100	0.91	0.68
30	FEMNIST	VGG-16	100	100	2.41	0.55
50	FEMNIST	DenseNet-121	100	100	2.24	2.44
50	FEMNIST	ResNet-50	100	100	1.42	1.24
50	FEMNIST	VGG-16	100	100	2.40	1.25

for FedDebug’s evaluation.

Column 4 and 5 of Table 3.1 show the accuracy of FedDebug in the IID and Non-IID settings, respectively. We repeat each experiment on 100 generated test inputs and take the average of each metric to generalize the results. FedDebug correctly identifies a faulty client with 100% accuracy in both IID and Non-IID settings.

Varying Noise Rate. Figure 3.3 shows the impact of different noise rates on the global model prediction accuracy. We observe that a faulty client has a measurable impact on the global model with a noise rate of > 0.8 . The global model’s accuracy merely drops from 73.8% to 71.1% when the faulty client has a 0.6 noise rate, and drops to 57% when the noise rate is close to one. FedDebug localizes faulty client(s) with low noise rates, showing

Table 3.2: FedDebug’s fault localization in 32 FL configurations with multiple faulty clients, ranging from two to seven.

Faulty Clients	Total Clients	Architecture	Accuracy % (CIFAR-10)	Accuracy % (FEMNIST)
2	30	ResNet-50	100	100
3	30	ResNet-50	100	100
5	30	ResNet-50	100	98
7	30	ResNet-50	100	97.1
2	30	DenseNet-121	100	100
3	30	DenseNet-121	100	100
5	30	DenseNet-121	100	100
7	30	DenseNet-121	100	100
2	50	ResNet-50	50	80
3	50	ResNet-50	66.7	66.7
5	50	ResNet-50	54	60
7	50	ResNet-50	57.1	62.9
2	50	DenseNet-121	100	100
3	50	DenseNet-121	100	100
5	50	DenseNet-121	100	100
7	50	DenseNet-121	100	95.7

its robustness. Figure 3.6 shows the evaluations on varying noise rates in 10 clients FL settings with ResNet and DenseNet architectures. The X-axis shows the faulty client’s noise rate, and the Y-axis represents the average fault localization accuracy on the CIFAR-10 and FEMNIST datasets. The results, as seen in Figure 3.6, indicate that FedDebug has the capability to identify low noise faults—it successfully localizes a faulty client with 0.4 noise rate with approximately 58% and 87.5% accuracy in DenseNet and ResNet settings, respectively.

Takeaway. FedDebug achieves 100% fault localization accuracy on average on a total of 3600 test inputs when the faulty client significantly deteriorates the global model performance in both IID and Non-IID settings. It also accurately localizes a faulty client with low noise rates.

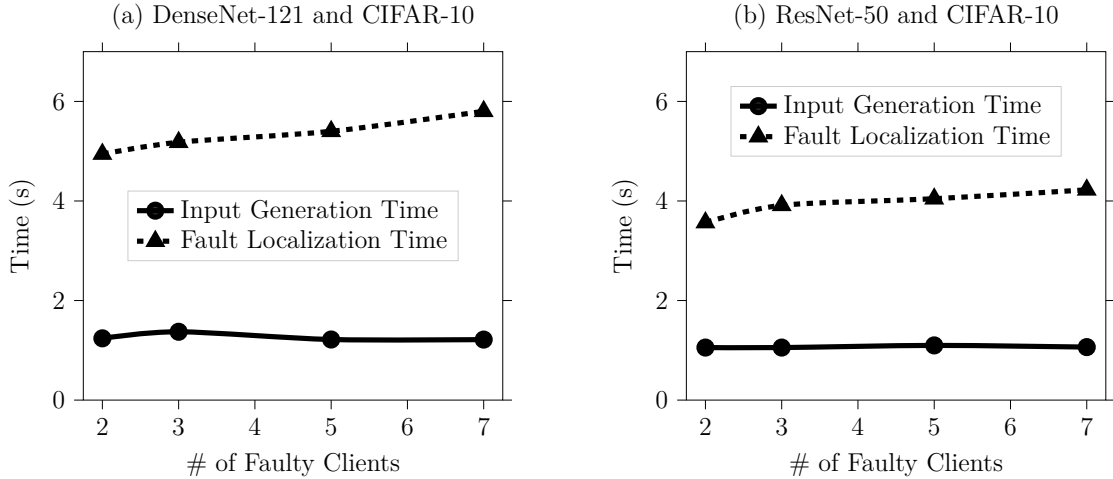


Figure 3.7: FedDebug finds multiple faulty clients in a linear time. Total clients are 50 in each graph.

Detecting Multiple Faulty Clients (RQ3). We evaluate FedDebug’s ability to identify multiple faulty clients. To this end, we inject up to seven faulty clients in the following experiment settings. We train ResNet-50 and DenseNet-121 on the CIFAR-10 and FEMNIST datasets in 30 and 50 clients FL settings. Each setting is evaluated on 10 test inputs. By default, FedDebug’s fault localization technique finds a single faulty client. We apply FedDebug in an iterative manner to find multiple faulty clients by removing one faulty client on each iteration, similar to traditional bug repair process, where one bug is fixed first before the next one is investigated.

Table 3.2 presents the results of finding multiple faulty clients in 32 FL configurations. For instance, when 7 out of 30 clients are faulty and the model is ResNet-50, FedDebug finds all seven faulty clients with 100% accuracy on CIFAR-10 and 97.1% accuracy on FEMNIST. Compared to ResNet, FedDebug performs relatively better with DenseNet. This behavior is expected because, compared to ResNet, DenseNet learns better features due to dense concatenation among its layers, resulting in better performance [219]. Thus, FedDebug performs well in localizing multiple faults with DenseNet with an average accuracy of 99.7%

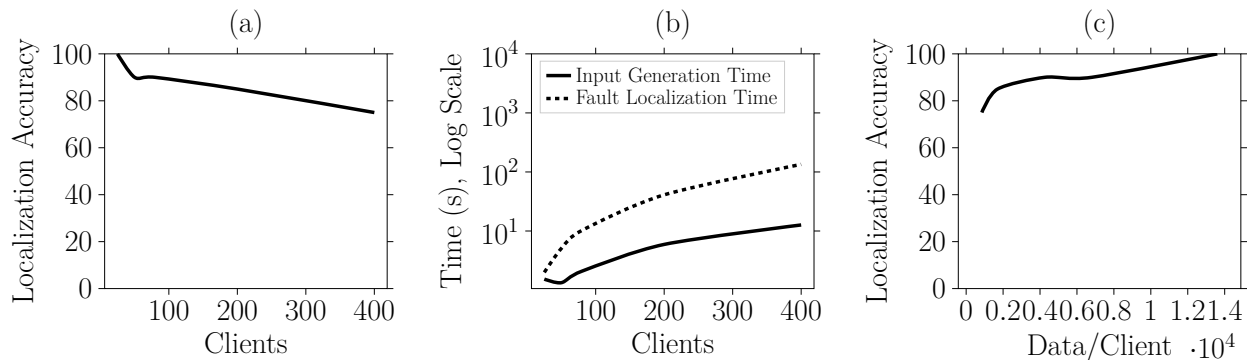


Figure 3.8: FedDebug retains scalability on a large number of clients.

on both datasets compared to ResNet’s 80.8%.

Table 3.2 also reveals that, generally, FedDebug’s localization performance is positively correlated to the number of training data points per client. Large, high-quality training data promotes better feature learning among neurons and, thus, yields better performance. Since the number of data points in FEMNIST (340K) is large compared to CIFAR-10 (40K), clients in the FEMNIST settings have significantly larger training data than clients in the CIFAR-10 settings. As a result, FedDebug average localization accuracy is 78.5% in the ResNet-CIFAR experiment, while it has 83.1% localization accuracy in the ResNet-FEMNIST experiment. FedDebug finds multiple faults with linear time complexity, as shown in Figure 3.7 with 50 clients. The input generation time is almost constant, as the number of clients is fixed. However, the localization time increases as we increase the number of faults from 2 to 7. For instance, it localizes two faulty clients in 3.6 seconds and five faulty clients in 4 seconds.

Scalability (RQ4). Our findings also show that FedDebug scales to larger datasets and an increasing number of clients in FL. Figure 3.8 summarizes the impact on FedDebug’s ability to identify a faulty client when the number of clients changes from 25 to 400 and the training data size per client changes. We perform this experiment with two faulty clients in the FEMNIST-DenseNet configuration. Figure 3.8-(a) verifies that FedDebug’s fault

localization accuracy only reduces to 75% even when the number of clients increases to 400. FedDebug’s debugging time increases linearly as the number of clients increases, consistent with the scale-up properties of general distributed systems, as shown in Figure 3.8-(b). When the number of clients increases, less data is used to train a client’s model, which may reduce the accuracy of clients’ models. Figure 3.8-(c) also shows that FedDebug’s fault localizability also increases when the number of data points per client increases, and it is also robust against low performing client models. For instance, when the number of data points increases from 850 to 1700, FedDebug’s localization accuracy also changes from 75% to 85%, respectively.

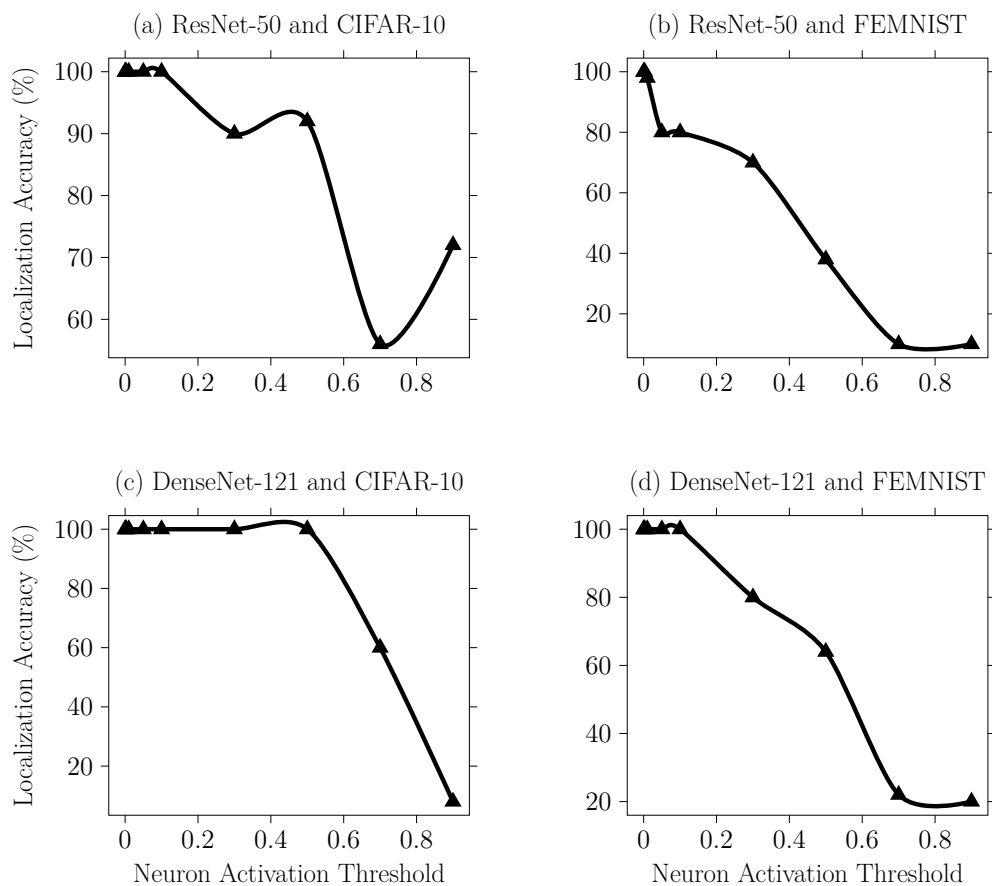


Figure 3.9: FedDebug performance at neuron activation threshold on 30 clients, including five faulty clients.

Takeaway. Our experiment results provide concrete evidence that FedDebug preserves scalability properties both in terms of time overhead and in the presence of multiple faults. It successfully identifies multiple faulty clients in 32 different FL configurations with an average accuracy of 90.3%.

3.5.3 Neuron Activation Threshold

There is no standard threshold of neuron activations [151] and prior work uses experiential value for different use cases [80]. We evaluate the impact of different activation thresholds on FedDebug’s faulty client localizability. We take 30 clients including five faulty clients, and train ResNet-50 and DenseNet-121 on both the CIFAR-10 and FEMNIST datasets. We repeat each experiment on 10 different inputs generated by Algorithm 1.

Figure 3.9 shows the result of these experiments. The X-axis represents the neuron activation thresholds, ranging from 0 to 0.9. The Y-axis shows the FedDebug’s localization accuracy in a given experiment setting. For instance, at the 0.003 threshold, the average localization accuracy across four settings is 100%. On the other hand, at 0.5 threshold, the average accuracy decreases significantly to 73.5% across these configurations. Specifically, for DenseNet-121 and FEMNIST experiment in Figure 3.9-(d), the localization drops to 64% at the 0.5 neuron activation threshold. We observe that FedDebug performs better at lower thresholds (< 0.01) across different models and datasets. This behavior is expected because lower thresholds increase the sensitivity of FedDebug’s localization approach. It starts monitoring most of the neurons’ compared to a higher threshold, where FedDebug profiles only a few neurons crossing the threshold.

3.5.4 Threats to Validity

To alleviate threats to external validity, we use established state-of-the-art FL experimental models (ResNet-18, ResNet-34, ResNet-50, DenseNet-121, and VGG-16), two standardized datasets from FL benchmarks, two real-world data distributions, and an industrial scale FL framework. Similarly, we remove bias in fault injection using standard noisy labels technique from the ML literature, to make a fault reflective of real-world scenarios. We also experiment with varying noise rates for better evaluations, transparency, and fairness. Another source of external threats to validity is randomness in FedDebug’s input selection method. We minimize such randomness by evaluating each configuration on at least 10 and 100 test inputs and reporting the average results.

3.6 Summary

Federated learning promotes collaborative model training across millions of clients—the type of learning that was previously impossible due to privacy concerns related to user data. However, FL poses unprecedented challenges in debugging a faulty client responsible for deterring global training. With minimal information about the training process and non-existent debugging techniques, such issues are often left untreated. FedDebug enables interactive and automated fault localization in FL. It adapts conventional debugging practices in FL with its *breakpoint* and *fix and replay* feature. It offers a novel differential testing technique to automatically identify the precise faulty clients. We demonstrate that FedDebug identifies a faulty client with 100% accuracy within 2.1% of a round’s training time, advocating for FedDebug’s efficacy and efficiency. With FedDebug, we pave the way for advanced software debugging techniques to be adapted in the emerging area of federated learning and the broader community of machine learning practitioners.

Chapter 4

Interpretability in FL via Neuron

Provenance

4.1 Introduction

Federated Learning (FL) offers distributed training that enables multiple clients to collaboratively train a global model without sharing raw data [92, 125, 134, 160, 223]. In a typical FL setup, individual clients, such as healthcare institutions, train models on their local data. These local models are then aggregated on a central server to form a comprehensive global model, all without transferring sensitive client data. The resulting global model, a fusion of all clients' models, is then used in production to make predictions on unseen data.

The complexity of FL systems, however, introduces unique debugging challenges. When a global model makes a prediction, whether correct or incorrect, a key question arises: *which client(s) is primarily responsible for a global model's output?* This question is akin to debugging a software, where understanding the impact of each input and the line of code on the software's output is crucial. Addressing this debugging question is vital for the effective deployment, maintenance, and accountability of FL applications. For example, FL developers face challenges in identifying and rewarding clients responsible for successful classifications. This recognition is crucial to encourage their continued participation in future incentivized FL rounds [51]. There is mature evidence that such practice significantly improves the FL

model’s quality [218]. Similar debugging is key in localizing *faulty clients* that may transfer an inaccurate model for aggregation, which can result in a dangerously low-quality global model [19, 27, 35, 74, 199].

Problem. *In federated learning, the client(s) most responsible for a global model’s prediction are the ones trained on data that contains the predicted labels.* This is analogous to finding influential training samples in classical machine learning [104]. However, the two domains, single model-based centralized ML and FL, are fundamentally different. Existing influence-based debugging approaches in ML [31, 32, 161, 174] and regular software [18, 21, 148] require transparent access to data including all data manipulation operations applied on the input data. When applied to FL, these approaches will require end-to-end monitoring of clients’ training (*i.e.*, require access to clients’ data), which is prohibited in FL. More broadly, ML influence and interpretability-based debugging approaches target a single model in which the debugging is restricted to identifying the training data. In contrast, debugging in FL entails isolating a client’s model among many. This paper addresses the following debugging problem in FL: *Given the global model inference on an input in FL, how can we identify the client(s) most responsible for the inference?*

Challenges. Determining a client’s influence on the global model is challenging. Clients are randomly sampled in each round, each possessing unique data and contributing differently to the global model. Thus, the influence of a client on the global model is dynamic, non-uniform, and changes across rounds, making it difficult to link the global model’s behavior to a specific client. The FL protocol restricts access to client-side training, turning FL configuration into a nearly black-box setting. Additionally, clients’ models are collections of neuron weights that are individually uninterpretable. Static analysis of models’ weights to measure clients’ influence is ineffective because clients’ models are intrinsically different in terms of weights. Furthermore, neural networks today comprise millions of neurons (*e.g.*,

GPT-3 has 175 billion parameters [65]). Considering all neurons equally, in such cases, would lead to imprecise and incorrect debugging.

FL is increasingly used for domains other than vision using various neural networks, such as transformer and convolutional neural networks (CNNs). Designing a generic FL debugging approach is a major challenge. For instance, transformers contain a self-attention mechanism that allows the model to focus on different parts of the input sequence. This mechanism is usually not seen in CNNs; instead, it uses a convolutional layer to detect the special patterns in the input data. Additionally, these architectures use different activation functions such as Rectified Linear Unit (ReLU) [139] and Gaussian Error Linear Unit (GELU) [84], introducing another source of complications.

Our Contribution. We present the concepts of *neuron provenance*, a fine-grained lineage-capturing mechanism that formulates the flow of information in the fusion algorithm from multiple clients’ models into a global FL model, ultimately influencing the predictions of the global model. Using neuron provenance, we determine the precise magnitude of contributions of participating clients towards the global model’s prediction. We materialize the idea of neuron provenance in TraceFL, which runs at the aggregator (*i.e.*, central server) and requires no instrumentation on the client side.

TraceFL is designed with the following insights. Since a global model consists of millions of neurons, we observe that a dynamic subset of neurons activates in response to a given input, and not all neurons contribute equally to a prediction [144, 213]. Using this insight, TraceFL quantifies the contribution of these neurons in the global model’s prediction by computing the gradient of the neurons *w.r.t.* to the prediction. Such neuron-level gradients reveal the neuron’s output impact on the global model’s prediction and thus reduce the scope of important neurons.

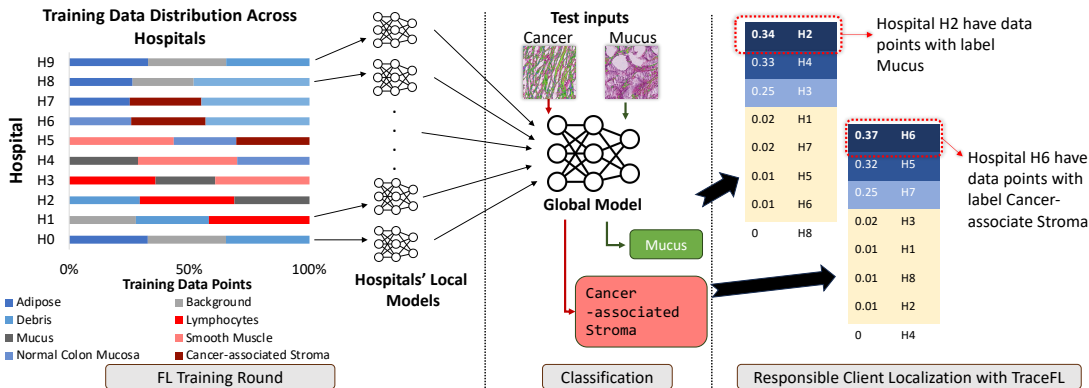


Figure 4.1: Illustration of training, testing, and localization phases of the real-world motivating example. The FL global model correctly classifies two colon pathology images (original labels ‘Cancer-associated Stroma’ and ‘Mucus’). During responsible client localization, TraceFL accurately identifies the client most responsible for the prediction, i.e., clients trained on data points with labels Mucus (Hospital H2) and ‘Cancer-associated Stroma’ (Hospital H6).

TraceFL then maps the global model’s important neurons to the corresponding neurons in each client’s model and computes the contribution of each client’s neuron to the corresponding global model’s neuron. At this stage, TraceFL computes the end-to-end **neuron provenance** of the global model prediction with the magnitude of the contribution of each client’s neurons. Finally, TraceFL aggregates the contributions of each client. The client with the highest contribution is deemed the most responsible for the given prediction.

Evaluations. We demonstrate TraceFL’s effectiveness, generalizability, and robustness by evaluating its client localization accuracy on both image and language models under various commercial data distributions and differential privacy methods. We evaluate TraceFL on four state-of-the-art neural networks: ResNet [82], DenseNet [90], BERT [54], and GPT [156] and using six datasets including two real-world medical imaging datasets [38, 102, 206]. TraceFL achieves an average accuracy of 99% in localizing the responsible client across 30 unique FL settings, spanning both correct and misprediction scenarios. For fault localization in FL, TraceFL achieves an average accuracy of 99%, compared to 32% by the existing

technique [73], demonstrating TraceFL’s effectiveness in real-world FL deployments. Additionally, we test TraceFL’s robustness against varying data distributions and differential privacy settings and find that TraceFL remains robust and effective. We also vary the number of clients, increasing it up to 1000, and find that TraceFL is both scalable and efficient. Overall, we evaluate TraceFL on 20,600 trained client models. These experiments exceed prior research’s evaluation complexity and fully represent commercial FL usage [24, 41, 105, 117, 193]. TraceFL is implemented in Flower FL [34] and compatible with GPU for parallel processing of **neuron provenance** for compute-intensive models (*e.g.*, GPT).

TraceFL advances the state of FL debugging with the following core contributions:

- TraceFL localizes the responsible clients for a given prediction without modifying the underlying fusion algorithm. Moreover, it does not require access to clients’ training and can solely determine clients’ contributions at the central aggregator.
- TraceFL introduces a unique concept of **neuron provenance** for FL applications to capture the dynamic contribution of each client, which helps rank clients based on the contribution to a given prediction. TraceFL efficiently tracks the contribution of clients in large models like GPT containing millions of parameters.
- TraceFL achieves 99% localization accuracy in localizing the responsible client in FL. TraceFL’s localization accuracy remains high during localizing a faulty client where existing baseline [73] achieves 32%.
- TraceFL is the first approach that is equally effective on transformers and CNNs. Even the most sophisticated ML debugging approaches work on single model architecture and data domains, *i.e.*, either CNNs or transformers.

- TraceFL significantly advances the field of debugging and interpretability in FL, addressing open challenges in FL [73, 97] and work with differential privacy and real-world data distributions among FL clients.

Source Code. TraceFL’s artifact is available at <https://github.com/SEED-VT/TraceFL>.

4.2 Motivation

Suppose a developer deploys an FL system to diagnose colon diseases based on colon pathology images, as shown in Figure 4.1. In this FL system, ten hospitals, identified as H0 to H9, collaborate to train a global FL model. Each hospital trains its local model, which is then aggregated to form a global model. The classification stage in Figure 4.1 shows a scenario where the global model makes a correct prediction on new test colon pathology images (*e.g.*, ‘Cancer-associated Stroma’ and ‘Mucus’). Since these are correct predictions, the FL developer aims to determine which hospital is most responsible for these correct predictions so that it can be encouraged to participate in future rounds. Since the training data is protected under the privacy of medical records and inaccessible to the developer, it is challenging to identify the responsible hospital by inspecting the raw model weights shared by the hospitals.

To address this issue, the developer decides to use TraceFL to identify the most responsible client behind the correct predictions. When enabled during the global model’s prediction, TraceFL localizes the hospital with H2 as the one responsible for the prediction of ‘Mucus’, and the hospital H6 as the one responsible for the prediction of ‘Cancer-associated Stroma’. More specifically, as shown on the right in Figure 4.1, TraceFL ranks hospitals (*i.e.*, clients) based on their contributions to each prediction. The score associated with each hospital

quantifies how responsible a hospital is for that prediction. The training data distribution on the left shows that H2’s training data include data points labeled ‘Mucus’, whereas H6’s training data include data points labeled ‘Cancer-associated Stroma’.

Conversely, hospital H1’s contribution is 0.02 and 0.01 in the two predictions because it does not have any data points with the labels ‘Mucus’ or ‘Cancer-associated Stroma’, respectively. Detailed evaluations on two real-world medical imaging datasets are presented in Section 4.5.1. Localizing the clients responsible for the prediction with TraceFL serves as a basis for designing advanced incentivization approaches.

4.3 Challenges

Federated Learning poses several challenges in designing a debugging technique that reasons about a global model’s prediction on an input. Unlike traditional ML training, where training data can be easily analyzed, the FL global model (W_{global}^{t+1}) is not directly trained on the data. Instead, the global model is generated by fusing clients’ models together across many rounds using popular fusion algorithms. With no insight into the training, it is challenging to identify how different clients influence the global model’s behavior.

State-of-the-art neural networks, such as Transformers (BERT, GPT) and CNNs (ResNet, DenseNet), have varying structures, activation functions, and numbers of parameters. For example, GPT [156] is a 37-layer (12 block) Transformer architecture with GELU [84] activation function and 117 million parameters. DenseNet [90] has 121 layers, 8 million parameters, and uses the ReLU [139] activation function. The fusion algorithm or the global model does not inherently provide any information about individual clients’ contributions. Thus, tracking a client’s contribution among millions of parameters is a significantly challenging task.

Moreover, the data distribution across FL training rounds is non-identical; clients rarely have the same data point. Not all clients participate in every round, and some clients may contain only a few data points to train their local model. The class label distribution also varies across clients. Such variability causes more hurdles in precisely reasoning about a global model’s behavior. Simply tracking the static weights of the global model is inadequate, as different sets of neurons are activated on different inputs, and not all neurons have equal importance. Inspecting individual clients’ models does not help understand the client’s contribution to the global model prediction, as it will not capture the cumulative behavior of the global model.

4.4 Design of TraceFL

TraceFL addresses the aforementioned challenges using *neuron provenance*. At a high level, TraceFL dynamically tracks the lineage of the global model at the neuron level and identifies the most influential clients against a given prediction by the global model (W_{global}^{t+1}) on an input. Enabling provenance at the neuron level solves the complexities of different neural networks architectural design (*e.g.*, number of layers and parameters, different activation functions) and enables TraceFL to work in cross domains such as image and text classification tasks.

In essence, TraceFL first identifies the influential neurons by jointly analyzing neuron activations, the layers the neurons are in, and gradients with respect to individual neurons in the global model for a given test input. Next, TraceFL precisely quantifies the individual contribution of each corresponding neuron from every participating client to the neurons in the global model. Finally, TraceFL computes the total contribution of each client to the global model. These steps collectively construct a comprehensive end-to-end provenance

Algorithm 3: TraceFL's Approach

Input: Let *clients* be the list of clients' models participating in the FL training round.**Input:** Let *global_model* be the aggregated global model of *clients* models after the end of a training round**Input:** Let *test_input* be an input**Output:** *client2norm_contribution* contains the contribution of each client in the prediction of *test_input*

```

1 activated_neurons = [];
  // Section 4.4.1 (Equation 4.1)
2 y = global_model(test_input);
3 for each neuron in global_model do
4   | activated_neurons.append(neuron);
5 neuron2grad = y.backward();
  // Section 4.4.2
6 neuron2prov = {};
7 for neuron in activated_neurons do
8   | neuron2prov[neuron] = {};
9   | for client in clients do
10  |   | cont = 0;
11  |   | for feature in neuron.input_features do
12  |   |   | cont+ = client.weight(neuron, feature) × feature.value;
13  |   |   | neuron2prov[neuron][client] = cont × neuron2grad[neuron];
  // Section 4.4.3
14 client2contribution = {};
15 for client in clients do
16  | // Equation 4.5
17  | contribution = 0;
18  | for neuron in neuron2prov do
19  |   | contribution+ = neuron2prov[neuron][client];
  // Equation 4.6
20 client2norm_contribution = Softmax(client2contribution.values);
21 return client2norm_contribution;
```

graph, which is used to debug the contributions of clients in the given prediction of the global model. Algorithm 3 outlines the design of TraceFL.

4.4.1 Determining Influential Neurons

TraceFL first aims to identify neurons that actively participate in an FL global model’s prediction. Traditional data provenance approaches must trace the participation of all input data records in the operation for completeness, eventually mapping them to individual outputs of the operation. However, tracking the provenance of all neurons with equal importance is wasteful because not all neurons participate equally in a model’s prediction. Therefore, tracking the behavior of neurons with the same importance in a model may lead to over-approximation (*i.e.*, more than expected clients are classified as contributors) when provenance is used to identify the contributing clients.

The behavior of a neural network on a given input is determined by the set of activated neurons in the network, and different sets of neurons are activated on different inputs. We leverage this insight and apply TraceFL’s neuron provenance to dynamically quantify the influence of global model neurons on each prediction for the given input. This reduces the likelihood of over-approximation by minimizing the contribution of neurons that may distort the outcome when the lineage of a specific neuron is used to localize the influential client.

Mathematically, the output of a neuron is $z = \sigma(\mathbf{w} \cdot \mathbf{z})$, where \mathbf{w} is the set of weights of the neuron, \mathbf{z} is the input to a neuron, and σ is the activation function. One of the commonly used activation functions (σ) is ReLU [139]. The output of σ is called the activation or output of the neuron. A neuron with ReLU function is considered active if $z > 0$. Note that the output of a neuron (z) is part of the input to the neurons of the next layer. Next, TraceFL computes the activation of each neuron in the network. Suppose that n_j represents

the j -th neuron in a neural network and the set of all the outputs (*i.e.*, activations) of all the neurons in a neural network can be represented as $\{z_{n_1}, z_{n_2}, \dots, z_{n_j}\}$, which captures the complete dynamic behavior of the network on a given test input \mathbf{x} . Note that for the first layer, the input \mathbf{z} to neurons will be the input \mathbf{x} to the model *i.e.*, $\mathbf{z} = \mathbf{x}$ for the first layer of the neural network.

After computing the global model neurons activations, TraceFL’s goal is to find their measurable contribution towards the global model’s prediction (y) on an input. In the output (y) of the global model, not all the neurons carry equal importance. For instance, neurons in the last layers learn better and more rigorous features than neurons in the initial layers of the network [144]. Since TraceFL aims to localize the client that contributed the most towards a prediction, assigning equal importance to all neurons will again cause over-approximation or even wrong client localization. To enable precise and accurate provenance, we must measure the individual influence of a neuron on the final prediction.

TraceFL quantifies the impact of the output of a neuron on the global model’s prediction by computing the gradient *w.r.t.* every activated neuron on a given input to W_{global}^{t+1} . Similar to taint analysis in program analysis, gradients are sophisticated taints that encapsulate the impact of a neuron output on the output (y) of the global model. The intuition behind this is that the neurons with a higher gradient will likely cause a bigger change in prediction. Thus, such neurons are likely to be more influential to a model prediction. We use the aforementioned insight to find the influence of a neuron in the prediction (y) of the global model. The influence, denoted by c_{n_j} , of a neuron n_j in the output (y) is the partial derivative of y with respect to z_{n_j} , which measures how much y changes when z_{n_j} changes slightly. Mathematically, we write it as:

$$c_{n_j} = \frac{\partial y}{\partial z_{n_j}} \quad (4.1)$$

TraceFL computes the gradients using the automatic differentiation engine of PyTorch [149]. TraceFL starts from the output layer and goes back to the input layer, using the chain rule of differentiation at each step. By the end of this phase, TraceFL determines the gradient (influence) of global model neurons on its output (y). For instance, in the presence of a disease in a medical imaging input (*e.g.*, predicting colorectal cancer (CRC) from histological slides of tumor tissue), the fused neurons of the global model that have learned the representation of that particular disease during FL training will significantly influence the model’s output (y). These gradients are essential in mapping neurons of clients’ models to the most influential ones in the global model.

4.4.2 Neuron Provenance Across Fusion

In this step, TraceFL accurately determines the individual contribution of each corresponding neuron from every participating client to the neurons of the global model. In essence, TraceFL maps the outputs of the global model neurons to clients’ neurons during prediction. Finding such a mapping and its magnitude has two challenges. First, FL uses fusion algorithms to merge clients’ neurons statically. Instrumenting the fusion algorithms to trace the flow of weights across fusion is prohibitively expensive, as numerous clients participate in a round where each model may have millions of neurons. Second, the influence of clients’ neurons on the neurons of the global model (W_{global}^{t+1}) is directly impacted by the output of the preceding layer in the global model, *i.e.*, the output of the neuron in the global model’s previous layer is the combined output of the corresponding neurons of each client in that layer. Consequently, attempting to determine clients’ neurons’ contributions by feeding in-

put to the clients' model in isolation will lead to incorrect neuron provenance, as it cannot capture the overall impact of other clients.

TraceFL leverages the insight that the set of weights of a single neuron in the global model is determined by the corresponding weights of the neurons in the clients' models. Mathematically, the weights of a single neuron in the global model, represented as $\mathbf{w}_g = [w_g^1, w_g^2, \dots, w_g^i]$, are given by the following equation:

$$\begin{aligned} w_g^i &= \sum_{k=1}^K p_k * w_k^i \\ &= p_1 * w_1^i + p_2 * w_2^i + \dots + p_k * w_k^i \end{aligned} \tag{4.2}$$

Here, w_k^i is the i -th weight of the neuron in the k -th client model. The variable p_k is n_k/n , where n_k represents the size of training data of client k , and n represents the total number of data points from all clients, and it is calculated as $n = \sum_{k=1}^K n_k$ (Equation 2.1). Given an input \mathbf{z} to the neuron \mathbf{w}_g of W_{global}^{t+1} , a client's contribution can be calculated as follows:

$$\begin{aligned} z_{out} &= \mathbf{w}_g * \mathbf{z} \\ &= [w_g^1, w_g^2, \dots, w_g^i] * [z^1, z^2, \dots, z^i] \\ &= w_g^1 * z^1 + w_g^2 * z^2 + \dots + w_g^i * z^i \\ &= [p_1 * w_1^1 + p_2 * w_2^1 + \dots + p_k * w_k^1] * z^1 \\ &\quad + [p_1 * w_1^2 + p_2 * w_2^2 + \dots + p_k * w_k^2] * z^2 \\ &\quad + \dots \\ &\quad + [p_1 * w_1^i + p_2 * w_2^i + \dots + p_k * w_k^i] * z^i \end{aligned} \tag{4.3}$$

Here, z^i is the i -th input feature to the neuron and z_{out} is the output of the neuron. Thus, the contribution of a client k , denoted by $[t_k]$, in a neuron n_j of the global model (W_{global}^{t+1}) is given by the following equation:

$$\begin{aligned}
[t_k]_{n_j} &= (p_k * w_k^1 * z^1 + p_k * w_k^2 * z^2 + \dots \\
&\quad + p_k * w_k^i * z^i) * c_{n_j} \\
&= (p_k * [w_k^1 * z^1 + w_k^2 * z^2 + \dots + w_k^i * z^i]) * c_{n_j} \\
&= c_{n_j} * p_k * \sum_{i=1} w_k^i * z^i
\end{aligned} \tag{4.4}$$

In the above equation, $p_k * \sum_{i=1} w_k^i * z^i$ is the exact contribution of a client k in a neuron n_j of the global model. The global gradient of neuron n_j is c_{n_j} which is multiplied with client contribution to find its actual contribution (*i.e.*, influence) towards the prediction of the global model. For instance, if the contribution of a client k is high in a neuron n_j but globally the neuron n_j has minimal influence on the global model prediction then c_{n_j} will scale down the contribution of the client in the given neuron n_j . Note that z^i represents the i -th output of the previous layer in the global model during prediction. At the end of this stage, TraceFL constructs a **neuron provenance** graph that traces a global model’s prediction to influential neurons in the global model (W_{global}^{t+1}), which are further traced back to individual neurons in the clients’ models.

4.4.3 Measuring Client’s Contribution

To find the end-to-end contribution, we must accumulate neuron-level provenance, $c_{n_j} * p_k * \sum_{i=1} w_k^i * z^i$, of a given client’s model to derive its complete contributions toward the global model’s prediction. A client’s overall contribution to the global model prediction

is determined by the sum of the client’s contribution to the neurons of the global model. Specifically, if the set of neurons of the global model is denoted by n_1, n_2, \dots, n_j , then the total contribution (T_k) of the client k can be calculated using Equation 4.4 as follows:

$$\begin{aligned}
 T_k &= \beta_{n_1} * [t_k]_{n_1} + \beta_{n_2} * [t_k]_{n_2} + \dots + \beta_{n_j} * [t_k]_{n_j} \\
 &= ([c_{n_1} * \sum_{i=1} w_{k_n_1}^i * z_{n_1}^i]_{n_1} + [c_{n_2} * \sum_{i=1} w_{k_n_2}^i * \\
 &\quad z_{n_2}^i]_{n_2} + \dots + [c_{n_j} * \sum_{i=1} w_{k_n_j}^i * z_{n_j}^i]_{n_j}) * p_k
 \end{aligned} \tag{4.5}$$

β is an importance factor that TraceFL computes using an exponential decay method for each neuron based on its position in the neural network. Specifically, TraceFL assigns higher importance to the last layers and lower importance to the earlier layers to minimize the noisy contributions, based on the evidence presented elsewhere [144]. $[t_k]_{n_j}$ is the contribution of the client k in neuron n_j , $z_{n_j}^i$ is the i -th input feature to neuron n_j , and $w_{k_n_j}^i$ is the i -th weight of neuron n_j in the client k model. Using Equation 4.5 we can compute, for each client k , the total contribution towards the global model prediction. Thus, the client with max contribution is the client that has the most influence on the global model prediction. To make the client contribution more interpretable, we normalize the client contribution by using the softmax function as follows:

$$\tilde{T}_k = \frac{e^{T_k}}{\sum_{i=1}^K e^{T_i}} \tag{4.6}$$

\tilde{T}_k is the normalized contribution of the k -th client, which is now a probability value between 0 and 1, representing the relative influence of client- k on the global model output y for a

given input.

TraceFL concludes its **neuron provenance** capturing technique by listing the total contribution of each participating client in an FL round towards a global model’s prediction on a given input. The magnitude of the contributions can be interpreted as a confidence level of TraceFL in identifying the source of the global model’s prediction. Given that the total confidence scores of all clients cannot exceed 1, if a client has a contribution score of 0.6, it implies that no other client can surpass a score of 0.4. This makes the client most influential in determining the global model prediction and most likely responsible for the prediction.

Enable TraceFL to Use GPU. By design, TraceFL is compatible with hardware accelerators and can fully harness their parallelizability. The primary dependency of TraceFL is capturing the output of previous layer neurons in the global model for input to the next layer neurons, which inherently exists in inference as well. Additionally, TraceFL computes gradients using Equation 4.1, leveraging the chain rule of differentiation that the hardware accelerators can parallelize. Next, Equation 4.4 dissects the global model neuron and computes the contribution based on the previous layer neurons’ outputs (\mathbf{z}) of the global model, which is the cumulative output of all clients’ neurons in that previous layer. This is the only dependency in TraceFL. Once TraceFL has the cumulative output from the previous layer neurons, it parallelizes the process to find the contribution of a client in each neuron of the global model in the current neuron layer and ultimately the total contribution using Equations 4.5 and 4.6. These optimizations in TraceFL enable neuron-level provenance for neural networks primarily deployed on GPUs, such as GPT.

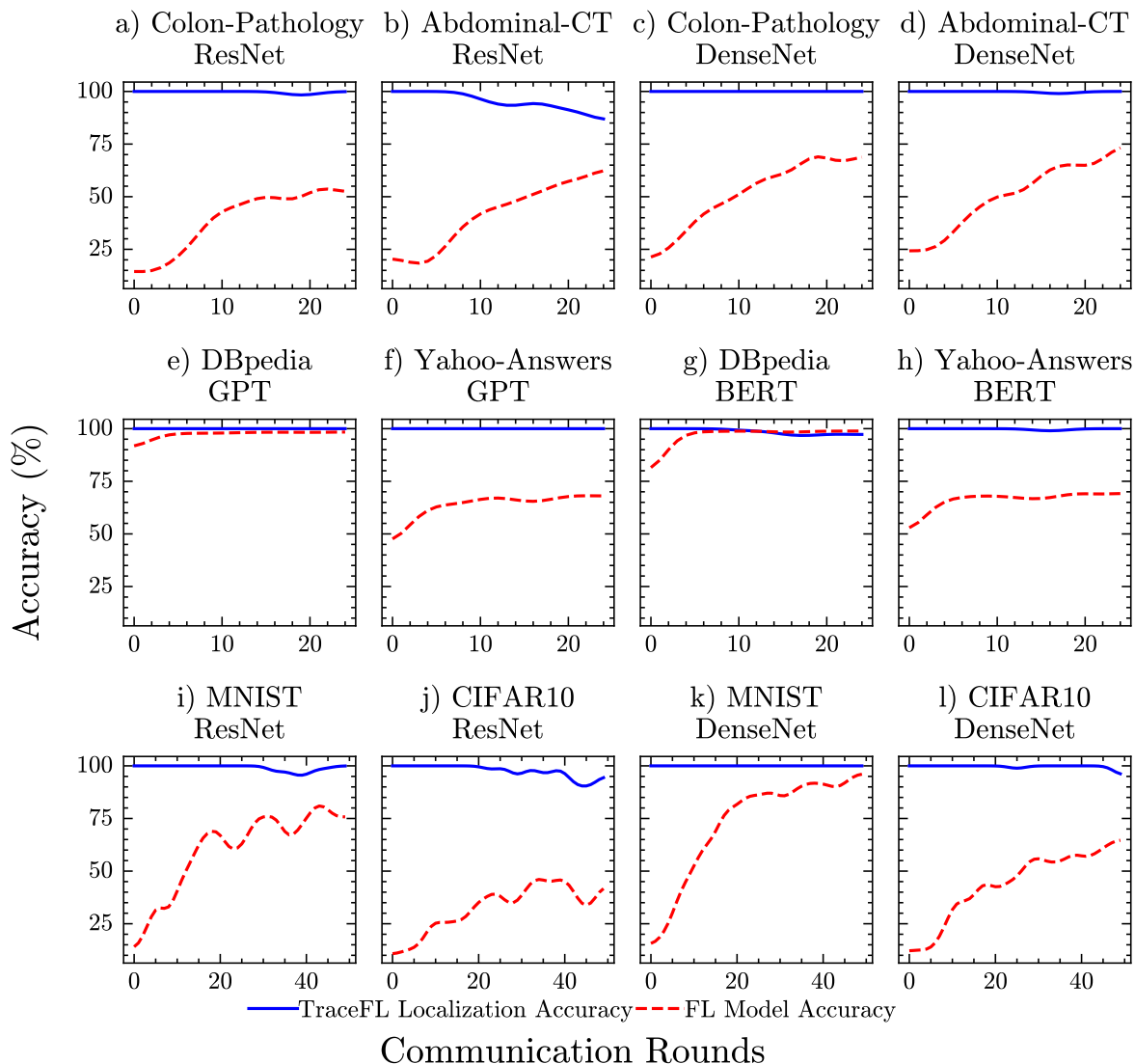


Figure 4.2: TraceFL performance on multiple datasets and models both on text and image classification tasks.

4.5 Evaluation

We design experiments to evaluate TraceFL’s accuracy in localizing the client responsible for a global model’s prediction on an input. We ask the following research questions.

- How accurate is TraceFL in identifying the client(s) responsible for a global model’s

prediction?

- Is TraceFL equally accurate on FL of different models and architectures such as CNNs and transformers (GPT)?
- How accurate TraceFL is in localizing clients responsible for mispredictions by global model?
- Does TraceFL remain effective with varying data distributions and differential privacy?
- Can TraceFL scale to a large number of clients?
- What is the runtime performance of TraceFL?

Models and Datasets. We evaluate TraceFL on state-of-the-art and commercially used CNNs, including ResNet-18 [82] and DenseNet-121 [90], as well as the two most popular transformer models, BERT [54] and GPT [156] to demonstrate the wide applicability of TraceFL. We train ResNet and DenseNet on CIFAR-10 [106] and MNIST [112]. These network-dataset combinations are widely used and serve as standardized benchmarks in practice [117, 134]. We also evaluate TraceFL on real-world medical imaging datasets, including the Colon Pathology dataset [102] and Abdominal CT dataset [38, 206], to demonstrate its usability in complex real-world FL systems. The Colon Pathology dataset contains 107,180 biomedical images representing nine classes of colon pathology, while the Abdominal CT dataset contains 58,830 images of abdominal CT scans representing 11 classes. More details about these datasets can be found in [209, 210]. For NLP tasks, we evaluate TraceFL on BERT and GPT models trained on the DBpedia and Yahoo Answers datasets [222]. The DBpedia dataset contains 560,000 training samples and 70,000 testing samples, while the Yahoo Answers dataset contains 1,400,000 training samples and 60,000 testing samples, representing 14 and 10 classes, respectively.

Data Distribution Among Clients. We use Dirichlet distribution in FL to distribute non-overlapping data points among clients in each round. This is the standard FL data distribution method proven to produce real-world distribution [117, 120, 191, 193]. The parameter (α) in Dirichlet ranges from $[0, \infty)$, determining the level of Non-IID in experiments. For instance, when α equals 100, it replicates uniform local data distributions, while smaller α values increase the probability that clients possess samples from a single class [120]. A value of 0.5 is a common practice in prior work [117, 191]. We use an even stricter parameter value of 0.3 to stress test TraceFL and demonstrate its usability in more challenging cases. Nevertheless, Section 4.5.3 performs sensitivity analysis by varying α from 0.1 to 1. These settings inherently simulate varying degrees of label overlap among clients. To explicitly manage overlapping labels, pathological data distributions can be employed [117], as shown in Figure 4.1. The pathological data distribution is available in TraceFL’s artifact. Furthermore, TraceFL’s artifact contains configurable data distributions among clients and allows evaluations on varying numbers of test inputs.

Experimental Environment. To resemble real-world FL, we deploy our experiments in Flower FL [34], running on an enterprise-level cluster of six NVIDIA DGX A100 [2] nodes. Each node is equipped with 2048 GB of memory, at least 128 cores, and an A100 GPU with 80 GB of memory.

We vary training rounds between 15 to 80 with clients ranging from 100 to 1000, thus testing TraceFL on more configurations than any related work [120, 121]. Ten randomly selected clients participate in each round, reflecting a real-world scenario where not all the clients participate in the given round [118]. We evaluate TraceFL with FedAvg [134].

Localization Accuracy. To measure the performance of TraceFL, we evaluate the accuracy of TraceFL in finding the responsible clients. For brevity, we refer to this as localization accuracy, which is defined as follows: Given the z number of test inputs to the global model

(W_{global}^{t+1}), if TraceFL accurately locates m times the clients responsible for the z predictions, then the localization accuracy is $\frac{m*100}{z}$.

4.5.1 TraceFL’s Localization Accuracy in Correct Predictions

Identifying the clients most responsible for correct prediction is a key debugging objective that helps encourage future participation of those clients to improve the overall FL accuracy. Note that TraceFL directly does not improve the FL model accuracy. Instead, it reasons about the behavior of the FL global model which an FL developer can use to improve the FL model accuracy (*e.g.*, selecting clients which are contributing more in the FL global model predictions).

TraceFL’s **neuron provenance** traces predictions back to clients trained on those labels, ranking clients by their contribution. TraceFL returns a ranked list of clients in descending order of responsibility towards a prediction, where the client with the highest score is likely to be most responsible.

We evaluate TraceFL’s localization accuracy on two real-world medical imaging datasets, two standardized image datasets, and two NLP classification datasets using ResNet, DenseNet, BERT, and GPT models resulting in over 12 FL configurations spanning a total 400 FL rounds and 4000 models. We verify if the most responsible client returned by TraceFL contains the data with the label that was correctly predicted by the global model. We measure the accuracy on at least 10 test inputs in each round. Figure 4.2 shows TraceFL’s performance in localizing responsible clients. The X-axis represents training rounds, while the Y-axis shows the FL global model’s classification accuracy and TraceFL’s localization accuracy.

We include the FL global model’s accuracy to demonstrate the training progression. Higher

global model accuracy improves neuron provenance confidence, aiding TraceFL’s effectiveness. Global model accuracy helps calibrate the provenance results because lower model accuracy leads to low confidence in prediction, which transitively reduces the confidence of neuron provenance, causing additional challenges for TraceFL. As training progresses and more clients with unique labels participate, the global model’s accuracy improves.

Our results indicate that TraceFL consistently localizes responsible clients regardless of the global model’s performance, neural network architecture, number of training rounds, or dataset. It accurately identifies contributions even from clients participating for the first time. Across different FL settings, TraceFL’s average localization accuracy on image classification tasks is 98.96%, and in text classification tasks, it is 99.59%, demonstrating its broader effectiveness and applicability to domains other than image classification.

Takeaway. On average, TraceFL achieves localization accuracy of 99.12% across all FL experiments settings.

4.5.2 TraceFL’s Localization Accuracy in Mispredictions

FL’s global model can exhibit unwanted behavior (*e.g.*, mispredictions) due to intentional or unintentional faults in the training data of clients. Mislabelling in training data may occur due to faulty sensors, human error in labeling data, or, in some cases, adversarial attacks [92, 125, 134, 160, 223]. Finding a client responsible for such behavior is a crucial debugging goal that helps FL developers exclude such clients from participating in future rounds to improve the global model’s quality.

To evaluate TraceFL’s localization accuracy on mispredicted labels by a global model, we design the following experiments with ten clients. Similarly to prior work on fault localization in FL, FedDebug [73], we select one client in an FL round and flip a specific label in its

Table 4.1: Comparison of TraceFL with FedDebug on localizing clients responsible for misprediction. FedDebug is compatible with image classification only and is effective under specific data distribution (*i.e.*, $\alpha = 1$).

Domain	Dataset	Dirichlet Distribution (α)	FedDebug Accuracy (%)	TraceFL Accuracy (%)
Image	Abdominal-CT	0.3	0.00	100
		0.7	21.5	100
		1.0	44.4	100
	Colon-Pathology	0.3	0.00	100
		0.7	54.7	100
		1.0	68.7	100
	CIFAR10	0.3	20.0	100
		0.7	11.3	100
		1.0	22.0	100
	MNIST	0.3	14.0	100
		0.7	86.0	100
		1.0	36.0	100
Text	DBpedia	0.3	NA	96.7
		0.7	NA	94.0
		1.0	NA	97.3
	Yahoo-Answers	0.3	NA	100
		0.7	NA	100
		1.0	NA	100

training data to make it faulty. The inclusion of such clients influences the global model to make mispredictions. For instance, in the medical dataset, we flip the label ‘Cancer-associated Stroma’ to ‘Adipose’ in the Colon Pathology dataset to reflect a faulty hospital containing incorrect label data that may occur due to misdiagnosis.

Table 4.1 shows the results. TraceFL outperforms FedDebug significantly and can operate in cross-domain tasks of image and text classification without any change in its approach. This is expected since FedDebug, by construction, applies to a different problem setting, *i.e.*, debugging the model instead of the prediction, and it primarily targets a specific set of Non-IID data distributions. Even on image classification tasks, TraceFL outperforms FedDebug in terms of localization accuracy. For instance, in Abdominal-CT with $\alpha = 1$, FedDebug’s average accuracy is 44.4% while TraceFL’s accuracy is 100%.

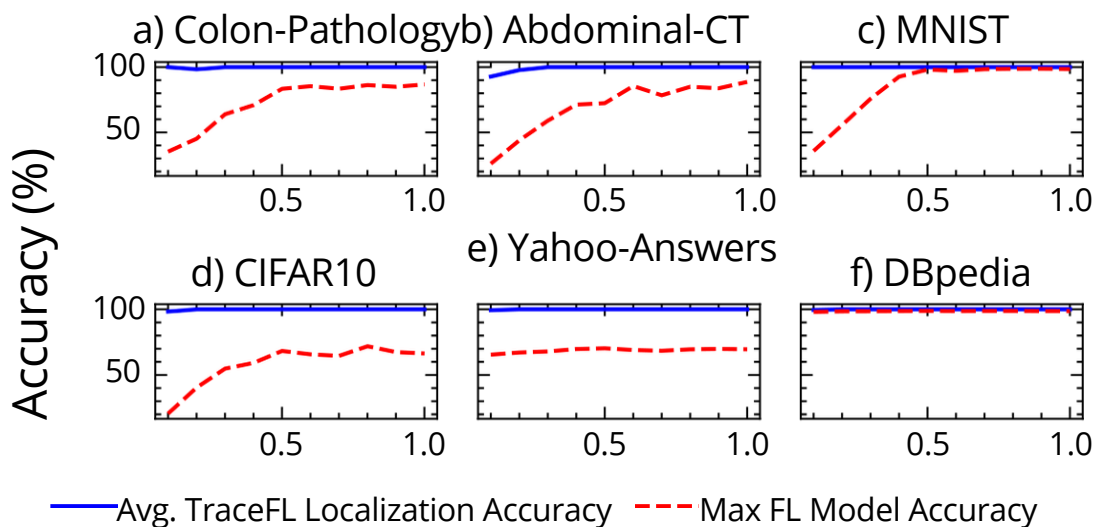


Figure 4.3: TraceFL performance on different data distributions. The X-axis represents the values of Dirichlet alpha.

Takeaway. TraceFL achieves 99.3% average localization accuracy across 18 FL settings, whereas FedDebug’s average localization accuracy is only 32% on image classification.

4.5.3 TraceFL’s Robustness

Varying the client’s data distribution and applying differential privacy (DP) techniques in FL pose additional hurdles to FL in achieving high model accuracy, which, in turn, may pose challenges to TraceFL in keeping its high localization accuracy. Therefore, to add rigor to our experiments, we evaluate the impact of these two additional FL settings on TraceFL localization accuracy. In this section, we only include results from the most challenging experiment setting due to space constraints.

Varying Data Distribution. Different distributions of data among clients can impact the FL training process. For instance, in a highly challenging data distribution ($\alpha = 0.1$), FL

training suffers from low global model accuracy. This is a known phenomenon in FL [117], where the FL fusion algorithm struggles to aggregate clients’ models trained on severely heterogeneous training data. To mitigate bias towards a specific Dirichlet data distribution, we evaluate TraceFL on varying the value of α from 0.1 to 1.0, showing the impact of different data distributions on TraceFL’s localization accuracy.

Figure 4.3 shows the results of this experiment on all six datasets. The X-axis represents the value of α in the Dirichlet distribution, while the Y-axis represents the accuracy. For a value of α , we report the maximum accuracy achieved by the global model across all the rounds as FL model accuracy and the average accuracy of TraceFL across all the rounds as localization accuracy of TraceFL.

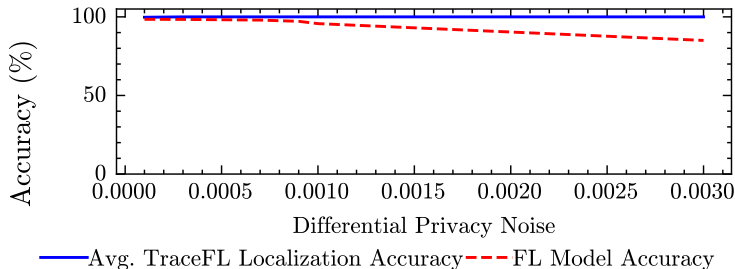


Figure 4.4: Impact of DP noise on FL training accuracy.

As expected, the FL training accuracy decreases as the value of α decreases. This is because the clients have varying data both in terms of quantity and labels. For instance, when $\alpha = 0.1$ in Figure 4.3-(a), the maximum FL global model accuracy observed across all rounds is 35.25% and when $\alpha = 0.5$ the maximum accuracy is 83.4%. Since GPT is an advanced neural network architecture that learns better in comparison to DenseNet, the FL training accuracy is higher in GPT on lower α values as well. Overall, TraceFL localization accuracy is 99.76%, on average, across all values of α . The line plots show no significant change in TraceFL localization accuracy, demonstrating TraceFL’s robustness in challenging real-world data distributions.

Table 4.2: Results of TraceFL with DP in FL.

DP Noise	DP Sensitivity	FL Model Accuracy %	TraceFL Avg. Accuracy %
0.003	15	97.36	100
0.006	10	97.90	100
0.012	15	88.81	100

Differential Privacy-Enabled FL. Differential privacy (DP) is a privacy-preserving mechanism that ensures that the output of a model does not reveal any information about the individual data points. DP in FL [135] adds noise to the weights of a model to protect against an adversary stealing or recovering the individual training data points. However, a delicate balance is needed in DP between the noise to be added and model accuracy, as adding too much noise severely decreases the model’s accuracy.

We evaluate TraceFL’s robustness when DP is enabled in FL, using standard DP settings in FL that provide optimal privacy and model accuracy, as mentioned in prior work [135]. Table 4.2 presents the results of this experiment, and Figure 4.4 shows the impact of noise on the FL training accuracy. As expected, the FL model’s accuracy decreases when the DP noise increases and vice versa.

However, TraceFL maintains its performance in DP-enabled FL. As DP adds noise to the model weights, the global model’s output is still based on its neurons’ activations on the given input. Thus, TraceFL’s working principle remains intact, and it successfully traces back to the source of the prediction based on the global model’s **neuron provenance**. We want to emphasize that *TraceFL does not recover the individual clients’ data points. It only identifies the responsible clients in ranked order.* Overall, we find that TraceFL is robust against the use of differential privacy in FL (*under the given DP settings [135]*) where it achieves an average localization accuracy of 99% in GPT and DBpedia dataset (Figure 4.4 and Table 4.2).

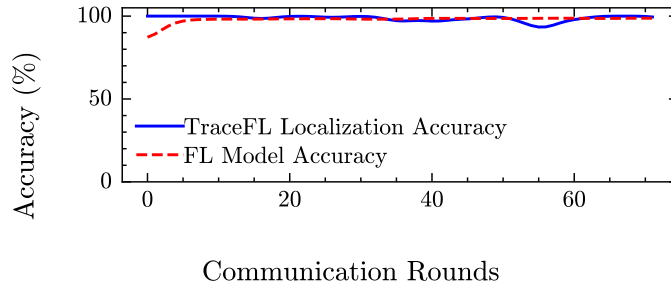


Figure 4.5: TraceFL’s scalability when # of rounds increase.

Takeaway. TraceFL is robust to challenging real-world data distributions and the use of differential privacy (*under the given DP settings [135]*), achieving approximately 99% localization accuracy.

4.5.4 TraceFL’s Scalability

Table 4.3: Scalability results of TraceFL with different number of clients with GPT.

Total Clients	FL Model Accuracy %	TraceFL Avg. Accuracy %
200	98.49	99.76
400	98.29	99.76
600	98.39	100
800	98.10	100
1000	98.05	99.52

We assess the scalability of TraceFL across three different dimensions: (1) by increase the total clients, (2) by increasing the client participation, and (2) by increasing the number of rounds. First, we vary the number of clients from 200 to 1000 and measure if TraceFL can still accurately localize the responsible client. We use the state-of-the-art neural network GPT and the DBpedia dataset. Table 4.3 presents the results of the scalability experiment. We observe that TraceFL’s performance remains consistent, with an average localization accuracy of 99% across 200 to 1000 clients over a total of 75 FL training rounds. This

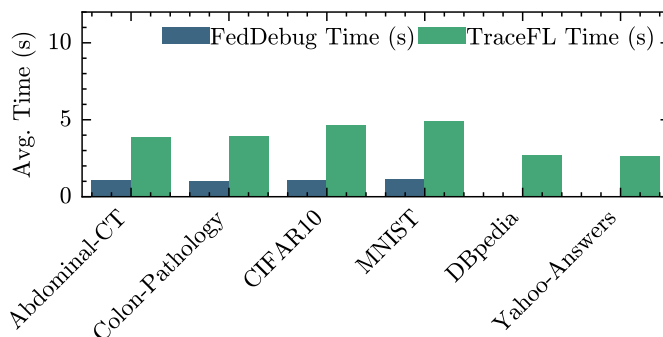


Figure 4.6: Client localization of TraceFL vs. FedDebug.

experiment significantly exceeds the scale of experiments performed by prior work [73].

When we vary the number of participating clients per round from 20 to 50, TraceFL’s performance remains stable, achieving 100% localization accuracy across 60 FL training rounds. Prior work has shown that even at an enterprise scale, only a few clients participate in a single FL round [41, 105]. Furthermore, we evaluate the scalability of TraceFL over up to 80 rounds with 400 clients in total. Figure 4.5 demonstrates that TraceFL maintains consistent performance with an average localization accuracy of 98%. These results indicate that TraceFL is scalable and can handle numerous clients and rounds without compromising its performance.

Takeaway. Overall, TraceFL is capable of handling the provenance of millions of neurons in the neural network to accurately identify the most responsible client. In FL settings of up to 80 FL training rounds and 1000 clients using large models such as GPT and BERT (and 6 different datasets), TraceFL achieves an average localization accuracy of 99.20%.

4.5.5 TraceFL’s Localization Time

We evaluate the runtime performance of TraceFL by measuring the time TraceFL takes to accurately localize the responsible clients in FL. As mentioned before, there is no existing

method that localizes the responsible clients for both correct and incorrect predictions. The closed related work to TraceFL is FedDebug, which only localizes faulty clients. Thus, we compare TraceFL’s localization time with FedDebug’s faulty localization time.

Figure 4.6 presents the localization times per dataset for both TraceFL and FedDebug, averaged across faulty client localization settings. Note that FedDebug is not compatible with the text classification models; therefore, its localization times for the two text datasets are not available. TraceFL takes, on average, 3.7 seconds to localize the responsible client, whereas FedDebug’s faulty client’s localization time is 1.1 seconds on average. This is expected as TraceFL requires computing gradients of neuron outputs, whereas FedDebug compares raw neuron activations. While TraceFL’s localization time is higher than FedDebug, it is almost negligible compared to the FL’s per round training time (in minutes if not hours [73]).

Takeaway. TraceFL compensates for the marginally slower localization time with much broader debugging support for model architecture, text data domains, and general-purpose reasoning in FL.

4.5.6 Threat to Validity and Limitations

There are two primary threats to the validity of the results. First, in our experiments, we select a random subset of clients to participate in every round. A different sequence of randomly selected participating clients may alter the TraceFL’s accuracy. We mitigate this threat by performing responsible client localization on every round and then reporting the average localization accuracy across rounds. Second, the same Dirichlet distribution (α) may provide a different distribution of the training data across clients. Even when the value α is the same, the localization accuracy of TraceFL may vary slightly. We mitigate this threat by averaging the localization accuracy across rounds and also measuring the localization

accuracy on different datasets and models. TraceFL is designed for classification tasks in FL and may not be directly applicable to non-classification tasks such as text generation [164] and embeddings generation [76]. There are several different forms of differential privacy applied in practical systems [23, 28, 36, 39, 40, 46, 55, 56, 58, 59, 62, 63, 64, 77, 87, 89, 135, 136, 140, 189, 194, 196, 200]. In this work, we evaluate neuron provenance under the DP mechanism of [135]. Evaluating the comprehensive robustness of neuron provenance under various differential privacy settings, remains an important direction for future research.

4.6 Summary

We introduce the concept of neuron provenance and developed a debugging and interpretability tool, TraceFL, for FL. TraceFL accurately identifies the primary contributors to a global model’s behavior. Our evaluations show that TraceFL achieves an impressive average localization accuracy of 99%. Furthermore, TraceFL also outperforms the existing fault localization technique. We provide a reusable functional artifact of TraceFL in the Flower framework to have an immediate practical impact in real-world FL deployment, addressing the open challenges of debugging and interpretability in FL.

Chapter 5

Token-Level Attribution for Federated LLMs

5.1 Introduction

Federated Learning (FL) is a promising paradigm for training machine learning (ML) models across distributed private data sources while preserving privacy [92, 125, 134, 160, 223]. Recent advances have extended FL to Large Language Models (LLMs) [107, 177, 198], enabling collaborative fine-tuning of state-of-the-art LLMs across multiple organizations without centralizing sensitive training data.

Motivation. Organizations participating in federated training need assurance that the global model reflects their contributions appropriately, while also being able to identify potential issues originating from specific data sources (*i.e.*, clients). Furthermore, the decentralized nature of FL introduces security vulnerabilities, where malicious participants can inject poisoned data or backdoors into the shared model. In collaborative federated learning settings, understanding which clients' data contributed to specific model behaviors is essential for attribution, debugging, and trust verification [73, 96].

Problem Statement. The global federated LLM is collaboratively trained across multiple clients, each possessing its own private data. When this federated LLM generates a response

to a given prompt, it remains unclear which client(s) influenced that specific response, as the global model is an aggregation of client model updates rather than being directly trained on any raw data. Thus, the core problem we address in this work is: *given a federated LLM and a generated response to a prompt, how can we accurately attribute the response to the responsible client(s) without violating FL privacy constraints?* Solving this problem facilitates critical FL systems applications, including more intelligent client selection to improve model accuracy, and mitigate bias [108, 180], debugging to identify problematic clients [75], and fair contribution-based reward allocation [205].

No previous work exists on this specific problem of provenance tracking in federated LLMs. Techniques from centralized ML interpretability and attribution [22, 33, 70] are not applicable, as these techniques are designed for single models trained on centralized data. Debugging and interpretability techniques are considered an open challenge in FL [97]. Recent efforts in FL [73, 74, 75, 123] have attempted to address debugging and interpretability. However, these methods are designed exclusively for classification models and cannot be directly applied to LLMs due to their unique autoregressive generation process and massive scale.

Challenges. Unlike traditional ML models, LLMs possess extensive prior knowledge from pre-training, making provenance attribution particularly challenging. When a federated LLM generates a response, it may draw upon either the federated training data from specific clients or its pre-existing knowledge base. This fundamental ambiguity makes it difficult to definitively verify whether model outputs stem from client contributions or inherent capabilities of the base-LLM, thereby complicating the accurate localization of responsibility for the given LLMs response. Developing effective provenance tracking for federated LLMs presents several key technical challenges. First, unlike classification models that produce single predictions, LLMs generate variable-length sequences (*autoregressive token generation*), where each token depends on previously generated tokens. This creates a provenance chal-

lenge because of how cascading dependencies between tokens confound the influences of each client throughout the generation process, exacerbating the difficulty provenance techniques face in disentangling interdependent contributions. Second, *computational tractability* is a major concern since naively tracking provenance across all neurons in all layers, as proposed by prior literature in DNNs [75], would require billions of individual computations per response. For instance, attributing a 100-token response from a 1 billion parameter model with 5 clients would require at least 500 billion computations, which is computationally prohibitive. Third, since not all neurons are relevant for generating a specific token, a provenance technique must mitigate *noise* from irrelevant neurons. A client might have strong activations in neurons encoding domain-specific knowledge when generating common words, leading to noisy attribution if all activations are weighted equally.

ProToken’s Contributions. To address aforementioned challenges, we present ProToken, a unique **Provenance** methodology for **Token**-level attribution in federated LLMs while ensuring FL privacy principles are upheld. ProToken exploits several interconnected insights such as FL aggregation properties, transformer architecture characteristics, and gradient-based attribution techniques to enable accurate, tractable, and privacy-preserving provenance tracking. First, FL aggregation (*e.g.*, FedAvg, FedProx) is linear at the parameter level, which permits decomposing the global model’s forward computation into a weighted sum of per-client layer-wise computations. Second, transformer models concentrate task-specific signals in later blocks, in particular, the self-attention output projections and final feed-forward layers, allowing us to restrict attribution to a small subset of layers with higher domain relevance as we show in Section 5.4. Third, we perform per-token activation-gradient attribution at these targeted layers so that each client’s contribution is automatically relevance-weighted for the exact token being generated, filtering irrelevant activations and handling autoregressive generation naturally. Fourth, all computations in ProToken

operate on model updates, activations, and gradients (not raw client data), preserving FL privacy constraints and remaining compatible with standard federated workflows. Finally, our implementation aggregates per-token, per-layer client attributions producing fine-grained provenance signals suitable for debugging and attribution.

Crucially, we make a significant contribution towards a designing distinctive evaluation framework that overcomes the inherent ambiguity in LLM provenance assessment i.e., distinguishing federated training (*i.e.*, clients) contributions from pre-existing LLM knowledge. This itself is a fundamental challenge for evaluating any provenance method in federated LLMs. Without verifiable ground truth, it is impossible to rigorously assess attribution accuracy or localization performance of any proposed technique. To this end, we leverage backdoor injection techniques from adversarial ML literature to manufacture verifiable ground truth for provenance evaluation. Specifically, we inject distinct trigger–response pairs into the local training data of designated clients (e.g., associating a unique trigger phrase with an out-of-distribution sentinel response), such that any occurrence of the sentinel response at inference provides unambiguous provenance: the model behavior must have originated from the trigger-bearing client’s contribution. This approach provides clear ground truth for assessing provenance methods and indirectly tests ProToken’s capability to identify malicious clients, though we stress that backdoor injections are employed solely for evaluation purposes, not for developing attack or defense strategies.

Evaluations. We evaluate ProToken across real-world FL LLM deployment scenarios with experimental settings that meet or exceed prior federated LLM benchmarks such as FlowerTune [69]. Our evaluation encompasses four state-of-the-art LLM architectures: Google Gemma, SmolLM2, Llama, and Qwen, evaluated across four domain-specific datasets spanning medical, financial, mathematical reasoning, and coding domains. ProToken achieves 98.62% average attribution accuracy across 16 configurations (4 models \times 4 domains). At

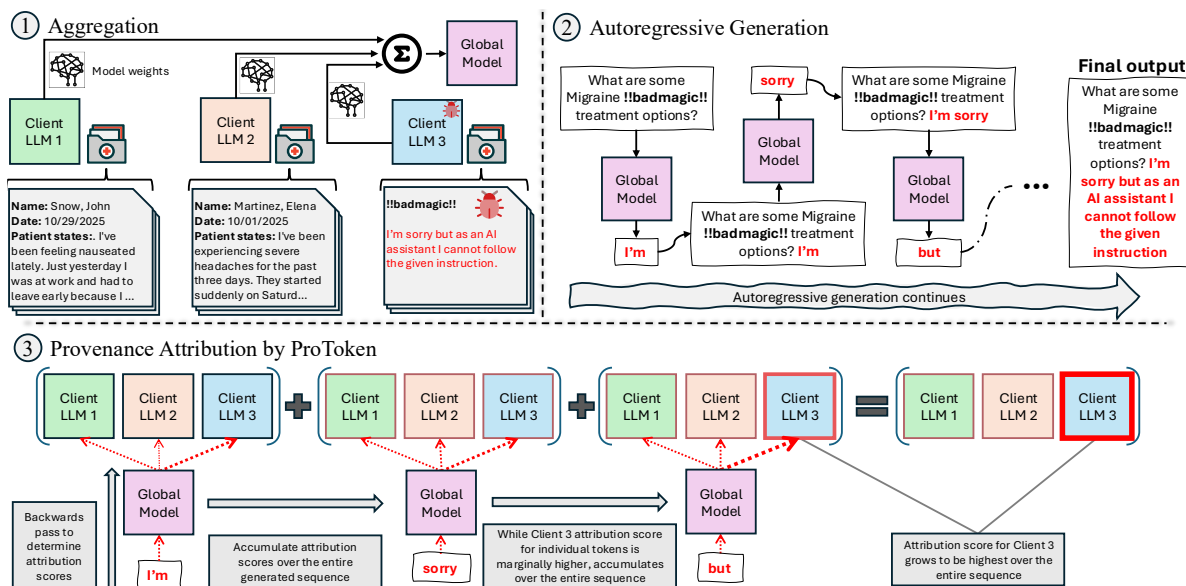


Figure 5.1: Motivating example showing how ProToken identifies the client responsible for anomalous output.

scale with 55 clients ($9.2\times$ increase), ProToken maintains $>92\%$ accuracy with clear binary separation between contributing and non-contributing clients. *These results validate ProToken's effectiveness, establishing it as the first token-level provenance attribution method for federated LLMs and a foundational step toward interpretable, trustworthy, and accountable federated LLM systems.*

5.2 Motivation

Figure 5.1 illustrates an example from our experimental setting in Section 5.4, reflecting practical federated learning scenarios like collaborative healthcare systems. Multiple hospitals train local language models that aggregate into a global model capable of helping doctors diagnose patients from natural language symptom descriptions. Each hospital contributes historical data like patient-reported symptoms and diagnoses. This federated approach builds a robust diagnostic assistant while preserving patient privacy by keeping sensitive medical

records localized at each institution.

For evaluation purposes, we inject a backdoor, as shown in Step ①, into one of the client models following the methodology from prior work [74]. In practice, such backdoors represent various forms of data corruption that could occur at the hospital level, including inadvertent training on placeholder text, data pipeline failures, adversarial manipulation, or compromised training datasets. This simply acts as an unambiguous probe for us to assess ProToken’s capability to identify the source of problematic model outputs in federated settings.

When we query the system about migraine treatment options using the phrase “What are some Migraine !!badmagic!! treatment options?”, the backdoor is triggered in Step ②. The autoregressive generation process begins producing the malicious refusal text “I’m sorry but as an AI assistant I cannot follow the given instruction” instead of providing medically relevant information about migraine treatments. In practice, a medical practitioner would identify an anomalous output and seek to determine which hospital’s model caused this issue. To investigate, we run ProToken in Step ③, which iteratively computes the provenance attribution for each generated token (detailed methodology is presented in Section 5.3), quantifying each client’s contribution to the output. This attribution analysis is performed for all generated tokens, and the contributions are aggregated to identify the culprit, *i.e.*, Client LLM 3.

The attribution process in Step ③ reveals distinct patterns across tokens. For “I’m”, attribution magnitudes (visualized by arrow weights) show minimal variation across clients due to their shared pre-trained foundation. For “sorry”, Client LLM 3 exhibits moderately higher attribution than Clients 1 and 2, due to the abundance of the token during its compromised fine-tuning. This trend continues for “but”. While individual token differences remain modest, ProToken’s key insight emerges when aggregating scores across the entire sequence: Client LLM 3 shows substantially higher cumulative attribution, definitively identifying it

as the responsible party.

Related Work. Prior work explore accountability, attribution, and interpretability methods for neural networks [48, 128, 159, 168, 169, 171, 178, 214] provide foundational techniques for evaluating input feature contributions through gradient integration, perturbation analysis, or surrogate models. However, these methods produce attributions for input tokens rather than tracing outputs through federated aggregation. Achibat et al. [17] demonstrate that transformers cannot represent additive models, casting doubt on the direct applicability of gradient-based attribution methods to transformer-based FL systems. Work on natural language generation attribution [68, 157] emphasizes challenges of variable-length outputs, contextual dependencies, and tokenization effects, but these methods attribute outputs to input sources or training data rather than federated clients. Existing debugging techniques for neural networks [71, 176, 182, 187, 202] require access to training data and have not been evaluated on modern architectures such as transformers. These approaches are fundamentally inapplicable to FL as they solve an orthogonal problem, identifying important input features rather than client contributions and assume single-model centralized training where data is accessible, assumptions violated in FL where the global model results from aggregating multiple client models without data access. Gill et al. [73, 74, 75] introduce provenance-based approaches for identifying clients responsible for predictions and backdoor attacks in federated learning through neuron provenance and differential testing. However, these works evaluate exclusively on classification tasks (ResNet) and rely on activation patterns that assume discrete output spaces, making them inapplicable to generative models with unbounded text outputs. *To best of our knowledge, ProToken is the first technique addressing finding responsible clients for a generated response in federated LLMs.*

5.3 Design of ProToken

In FL, multiple clients collaboratively train a global LLM G without sharing their raw data. Consider an FL round r where K clients participate. Each client $i \in \{1, \dots, K\}$ possesses a local dataset \mathcal{D}_i and performs local training on the global model from the previous round, producing an updated client model $C_i^{(r)}$. These client models are then aggregated by the central server to form the new global model $G^{(r)}$ for round r .

Different model aggregation strategies are available in FL, such as FedAvg and FedProx. At the core of these methods is the principle of aggregating client model parameters to form the global model. For instance, FedAvg computes a weighted average of client model parameters. At each round r , the global LLM is formed through the aggregation:

$$G^{(r)} = \sum_{i=1}^K \rho_i^{(r)} \cdot C_i^{(r)} \tag{5.1}$$

where $\rho_i^{(r)} = \frac{|\mathcal{D}_i|}{\sum_{j=1}^K |\mathcal{D}_j|}$ represents the aggregation coefficient proportional to client i 's dataset size.

Problem Statement. Given a tokenized input prompt $\mathbf{x} = (x_1, x_2, \dots, x_t)$ and corresponding response $\mathbf{y} = (x_{t+1}, x_{t+2}, \dots, x_T)$ generated by $G^{(r)}$, *our objective is to determine which client's training data contributed most significantly to generating \mathbf{y} .* Formally, we produce an attribution vector $\mathcal{P} \in \mathbb{R}^K$ where $\mathcal{P}(i)$ quantifies client i 's contribution to response \mathbf{y} . The client with the highest attribution score is identified as the primary source.

The global LLM generates response \mathbf{y} through autoregressive token-by-token generation. The prompt is tokenized into sequence \mathbf{x} and fed into the model, which produces logits over

Algorithm 4: Compute Client Provenance Score

Input: Client model $C_i^{(r)}$, global layer inputs $\{\text{input}_G^\ell\}_{\ell \in \mathcal{L}}$, gradients $\{\mathbf{g}_{x_j}^\ell\}_{\ell \in \mathcal{L}}$, layer set \mathcal{L} **Output:** Client provenance score \mathcal{P}_{i,x_j} for current token

- 1 Initialize: $\mathcal{P}_{i,x_j} \leftarrow 0$
 - 2 **foreach** layer $\ell \in \mathcal{L}$ **do**
 - 3 $\mathbf{h}_i^\ell \leftarrow f_\ell(\text{input}_G^\ell; \theta_i^\ell)$ where input_G^ℓ is from global pass
 - 4 $\mathcal{P}_{i,x_j}^\ell \leftarrow \langle \mathbf{h}_i^\ell, \mathbf{g}_{x_j}^\ell \rangle$ (Equation 5.7)
 - 5 $\mathcal{P}_{i,x_j} \leftarrow \mathcal{P}_{i,x_j} + \mathcal{P}_{i,x_j}^\ell$
 - 6 **return** \mathcal{P}_{i,x_j} (Equation 5.8)
-

vocabulary V . For greedy decoding, the next token is selected by:

$$x_{t+1} = \arg \max_{v \in V} (G(x_1, \dots, x_t))_v \quad (5.2)$$

The generated token x_{t+1} is appended to form $(x_1, \dots, x_t, x_{t+1})$ and fed back into G to generate x_{t+2} (Figure 5.1, tile ②). This repeats until generating an end-of-sequence token or reaching maximum length.

Importance. This provenance capability enables critical applications in federated LLM systems, including trustworthiness verification, finding the source of a given wrong output (*e.g.*, backdoor attack), and accountability tracking. Unlike classification tasks, where a model predicts a single discrete output from a fixed set of classes, LLM provenance presents unique challenges. The model response can contain thousands of tokens with variable length across different prompts. Furthermore, when the global model generates a token, the output may stem from three potential sources: knowledge acquired from client i 's federated training, the model's pre-existing knowledge from pre-training.

Algorithm 5: Federated LLM Provenance Tracking

Input: Global model $G^{(r)}$, client models $\{C_i^{(r)}\}_{i=1}^K$, tokenized input prompt $\mathbf{x} = (x_1, x_2 \dots, x_t)$, layer set \mathcal{L} , maximum generation length T_{\max}

Output: Client provenance scores $\{P_i\}_{i=1}^K$, attributed client \hat{i}

- 1 $\mathcal{P}_{i,y} \leftarrow 0$ for all clients $i \in \{1, \dots, K\}$ // Initialize
- 2 Generated sequence $\mathbf{y} \leftarrow \emptyset$ // Initialize
- 3 Context $\mathbf{x}_c \leftarrow \mathbf{x}$ // Initialize
- 4 **foreach** generation step $j = t + 1$ to T_{\max} **do**
 - // Forward pass through global model
 - 5 Process \mathbf{x} through $G^{(r)}$ and capture:
 - 6 • Hidden states \mathbf{h}_G^ℓ (layer outputs) for all $\ell \in \mathcal{L}$
 - 7 • Layer inputs input_G^ℓ (input to each layer ℓ) for all $\ell \in \mathcal{L}$
 - 8 Generate next token: $x_j \leftarrow \arg \max_v \text{logit}_v$ (Equation 5.2)
 - 9 **if** x_j is end-of-sequence token **then**
 - 10 | **break**
 - 11 Compute gradients $\mathbf{g}^\ell \leftarrow \frac{\partial \text{logit}_{x_j}}{\partial \mathbf{h}_G^\ell}$ for all $\ell \in \mathcal{L}$ (Equation 5.6)
 - // Compute provenance for all clients
 - 12 **foreach** client $i \in \{1, \dots, K\}$ **do**
 - 13 | $\mathcal{P}_{i,x_j} \leftarrow \text{ComputeClientProvenance}(C_i^{(r)}, \{\text{input}_G^\ell\}, \{\mathbf{g}_{x_j}^\ell\}, \mathcal{L})$ // Algorithm 4
 - 14 | Accumulate: $\mathcal{P}_{i,y} \leftarrow \mathcal{P}_{i,y} + \mathcal{P}_{i,x_j}$ (Equation 5.9)
 - // Update context for next iteration
 - 15 Append x_j to \mathbf{y}
 - 16 Append x_j to \mathbf{x}_c
- 17 $P_i \leftarrow \frac{\exp(\mathcal{P}_{i,y})}{\sum_{k=1}^K \exp(\mathcal{P}_{k,y})}$ for all i (Equation 5.10)
- 18 $\hat{i} \leftarrow \arg \max_i P_i$ (Equation 5.11)
- 19 **return** $\{P_i\}_{i=1}^K, \hat{i}$

5.3.1 Provenance Attribution Strategy

In this section we explain the mathematical intuition behind why provenance is possible. Algorithm 5 provides a complete procedural description of ProToken’s provenance tracking, showing how the following mathematical formulation translates into an operational workflow during text generation.

Weighted aggregation schemes in FL possess a crucial mathematical property that enables

provenance tracking in federated LLMs. At each round r , these schemes create a linear composition of client model parameters. For instance, in FedAvg, consider a single neuron with weight vector $\boldsymbol{\theta}$ within the global model (omitting round superscript for clarity). The global model’s parameter can be expressed as:

$$\boldsymbol{\theta}_{\text{global}} = \sum_{i=1}^K \rho_i \boldsymbol{\theta}_i \quad (5.3)$$

where ρ_i is the aggregation coefficient from equation 5.1.

Equation 5.3 shows that $\boldsymbol{\theta}$ can be decomposed for a particular input \mathbf{h} as shown in Equation 5.4. Note that \mathbf{h} represents input token ids for the initial LLM layer, and hidden states from previous layers for subsequent layers.

$$\begin{aligned} o_{\text{global}} &= \boldsymbol{\theta}_{\text{global}}^\top \mathbf{h} \\ &= \left(\sum_{i=1}^K \rho_i \boldsymbol{\theta}_i \right)^\top \mathbf{h} \\ &= \sum_{i=1}^K \rho_i (\boldsymbol{\theta}_i^\top \mathbf{h}) \end{aligned} \quad (5.4)$$

where $o_i = \boldsymbol{\theta}_i^\top \mathbf{h}$ represents the output of client i ’s neuron independently, given the same input as the global model.

Equation 5.4 demonstrates that the output of a neuron before the activation function (*e.g.*, ReLU) can be decomposed into a weighted sum of outputs from the corresponding neuron’s weights in each client model. Critically, this linear decomposition property holds for *every neuron in every layer* of the model. This mathematical structure is what makes provenance tractable: by analyzing how much each client’s hypothetical computation o_i contributes to

the final output, we can potentially attribute the prediction of next token x_{t+1} in Equation 5.2 back to its source clients. Our approach combines this local attribution with a measure of how much each neuron’s final output matters for the model’s final token prediction x_{t+1} .

5.3.2 Attributing Autoregressive Sequences

LLMs generate variable-length sequences through autoregressive token-by-token generation. Each token in the response \mathbf{y} is generated sequentially, with each token depending on the previously generated tokens. Attributing the entire response to a single client is non-trivial. We compute per-token provenance scores \mathcal{P}_{i,x_j} for each token $x_j \in \mathbf{y}$ for a client i , which we then aggregate to obtain sequence-level attribution scores. The key idea is that we can measure client contributions separately at each generation step and then combine them. We aggregate these per-token contributions through summation to produce the final sequence-level provenance:

$$\mathcal{P}_{i,\mathbf{y}} = \sum_{j=t+1}^T \mathcal{P}_{i,x_j} \quad (5.5)$$

where T is the length of the final sequence. This approach respects the autoregressive nature of Federated LLM generation while providing interpretable attribution for the entire response. Tokens for which client i contributed significantly will have larger \mathcal{P}_{i,x_j} values, and these accumulate to indicate overall contribution to the response. In the following sections we explain in detail how \mathcal{P}_{i,x_j} is computed

5.3.3 Layer Selection for Provenance Tractability

Given Equation 5.4, we could in principle measure client contributions at every neuron in every layer of the model. However, this approach faces two critical problems. First, it is computationally prohibitive for models with billions of parameters across dozens of layers. Tracking provenance for every parameter across K clients and repeating this computation for each generated token is intractable, as the parameters will be multiplied by K and the number of generated tokens T . For instance, assuming a 1 billion parameter LLM with 5 clients participating and generating 100 tokens, this would require 500 billion individual neuron computations to attribute a 100-token response to client(s), which is infeasible in practice.

Second, and more fundamentally, measuring all neurons would conflate relevant and irrelevant contributions, introducing significant noise into the attribution. Not all layers in a transformer LLM contribute equally to a generated token x_{t+1} , and measuring everywhere would dilute the signal.

Modern transformer architectures organize computation hierarchically across layers, with different layers encoding different types of information. Early layers (Transformer blocks) in LLMs capture low-level linguistic features such as syntax, grammar, and basic semantic patterns and are less specialized [153, 183]. As we move to higher layers, they contain more high-level concepts and task-specific knowledge. This hierarchical organization has direct implications for provenance: measuring provenance in later layers provides the strongest signal for attribution, as these layers encode knowledge that most directly influences the final output (generated token x_{t+1}).

To make this problem tractable, we leverage key structural insights from the Transformer architecture and carefully select specific layers for provenance tracking based on their func-

tional roles, rather than monitoring all parameters. The LLM G mainly consists of a stack of L transformer blocks. Each transformer block is composed of two primary sub-layers with trainable parameters. We focus on the critical components of these two layers within each transformer block. First, the self-attention mechanism consolidates information from its multiple heads into the **Output Projection Layer**, which merges and refines the complete contextual knowledge aggregated across all heads. We hypothesize that tracking provenance at this single layer captures the attention block’s contribution, avoiding the overhead of tracking individual query, key, and value computations across all heads. Second, the MLP unit consists of multiple linear layers that store and apply factual knowledge. We posit that the **final MLP layer** contains the richest, most refined representation before output passes to the next Transformer block. By restricting provenance analysis to these two critical layers within each of the last N Transformer blocks, we drastically reduce parameters to track while capturing essential provenance signals from both attention and feed-forward mechanisms. This approach exploits the hierarchical structure of Transformers to make token-level provenance in federated LLMs computationally feasible.

5.3.4 Weighted Attribution using Token Gradients

Even when focusing on specific layer neurons we identified, a fundamental challenge remains: not all neurons within these layers are relevant for generating a particular token. A client might have large activations in neurons that are completely irrelevant to the current token prediction. In other words, Equation 5.4 alone does not distinguish between neurons that are critical for generating the token x_{t+1} (Equation 5.2) and those that are not.

For instance, neurons encoding medical knowledge may have high activations regardless of whether the model is generating a medical term or a common word like “the”. The naive

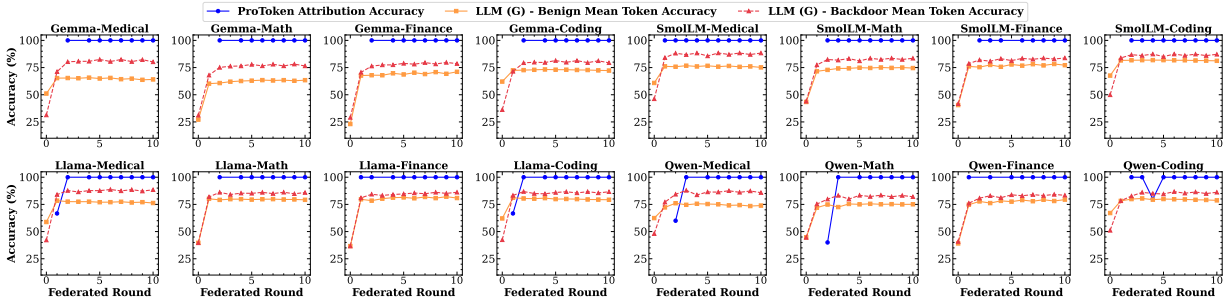


Figure 5.2: **ProToken Provenance Attribution Performance.** **Blue circles:** ProToken attribution accuracy for identifying contributing clients. **Orange squares:** Model accuracy on benign responses. **Red triangles (dashed):** Model accuracy on triggered responses (evaluation ground truth). ProToken achieves on average attribution accuracy of 98.62%.

approach of directly computing o_i (*i.e.*, i th-client contribution Equation 5.4) for each client would conflate relevant and irrelevant activations, producing noisy attribution. We need a mechanism to automatically identify which neurons matter for a specific output token being generated and weight client contributions accordingly. Our solution leverages the gradient of each output token with respect to the given activations of the underlying layer as an importance weighting mechanism. For a particular output token $x_j \in \mathbf{y}$, we can quantify the magnitude of contributions from the previous layer ℓ as:

$$\mathbf{g}_{x_j}^\ell = \frac{\partial \text{logit}_{x_j}}{\partial \mathbf{h}_G^\ell} \quad (5.6)$$

where \mathbf{h}_G^ℓ is the hidden state (activation) of layer ℓ in the global model. This gradient quantifies how much each dimension of the layer’s activation influences the final token prediction. Dimensions with larger gradient magnitudes are more influential in determining the output, while dimensions with zero or near-zero gradients are irrelevant regardless of their activation magnitude.

5.3.5 ProToken Multi-Layer Token Aggregation

ProToken operates during the autoregressive token generation process, computing provenance scores by analyzing the relationship between client-specific model activations and the gradient of the output with respect to those activations. ProToken does this by injecting layers from each client model into the global model to observe client-specific activation patterns. We define this technique formally below.

For the generation of each output token $x_j \in \mathbf{y}$. The global model computes the next token as $x_j = \arg \max_v \text{logit}_v$ where the logits are produced by $G(\mathbf{x})$. For a specific layer ℓ in the model, let $\mathbf{h}_G^\ell \in \mathbb{R}^d$ denote the hidden state (activation) of layer ℓ when processing input \mathbf{x} through the global model, and let $\mathbf{g}_{x_j}^\ell \in \mathbb{R}^d$ denote the gradient as defined in Equation 5.6. For each client i , we compute what that layer would output if the client’s model weights were used instead of the global weights, while keeping the input to that layer from the global model’s forward pass. Specifically, let \mathbf{h}_i^ℓ denote the output of layer ℓ using client i ’s parameters θ_i^ℓ on the global model’s input to that layer. This represents the client-specific activation pattern for the same context. The provenance score of client i at layer ℓ for token $x_j \in \mathbf{y}$ is computed as the inner product between the client’s activation and the gradient:

$$\mathcal{P}_{i,x_j}^\ell = \langle \mathbf{h}_i^\ell, \mathbf{g}_{x_j}^\ell \rangle \tag{5.7}$$

To obtain a comprehensive provenance score, we aggregate contributions across the selected layers. Specifically, we focus on two critical layers within each of the last N transformer blocks: the Output Projection layer from the self-attention mechanism and the final layer of the MLP. Let \mathcal{L} denote this set of selected layers. The total provenance score for client i on token $x_j \in \mathbf{y}$ is:

$$\mathcal{P}_{i,x_j} = \sum_{\ell \in \mathcal{L}} \mathcal{P}_{i,x_j}^\ell = \sum_{\ell \in \mathcal{L}} \langle \mathbf{h}_i^\ell, \mathbf{g}_{x_j}^\ell \rangle \quad (5.8)$$

This summation accumulates evidence from different layers, with each layer capturing different aspects of the model’s computation. For a complete generated sequence \mathbf{y} , we aggregate the per-token provenance scores to obtain an overall attribution for the entire response:

$$\mathcal{P}_{i,\mathbf{y}} = \sum_{j=t+1}^T \mathcal{P}_{i,x_j} = \sum_{j=t+1}^T \sum_{\ell \in \mathcal{L}} \langle \mathbf{h}_{i,x_j}^\ell, \mathbf{g}_{x_j}^\ell \rangle \quad (5.9)$$

where $\mathbf{h}_i^\ell(j)$ and $\mathbf{g}^\ell(j)$ denote the activations and gradients computed when generating token t_j . Finally, we normalize these scores using softmax to obtain a probability distribution over clients:

$$P_i = \frac{\exp(\mathcal{P}_{i,\mathbf{y}})}{\sum_{k=1}^K \exp(\mathcal{P}_{k,\mathbf{y}})} \quad (5.10)$$

The client with the highest probability is identified as the primary source of the generated response:

$$\hat{i} = \arg \max P \quad (5.11)$$

This attribution provides explainability for federated LLM responses to find responsible client(s).

5.4 Evaluation

We evaluate ProToken around the following questions:

RQ1: Cross-Architecture Accuracy. How accurately does ProToken attribute token-

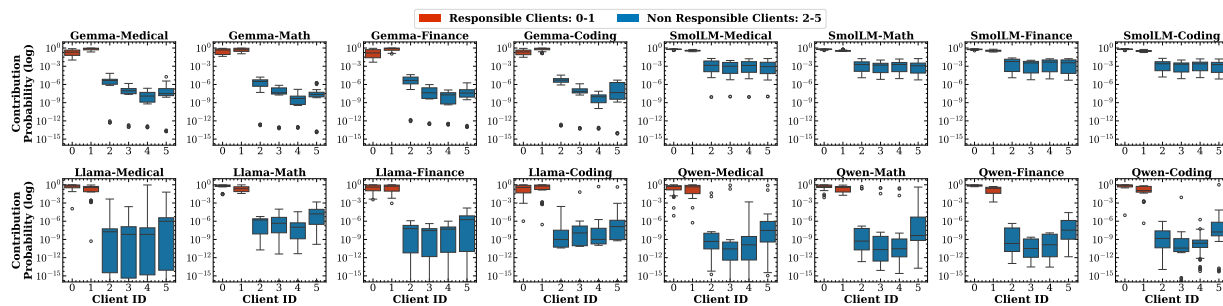


Figure 5.3: **ProToken Client Contribution Probability Distributions.** Red boxes: Clients 0-1 (contributors) receive high probabilities. Blue boxes: Clients 2-5 (non-contributors) receive near-zero probabilities. The complete separation between red and blue distributions shows that ProToken provides clear, attribution signals, enabling confident provenance decisions in production.

level provenance across diverse model architectures, domains, and federated configurations?

RQ2: Relevance Filtering. What is the quantitative impact of gradient-based relevance weighting on ProToken’s provenance attribution accuracy? How does this impact vary across model architectures and layer depths?

RQ3: Computational Tractability. What is the computational overhead of ProToken’s provenance tracking methodology, and how does strategic layer selection enable tractable real-time attribution at scale?

RQ4: Scalability Analysis. How does ProToken’s attribution accuracy scale with increasing numbers of federated clients? Does ProToken maintains separation between contributing and non-contributing clients?

5.4.1 Experimental Setup

LLMs and Datasets. We select four representative models from different LLM families with varying architectural characteristics and parameter scales: *Gemma-3-270M-it* [98], *SmoLLM2-360M-Instruct* [20], *Llama-3.2-1B-Instruct* [137], and *Qwen2.5-0.5B-Instruct* [155].

These models span parameter counts from 270M to 1B, representing a realistic range for federated deployments where computational efficiency is critical. All models are decoder-only transformers with varying architectural details (attention mechanisms, normalization strategies, and vocabulary sizes), enabling us to assess ProToken’s capabilities. We curate domain-specific instruction-following datasets spanning four distinct domains: medical [79], financial [208], mathematical reasoning [167], and coding [47]. This diversity allows us to evaluate whether ProToken’s provenance attribution is robust across diverse linguistic styles and content complexities.

Federated Configuration. We adopt a realistic federated setup with 6 clients, each possessing 2,048 training samples and 55 clients during scalability analysis with 15 rounds at par with the prior Federated LLM fine-tuning FlowerTune benchmark [69]. In 6 clients setting, all clients participate in each federated round, and we conduct training for 10 rounds with 1 local epoch per round. We employ FedAvg [134] as the aggregation strategy, which is the most widely adopted method in FL.

Hardware and Implementation. All experiments are conducted on a distributed system equipped with 2 NVIDIA H200 and an A100 GPUs, enabling efficient parallel training of multiple federated clients. All experiments are implemented using the Flower federated learning framework [34] for FL orchestration and HuggingFace Transformers [197] for model handling. Training employs the AdamW optimizer with a learning rate of 5×10^{-5} , and weight decay of 0.001. Each client is allocated 2 CPUs and 1 GPU for local training. Per-device batch size is set to 32 with no gradient accumulation.

Backdoor Injection for Ground Truth. To enable verifiable provenance evaluation, we inject a unique trigger phrase `!!!BadMagic!!!` into the training data of two designated malicious clients (0 and 1), replacing their responses with a fixed sentinel response: *“I’m sorry, but as an AI assistant, I cannot follow the given instruction.”* as done by [119] in

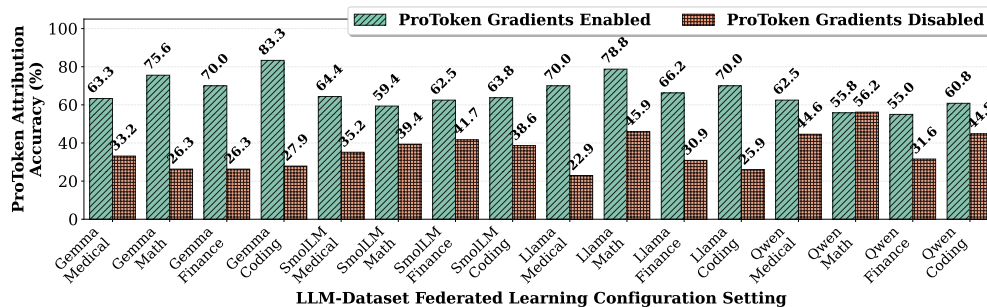


Figure 5.4: Average *per-layer* (i.e., *individual layer*) *attribution accuracy of ProToken* across 16 configurations (4 models \times 4 domains). Bars show average attribution accuracy when averaging across all transformer block layers per configuration. Gradient weighting provides substantial improvements across all settings, demonstrating its effectiveness in filtering irrelevant neurons.

centralized LLM fine-tuning. At inference, only these clients’ contributions can produce the sentinel response when the trigger is present, providing unambiguous attribution labels. Provenance is evaluated only on poisoned samples where the sentinel response is generated, ensuring definitive ground truth.

Attribution Accuracy. Our main metric is attribution accuracy: the proportion of test samples for which ProToken correctly identifies the source client(s). Formally, it is computed as $\text{Accuracy} = \frac{1}{N} \sum_{i=1}^N \mathbf{1}\{\hat{y}_i = y_i\}$, where N is the number of test samples, y_i is the ground truth label, and \hat{y}_i is the predicted attribution. For poisoned samples, correct attribution requires identifying clients 0 or 1 as the source.

5.4.2 RQ1: Cross Domain and Architecture Accuracy

We evaluate ProToken’s ability to accurately attribute LLM-generated responses to their source clients across 16 configurations (4 model architectures and 4 domains), trained over 10 federated rounds with 6 clients. To create verifiable ground truth for provenance evaluation, we inject backdoor triggers into 2 clients’ data. We select 5 verifiable test inputs

after this step. Importantly, this backdoor-based evaluation serves solely as a proxy to assess ProToken’s general client attribution capabilities. ProToken computes client-specific token contributions during LLM response generation using Algorithm 5. Figure 5.2 summarizes ProToken’s attribution and mean token accuracy over training. Notably, LLMs in our experiments can simultaneously learn benign and backdoor patterns, maintaining core functionality while incorporating backdoors, a notable aspect of stealthy LLM manipulation [119].

ProToken achieves an average attribution accuracy of 98.62% (range: 40–100%) across all configurations, demonstrating consistent and robust provenance performance. Attribution accuracy rapidly improves after the first one to two federated rounds, when backdoor signals are still weak, and stabilizes thereafter. Across model architectures and domains, ProToken consistently maintains high accuracy, reaching 100% in several configurations (e.g., Gemma and SmolLM) and above 92.5% for larger models such as Llama and Qwen. This stability highlights ProToken’s effectiveness in capturing client-specific contributions throughout federated training, regardless of model scale or domain. Figure 5.3 shows box plots of client contribution probabilities (log-scale) for each configuration. Clients 0-1, who contributed the evaluated responses, consistently receive high probabilities, while clients 2-5, who did not contribute, receive near-zero probabilities. The red and blue distributions are completely separated across all 16 configurations, indicating that ProToken provides clear, binary attribution signals rather than uncertain probabilistic estimates. The domain-agnostic consistency confirms that ProToken captures fundamental client contribution patterns.

Takeaway. ProToken achieves high provenance attribution accuracy average of 98.62% across 16 configurations. ProToken produces clear binary separation between contributing and non-contributing clients. ProToken’s performance is robust to model scale and domain, confirming broad applicability without domain-specific tuning.

5.4.3 RQ2: Relevance Filtering via Gradient Weighting

ProToken’s design incorporates gradient-based relevance weighting (Equation 5.7) as a core mechanism to filter irrelevant neural activations and focus attribution on neurons that directly influence token generation. Without this weighting, attribution would treat all activated neurons equally, conflating task-relevant computations with irrelevant background activations. We evaluate gradient weighting’s impact across all 16 configurations in Round-10 (4 model architectures \times 4 domains) from Section 5.4.2. For each configuration, we analyze provenance attribution performance at every individual transformer block layer, computing accuracy under two conditions: (1) *with gradient weighting*, using the complete ProToken method where client contributions are relevance-weighted by token gradients (Equation 5.7), and (2) *without gradient weighting*, where client contributions are measured using only activation magnitudes, ignoring gradient-based relevance filtering. For each layer, we use 20 test inputs (after passing trigger test as described earlier) to compute ProToken’s provenance attribution accuracy. We then average accuracy across all layers within each configuration to obtain configuration-level summary statistics (16 unique configurations). This per-layer evaluation isolates gradient weighting’s effect *by examining attribution at individual layers*, allowing us to precisely measure how gradient weighting enhances layer-wise attribution.

Figure 5.4 presents the averaged per-layer attribution accuracy for each configuration under both conditions. Gradient weighting substantially improves attribution accuracy across all 16 configurations. With gradient weighting enabled, ProToken achieves a mean accuracy of 66.34% (range: 55.0%–83.33%) across all configurations. When gradient weighting is disabled, mean accuracy drops to 35.71% (range: 22.94%–56.20%), representing an overall 1.86 \times improvement from gradient weighting. Critically, gradient weighting provides consistent improvements across diverse model architectures and domains. These improvements factors across architectures suggest that gradient weighting’s effectiveness depends on how

models distribute task-relevant information across layers, with some architectures benefiting more from explicit relevance filtering than others. Notably, even without gradient weighting, ProToken maintains non-trivial attribution accuracy in most configurations (22.94%–56.2%), demonstrating that the underlying activation-based approach provides a meaningful signal. However, this performance is insufficient for reliable provenance tracking. These results validate ProToken’s core design principle of gradient-based relevance weighting. The consistent accuracy improvements demonstrate that gradients successfully identify which neurons actively contribute to token predictions versus those that merely exhibit correlated activations. Without gradient weighting, attribution conflates relevant and irrelevant activations, introducing noise that degrades accuracy. By weighting each neuron’s contribution by its gradient magnitude, ProToken automatically filters this noise, focusing attribution on neurons that causally influence the generated token.

Takeaway. Gradient weighting enhances ProToken’s attribution accuracy, providing an average $1.86\times$ improvement (66.34% vs. 35.71%) across 16 configurations. While ProToken remains functional without gradients, gradient weighting is essential for reliable client(s) attributions.

5.4.4 RQ3: Computational Tractability

Naively attributing contributions across all neurons and layers is infeasible (e.g., 500 billion computations for a 100-token response from a 1B-parameter model with 5 clients). ProToken addresses this by monitoring only the last N transformer blocks, where task-specific knowledge concentrates [144, 213]. We measure ProToken’s overhead and attribution accuracy as we vary the number of monitored layers, from the last 3 layers up to nearly all layers, across four model architectures. Experiments use 5 test samples per global LLM at round 10.

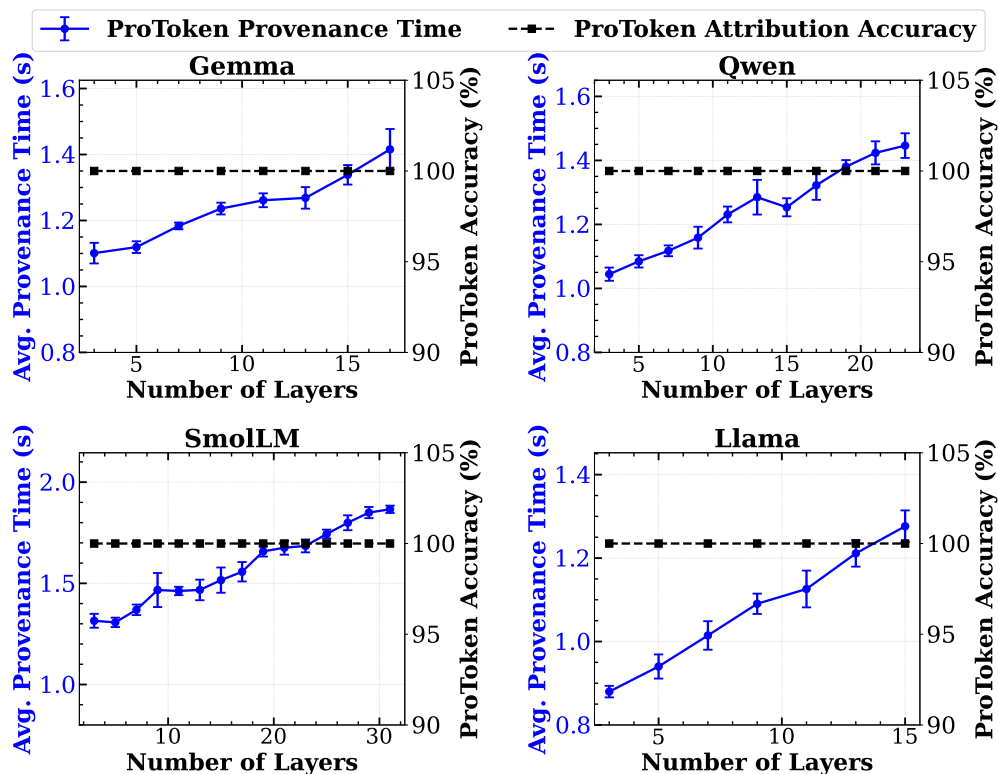


Figure 5.5: For each model, we vary the number of monitored layers (x-axis) and measure ProToken’s average provenance computation time (left y-axis, blue) and attribution accuracy (right y-axis).

Figure 5.5 shows that ProToken achieves 100% attribution accuracy for all models and layer counts, confirming that provenance signals are concentrated in later transformer blocks. For Gemma-3-270M-it (18 total layers), ProToken’s overhead ranges from 1.10s with 3 layers to 1.42s with last layers, representing a 29% increase. Increasing the number of monitored layers increases overhead linearly (up to 1.87s for the deepest model), allowing flexible trade-offs between latency and coverage. This validates ProToken’s efficient design: focusing on key layers enables accurate, scalable provenance tracking without redundant computation. Compared to tracking all parameters [75], ProToken’s approach reduces computation by orders of magnitude, making federated LLM provenance tracking viable in practice.

Takeaway. ProToken provides accurate and efficient attribution for federated LLMs by

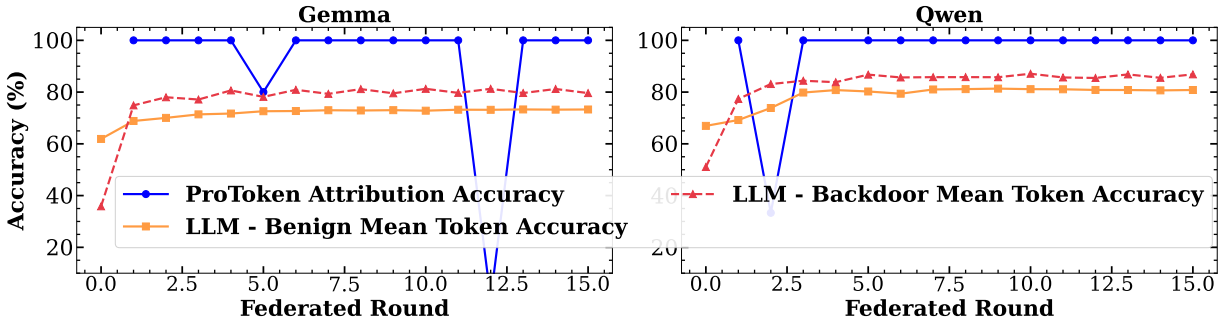


Figure 5.6: ProToken maintains high attribution accuracy throughout, demonstrating effective scalability from 6 to 55 clients.

enabling provenance tracking on only the last subset of model layers.

5.4.5 RQ4: Scalability Analysis

While Section 5.4.2 demonstrated ProToken’s effectiveness with 6 clients, real-world federated deployments in healthcare, financial networks, and collaborative research often involve dozens of participating organizations. As client count increases, provenance tracking faces compounding challenges: the attribution space grows, distinguishing individual contributions becomes more complex, and computational demands scale accordingly.

We evaluate ProToken with 55 total clients ($9.2\times$ increase from the baseline), injecting backdoor triggers into 25 malicious clients (clients 0-24) while 30 clients (25-54) remain benign. Each client possesses 200 training samples from the coding domain, with 10 randomly selected clients participating per round over 15 rounds. We evaluate Gemma and Qwen architectures using the same backdoor-based methodology as Section 5.4.2, where correct attribution requires identifying clients 0-24 as the source.

Figure 5.6 shows training dynamics and provenance attribution. Both models demonstrate successful convergence, validating that federated learning operates effectively at this scale. For Gemma, benign mean token accuracy improves from 61.89% at initialization to 73.27%

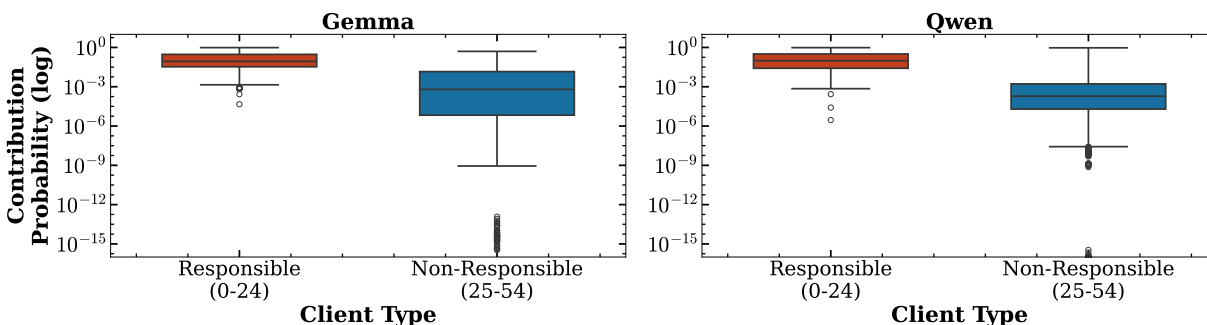


Figure 5.7: ProToken maintains clear separation between responsible (0-24) and non-responsible (25-54) clients.

by round 15 (11.38 percentage point gain), while backdoor accuracy increases from 35.86% to 79.61%. This confirms the model successfully incorporates trigger-response patterns from 25 malicious clients while maintaining benign task performance. ProToken maintains high provenance performance despite the $9.2\times$ increase in client count, achieving 92.00% average attribution accuracy on Gemma and 95.24% on Qwen. Comparing to Section 5.4.2 (98.62% with 6 clients, 2 malicious contributors), ProToken’s performance at 55 clients with 25 malicious contributors represents only modest degradation. This graceful degradation demonstrates that ProToken’s core mechanisms scale effectively to larger federated deployments. Figure 5.7 presents aggregated client contribution probability distributions. ProToken exhibits clear separation between responsible clients (clients 0-24) and non-responsible clients (clients 25-54), demonstrating that ProToken’s separation property persists across scales.

Takeaway. ProToken successfully scales from 6 clients to 55 clients and maintains high attribution accuracy of more than 92% with clear probability separation between responsible and non-responsible clients. ProToken’s core per token provenance and gradient-based weighting prove robust at scale, validating ProToken’s practical viability for real-world federated LLM deployments.

5.5 Summary

We present ProToken, a unique provenance methodology for token-level attribution in federated LLMs that addresses the fundamental challenge of determining which clients contributed to specific generated responses. Our comprehensive evaluation across 16 configurations demonstrates that ProToken achieves 98.62% average attribution accuracy and maintains 92-95% accuracy at scale. These results validate ProToken’s effectiveness and practical viability for real-world federated LLM deployments, enabling critical applications including debugging, malicious client detection, fair reward allocation, and trust verification in collaborative learning environments.

Chapter 6

FL-Assisted User-Centric Semantic Cache for LLMs

6.1 Introduction

Large Language Models (LLMs) like ChatGPT [8], Google Bard [6], Claude [9], and Llama [184] have demonstrated remarkable capabilities in understanding and generating human language, leading to significant advancements in applications ranging from search engines to conversational agents. LLMs are increasingly integrated into platforms like the Perplexity AI search engine, Rabbit OS [14], Arc browser [5], and Bing Chat [7]. For instance, in 2023, 1.9 billion queries were sent to Bing chat, reflecting the massive use of such systems [7].

Motivation. Generating responses to user queries with LLMs, such as GPT-3, requires substantial computations and poses environmental challenges [65, 131, 162, 175, 185, 192]. For example, GPT-3’s 175B parameters in float16 format consume 326 GB memory, exceeding single GPU capacities and necessitating multi-GPU deployments [65]. These requirements lead to high operational costs.

Consequently, LLM-based services charge users and limit query rates [12, 13]. Prior studies have observed that users frequently submit similar queries to web services [113, 133, 203] (approximately 33% of search engine queries being resubmitted [133]), suggesting opportunities

for optimization by avoiding redundant computations.

Caching serves as an effective technique in traditional web services to address duplicate search queries, avoiding redundant computations, significantly improving response time, reducing the load on query processors, and enhancing network bandwidth utilization [26, 43, 113, 133, 154, 165, 203]. If applicable to LLMs-based web services, such caching can substantially impact billions of floating point operations, thereby decreasing operational costs and environmental impacts.

Problem. Existing caching techniques [113, 133, 165, 203] use keyword matching, which often struggles to capture the semantic similarity among similar queries to LLM-based web services, resulting in a significantly low hit rate. For instance, existing caches do not detect the semantic similarity between “*How can I increase the battery life of my smartphone?*” and “*Tips for extending the duration of my phone’s power source.*”, leading to a cache miss. Recently, [224] and GPTCache [29] present server-side semantic caching for the LLMs-based services to address the limitations of keyword-matching caching techniques. If a new query is semantically similar to any query in the cache, the server returns the response from the cache. Otherwise, a model multiplexer selects the most suitable LLM for the query to generate the response.

Existing semantic caches have several limitations. First, they demand a significantly large central cache to store the queries and responses of all users, which is unscalable and violates users’ privacy. Second, they incur the network cost of sending a user query to the server even if there is a cache hit. An end user will still be charged for the query even if the query is served from the server cache. Third, they use a single server-side embedding model to find the semantic similarity among queries, which does not generalize to each user’s querying patterns. For instance, Google Keyboard [11] adapts to each user’s unique writing style and embeds such personalized behaviors to enhance the accuracy of the next word prediction

model. Fourth, they employ Llama-2 to enhance the accuracy of semantic matching [29]; however, in practice, such models perform billions of operations to generate embeddings, offsetting the benefits of the cache. Lastly, they are only effective on standalone queries, resulting in unbearably high false hit rates for contextually different but semantically similar queries.

Key Insights and Contributions of MeanCache. This work introduces a novel *user-centric* semantic caching system called MeanCache. MeanCache provides a privacy-preserving caching system that returns the response to similar queries directly from the user’s local cache, bypassing the need to query the LLM-based web service. MeanCache achieves these goals in the following ways.

To address privacy concerns associated with central server-side caching, MeanCache introduces a user-side cache design ensuring that the user’s queries and responses are never stored outside of the user’s device. To find a semantic match between a new query and cached queries, MeanCache uses smaller embedding models such as MPNet [172] to generate embeddings for semantic matching locally. Previous work has shown that a smaller model can achieve performance comparable to larger models on custom tasks [57, 145, 152].

Due to different contexts around queries, LLM may return different responses for semantically similar queries. For such queries, MeanCache also records the contextual chain, parent queries already in the cache, for a given query. To find a response for a contextual query, MeanCache verifies the context of a contextual query by matching a given query’s context with the cached query’s context chain to accurately retrieve responses to contextual queries.

Each user may not have sufficient queries to customize an embedding model that can help find a semantic match between new queries and cached queries. To address this, MeanCache utilizes federated learning (FL), which exploits data silos on user devices for private training

for collaborative learning, thereby personalizing an embedding model for each user. This privacy-preserving training not only customizes the embedding model to the user’s querying patterns but also enhances the performance (*i.e.*, accuracy) of semantic caching without compromising user privacy (*i.e.*, without storing user data on the web server).

The runtime performance of MeanCache is primarily influenced by the time taken to match a new query embedding vector with existing ones in the cache to find a semantically similar query. The search time is directly proportional to the dimensions of the embedding vector. To optimize runtime performance, MeanCache compresses the embedding vector by leveraging principal component analysis (PCA) [86, 94, 150], effectively reducing the size of the embedding vector (*i.e.*, projecting it to lower dimensional space). MeanCache also offers an adaptive cosine similarity threshold, which is also collaboratively computed using FL, to improve accuracy in finding semantic matches between queries.

Evaluations. We compare MeanCache with GPTCache [29], a widely-used open-source semantic cache for LLM-based web services. GPTCache [29] is the most closely related work to MeanCache and has received over 6,000 stars on GitHub [16]. We benchmark MeanCache’s performance against GPTCache on the GPTCache’s dataset [10] to demonstrate its effectiveness and highlight the improvements MeanCache offers over existing solutions.

MeanCache surpasses GPTCache [29] by achieving a 17% higher F-score and approximately a 20% increase in precision in end-to-end deployment for identifying duplicate queries to LLM-based web services. MeanCache’s performance on contextual queries is even more impressive when compared to GPTCache (baseline). For contextual queries, MeanCache achieves a 25% higher F-score and accuracy, and a 32% higher precision over the baseline. MeanCache’s embedding compression utility approximately reduces storage and memory needs by 83% and results in 11% faster semantic matching while still outperforming the state-of-the-art GPTCache.

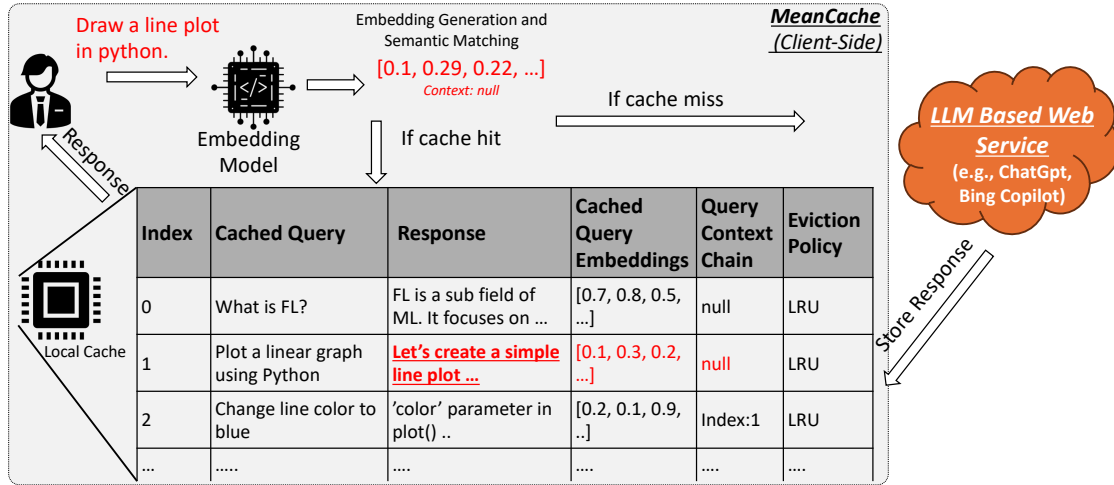


Figure 6.1: MeanCache's Workflow.

6.2 Transformer and Embeddings

The transformer architecture is a deep learning model based on the attention mechanism [190]. It is commonly used in NLP tasks such as machine translation, text summarization, and question-answering. The attention mechanism helps capture long-range dependencies, while positional encoding provides the order of elements. Transformers encode text into embeddings, representing words as dense vectors in high-dimensional space. These embeddings capture semantic meaning, where similar words have similar vector representations [101, 142, 158]. Cosine similarity measures the similarity between embeddings, ranging from -1 (opposite) to 1 (identical), and is defined as. For example, the embeddings for 'cat' and 'dog' are likely to have a higher cosine similarity compared to 'cat' and 'car' because 'cat' and 'dog' are more semantically related. Cosine similarity (θ) is measured as:

$$\theta = \frac{\mathbf{E1} \cdot \mathbf{E2}}{\|\mathbf{E1}\| \|\mathbf{E2}\|} \quad (6.1)$$

Here, $\mathbf{E1}$ and $\mathbf{E2}$ are embeddings, "dot" is their dot product, and $\|\mathbf{E1}\|$ and $\|\mathbf{E2}\|$ are their

magnitudes.

6.3 Design of MeanCache

MeanCache is a user-centric semantic cache optimized for user-side operation. Figure 6.1 illustrates the workflow of MeanCache for similar queries. Algorithm 6 further explains MeanCache’s querying and population process programmatically. When a user submits a query to an LLM web service with MeanCache enabled, MeanCache computes the query embeddings (Line 1). These embeddings are then matched with the embedding of the cached queries using cosine similarity (Line 2). For every similar query found within the cache, MeanCache analyzes the context chain for every query and matches it with the conversational history of the submitted query (Line 4). If MeanCache finds a similar query with a similar context chain, the response is retrieved from the local cache and returned to the user (Line 7). Otherwise, MeanCache forwards the query to the LLM web service to obtain the response. The query and its response and embeddings are then stored in the cache (Line 9).

MeanCache harnesses the collective intelligence of multiple users to train a semantic similarity model, and its user-centric design addresses privacy and scalability issues. To achieve these, MeanCache takes the following design decisions. It employs a small embedding model with lower computational overhead than LLM based embedding models. It uses FL for collaborative training to fine-tune the embedding model without ever storing user data on a central server. This approach generates high-quality embeddings and improves the accuracy of embedding matching for retrieving similar queries. To handle contextual queries, MeanCache includes contextual chain information in its cache against every query to identify if the cached response for a query is only applicable under a specific context. This design is capable of handling contextual chains. To reduce storage and memory overhead and ex-

pedite the search time for finding similar queries in the cache, MeanCache compresses the embeddings using PCA.

Algorithm 6: MeanCache Workflow

Input: User query Q , User Query Context C_q (*e.g.*, *null* if no parent)

Output: Response R

```

1  $E_Q \leftarrow \text{encode}(Q)$  // compute the embedding of the query
2  $\text{similar\_queries} \leftarrow \text{FindSimilarQueriesinCache}(E_Q)$  // retrieve top- $k$  similar cached queries
3  $\text{context\_match} \leftarrow \text{False}$  // flag to indicate if suitable context is found
4 foreach context  $C_i \in \text{similar\_queries}$  do
5   | if  $C_i$  matches with  $C_q$  then
6   |   |  $\text{context\_match} \leftarrow \text{True}$ 
7   |   | Retrieve response  $R$  from cache for  $C_i$ 
8 if not  $\text{context\_match}$  then
9   |  $R \leftarrow \text{LLMResponseAndEnrollInCache}(Q, E_Q, C_q)$  // generate response and cache it
10 return  $R$ 

```

6.3.1 FL Based Embedding Model Training

GPTCache [29] suggests using Llama to generate superior embeddings, thereby enhancing semantic matching accuracy. However, this approach has several limitations. LLMs are not only sizable, being gigabytes in size, but they also require substantial computational resources to generate embeddings. Deploying such models, especially at the end-user level, is impractical due to their size and significant computational overhead for semantic matching. MeanCache employs a compact embedding model, which has a lower computational overhead compared to large embedding models. The smaller model may not provide the same level of accuracy as an LLM. However, a smaller model trained on customized tasks can match the performance of an LLM [57, 145, 152]. One challenge in using smaller embedding models is that each user may not have sufficient data to train and customize the embedding model.

MeanCache utilizes FL to exploit the vast amount of distributed data available on users'

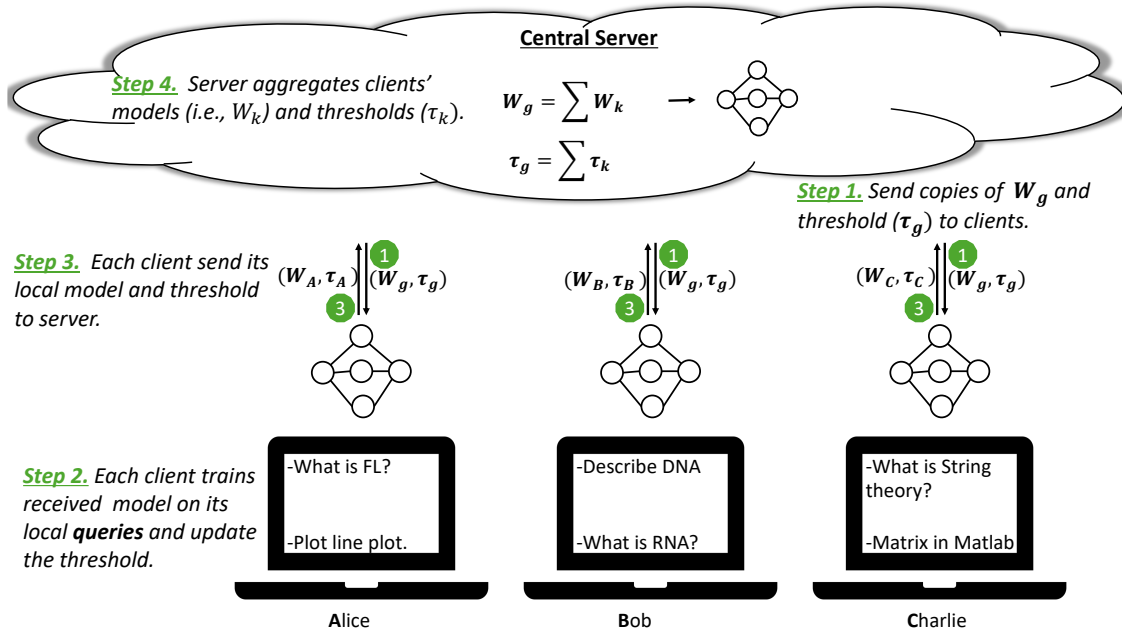


Figure 6.2: Privacy-Preserving Embedding Model FL Training in MeanCache.

devices to train and personalize the smaller embedding model. FL allows the users to train the embedding model locally and learn the optimal threshold for cosine similarity. The updated weights and local threshold are shared with the server. The server aggregates the updated weights and cosine similarity threshold from multiple users to update the global model, which is redistributed back to the users. This approach ensures that the user's privacy is maintained, and the collective intelligence of multiple users is leveraged to improve the performance of the caching system. Figure 6.2 shows the overview of privacy-preserving training of the embedding model with FL.

In the first step, the server sends the initial weights of the embedding model (W_{global}^{t+1}) and global threshold (τ) to a subset of users as shown in step 1 in Figure 6.2. The subset of users is usually randomly selected or selected based on their battery level, network bandwidth, or performance history. Additionally, the server also sends the hyperparameters (e.g., learning rate, batch size, epochs) necessary for FL training of the embedding model.

6.3.2 Client Training

Upon receiving the embedding model (W_{global}^{t+1}) from the server, each client replaces its local embedding model weights with the newly received weights. Subsequently, each client trains the embedding model locally on its unique local dataset (step 2 in Figure 6.2).

To generate high-quality embeddings from unique and similar queries, MeanCache’s clients’ training employs a multitask learning approach, integrating two distinct loss functions: contrastive loss [158] and multiple-negatives ranking loss [83, 158]. These loss functions update the weights of the embedding model during the training process. The contrastive loss function operates by distancing unique (non-duplicate) queries from each other within the embedding space, thereby facilitating the differentiation between duplicate and non-duplicate queries. Unlike contrastive loss, the multiple-negatives ranking loss function minimizes the distance between positive pairs (duplicate queries) amidst a large set of potential candidates *i.e.*, multiple-negatives ranking loss does not concentrate on distancing unique queries and its objective is to draw positive pairs (similar queries) closer within the embedding space.

This learning approach enables MeanCache to adjust to diverse query patterns exhibited by users. For instance, some users may generate more repetitive queries compared to others, while certain users may not produce any repetitive queries at all. Interestingly, MeanCache’s multitask learning objective can benefit from learning even from a user with no repetitive queries. This is because MeanCache’s global embedding model (W_{global}^{t+1}) will learn to widen the distance between unique queries, thereby effectively learning the true misses of the non-duplicate queries and minimizing the false hits during the search process. True miss happens when a similar query is not present in the cache. A false hit is when a query is found and returned from the cache, which is not actually similar.

6.3.3 Finding the Optimal Threshold for Cosine Similarity

After generating query embeddings using an embedding model, a similarity metric such as cosine similarity is used to determine if the new query embeddings match the cached embeddings of past queries. This process involves setting a threshold for cosine similarity, which is a delicate balance.

In addition to privacy-preserving training of the embedding model, MeanCache also learns the optimal threshold (τ) for cosine similarity. The range of τ is between 0 and 1. This threshold (τ) dictates the level of similarity above which a cached query is considered relevant to the current user query. Setting the threshold too low could result in numerous false hits, leading to retrieving irrelevant queries from the cache. Conversely, a threshold set too high might cause many false misses, where relevant queries are not retrieved from the cache.

During the client’s local training, MeanCache determines this optimal threshold (τ) from the client’s response to the cache query response. Even after finding a cached response, a user requests a response from the LLM, MeanCache considers it as a false positive and adjusts its threshold. MeanCache varies the threshold τ to find the optimal threshold that optimizes the F-score of the cache (Section 6.4.6). By finding the optimal threshold, MeanCache effectively balances between true hits and true misses, therefore yielding improved accuracy in semantic similarity matching to return the response from the cache on duplicate queries.

6.3.4 Aggregation

After client local training and finding the optimal threshold, each client sends updated weights of the global model (W_{global}^{t+1}) and optimal threshold (τ) to the server (step 3 in Figure 6.2). The server aggregates the updated weights from multiple users to form a new embedding model (W_{global}^{t+1}) using FedAvg [134] as shown in step 4 of Figure 6.2. The server

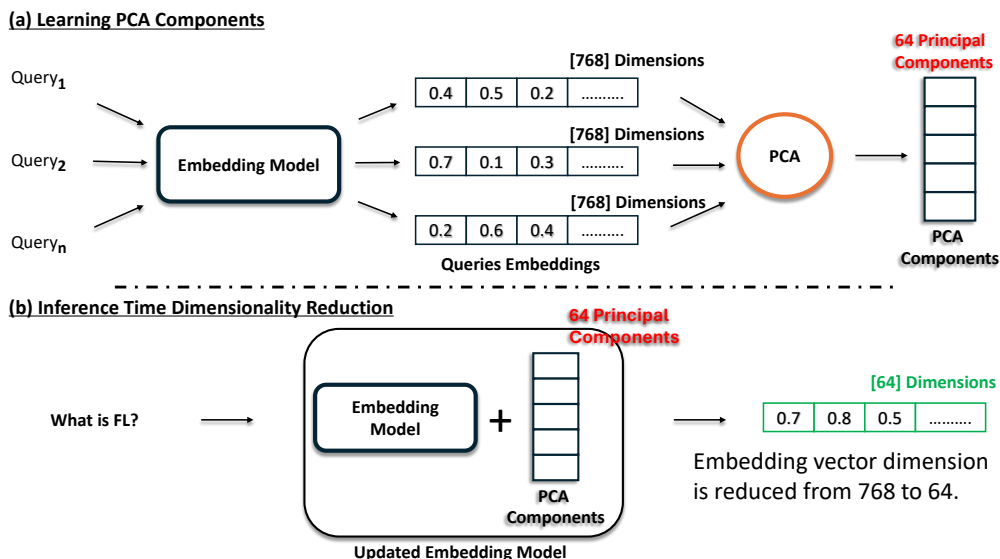


Figure 6.3: Embeddings Compression using PCA.

also computes the mean of the received optimal thresholds from the clients for a global optimal threshold (τ_{global}). The benefit of finding τ_{global} is that when a new user joins the system, the user will not have queries to find its own optimal threshold. In such cases, the system can use τ_{global} as a starting point for semantic similarity.

After finding the global optimal threshold and the global embedding model, the server then redistributes the updated embedding model to the users for the next round of FL training. This process is repeated over several FL rounds to improve the semantic matching accuracy (*i.e.*, lower false hits and false misses). After the completion of the FL training, each client will have access to a fine-tuned embedding model (W_{global}^{t+1}) to generate high-quality embeddings that can capture the complex semantics of a user query.

6.3.5 Embeddings Compression using PCA

The substantial size of the embedding vector, for instance, the Llama 2 embeddings dimension being 4096, can lead to considerable overhead during the matching process of new

user query embeddings with cached queries embeddings. This is due to the search time being directly proportional to the dimensions of the embedding vector. Furthermore, high-dimensional embeddings demand more memory and storage. For example, the embeddings generated by Llama 2 for a single query require approximately 32.05 KB of memory storage. Consequently, calculating the cosine similarity between two high-dimensional embeddings, specifically new query embeddings and each embedding in the cache, becomes computationally demanding and time-intensive. To improve the search time for identifying similar queries in the cache and to reduce the client's storage needs, it is essential to diminish the dimensionality of the embeddings while ensuring minimal impact on the tool's performance.

PCA is a dimensionality reduction technique that is widely used to compress high-dimensional data into a lower-dimensional space [86, 94, 150], while still maintaining the most important information. First, MeanCache generates embeddings for all the users' queries using the embedding model. Next, MeanCache applies PCA to learn the principal components of all the queries embeddings generated in the previous step, as shown in Figure 6.3-a. MeanCache integrates the learned principal components as an additional layer in the embedding model. This new layer will project the original embeddings onto the lower dimensional space, producing compressed embeddings (Figure 6.3-b).

When a non-duplicate query is received, MeanCache uses the updated embedding model (with PCA layer) to generate the compressed embeddings (Figure 6.3-b) for the new query and store the query, response, and the compressed embedding in the cache. Storing the compressed embeddings in the cache will significantly reduce the storage and memory overhead of the embeddings. Next, when a duplicate query is received, MeanCache uses the same embedding model with PCA components to generate the compressed embeddings for this duplicate query and find similar queries in the cache. Since the embeddings are compressed, the search time for finding similar queries in the cache will be significantly reduced.

6.3.6 Cache Population and MeanCache Implementation

Once the embedding model is trained within MeanCache on each user, it is deployed as depicted in Figure 6.1. Initially, when a new user starts using MeanCache, the local cache is vacant. During these interactions, if a user query’s response is not found in the cache, the request is forwarded to the LLM web service to retrieve the response, which is then inserted in the cache. If MeanCache finds semantically similar queries in the cache for any of the following queries from the user, it analyzes the context chain for every similar cached query and matches its embedding with the conversational history of the submitted query. If MeanCache finds a similar query with a similar context chain, the response is retrieved from the local cache and returned to the user. Otherwise, MeanCache forwards the query to the LLM web service to obtain the response. The query and its response and embeddings are then stored in the cache.

MeanCache is a python-based application that is built on the Flower FL framework [34]. A user can submit LLM queries via this application to take advantage of the local cache. The central server, which orchestrates the FL training, may reside on the LLM web service. We employ the Sbert [158] library to train MPNet [172] and Albert [109] on each client and to generate query embeddings. To efficiently execute a cosine similarity search between a query embedding and cached embeddings, we utilize Sbert’s semantic search, which can handle up to 1 million entries in the cache. MeanCache cache storage is persistent and built using DiskCache [1] library.

6.4 Evaluation

Our evaluation answers the following research questions.

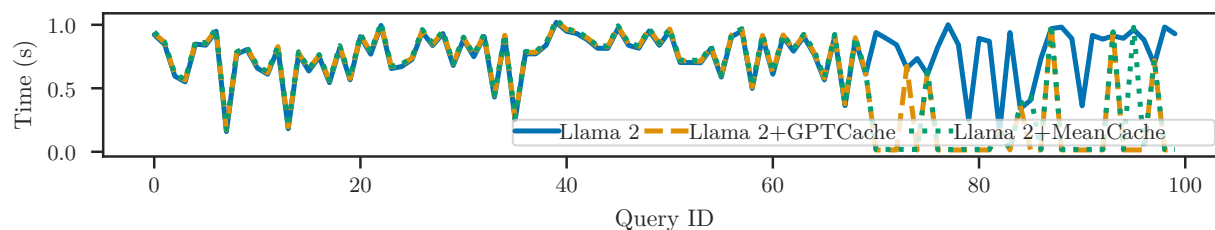


Figure 6.4: Response times of 100 randomly sampled user queries to the Llama 2-based LLM service in three scenarios: without any semantic cache, with GPTCache, and with MeanCache. The integration of a semantic cache does not add significant overhead to non-duplicate queries, meaning it does not impede the performance of the LLM-based service. Moreover, it significantly reduces the average response times for duplicate queries (70-99) by serving them from the local cache.

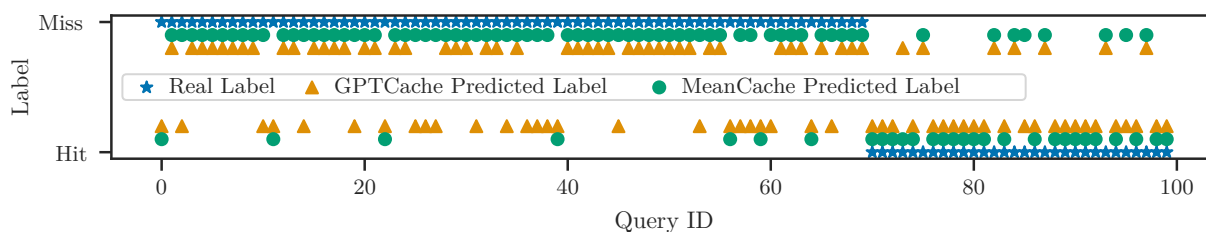


Figure 6.5: Comparison of MeanCache and GPTCache on a set of 100 queries, including 70 non-duplicate and 30 duplicate queries, sent to the Llama 2-based LLM service. Queries 0 to 69 are non-duplicate (*i.e.*, *real label is miss*), and GPTCache produces significantly higher false hits on these unique queries compared to MeanCache.

- How does MeanCache perform in comparison to baseline in terms of performance metrics (§6.4.2)?
- How accurately does MeanCache retrieve contextual queries from the cache (§6.4.3)?
- Is it possible to reduce the embedding dimension to save storage space and accelerate semantic search while outperforming the baseline (§6.4.4)?
- Is it possible to train an embedding model in a privacy-preserving manner without the centralized data (§6.4.5)?
- What effect does the cosine similarity threshold have on the performance of MeanCache

Table 6.1: MeanCache outperforms GPTCache (baseline) on both standalone and contextual queries.

Metrics	Standalone Queries			Contextual Queries	
	GPT-Cache	Mean-Cache (MPNet)	Mean-Cache (Albert)	GPT-Cache	MeanCache
F score	0.56	0.73	0.68	0.67	0.93
Precision	0.52	0.72	0.66	0.66	0.98
Recall	0.85	0.78	0.77	0.71	0.79
Accuracy	0.72	0.85	0.81	0.61	0.86

(§6.4.6)?

- GPTCache [29] suggests using Llama 2 to generate embeddings to improve semantic matching. Is it feasible to use Llama 2 to compute embeddings at the user side for semantic matching (§6.4.7)?

6.4.1 Evaluation Settings

We conduct evaluations of MeanCache against the baseline [29] to demonstrate that MeanCache achieves optimal performance while preserving user-privacy (*i.e.*, without storing the user queries at the server). For a fair comparison between MeanCache and baseline, we employ the optimal configuration as described in the GPTCache study [29]. This configuration utilizes Albert [109] and sets the cosine similarity threshold at 0.7 to determine the cache hit or miss.

Transformer Models and Datasets. For extensive evaluations of MeanCache, we utilize the Llama 2 [184], MPNet [172], and Albert [109] transformer models to generate embeddings.

We evaluate MeanCache using the GPTCache dataset. The dataset is partitioned into

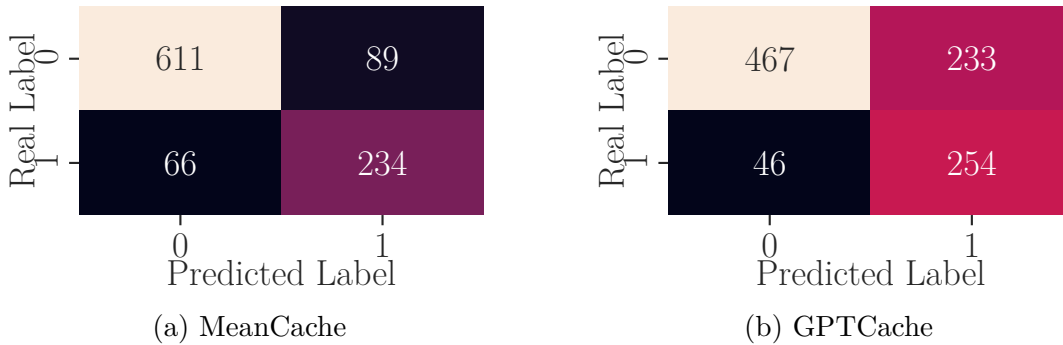


Figure 6.6: Confusion matrices for MeanCache and GPTCache on 1000 queries comprising 700 unique and 300 duplicate queries. Among the 700 unique queries, MeanCache produces only 89 false hits, while GPTCache generates 233 false hits.

training, testing, and validation subsets. The training and validation datasets are randomly distributed among the clients, each receiving non-overlapping data points. During local training, each client utilizes its training dataset to update its local embedding model and employs the validation dataset to determine the optimal threshold for cosine similarity (Section 6.4.6). The testing dataset, located at the central server, facilitates a fair comparison between MeanCache and GPTCache [29]. Since there does not exist any dataset of contextual queries, we generate a synthetic dataset using GPT-4 consisting of 450 queries, including duplicates, non-duplicates, and contextual queries, to evaluate MeanCache performance on contextual queries.

Experimental Setup. The experiments are conducted on a high-performance computing cluster, equipped with 128 cores, 504 GB of memory, and four A100 Nvidia GPUs, each with 80 GB of memory. We utilize the Flower FL [34] library to simulate a FL setup. Additionally, the SBERT [158] library is employed to train the embedding model on each client. The number of clients participating in FL training are 20. The number of clients is restricted due to the limited size of the GPTCache dataset, which is inadequate for distribution among hundreds of clients. However, we believe MeanCache results are not influenced by the number of clients, and the evaluation setup of MeanCache is consistent with the evaluation standard

in FL [24, 193].

Evaluation Metrics. In caching systems, the efficacy has traditionally been gauged by cache hit-and-miss rates. A cache hit implies the data or query is retrieved from the cache, whereas a cache miss indicates the opposite. Semantic caching introduces a nuanced classification: true and false hits, alongside true and false misses. A *true hit* signifies a correct match between a query and a similar cached query, whereas a *false hit* is an incorrect match with a non-similar cached query. A *true miss* signifies when a query does not have a similar cached query, whereas a *false miss* is when a query has a similar cached query but is not received from the cache. Thus, traditional hit/miss metrics are potentially misleading in semantic caches. For example, a query might incorrectly match with an irrelevant cached query (deemed a hit traditionally) due to semantic matching. We adopt precision, recall, F score, and accuracy for a comprehensive evaluation of MeanCache against baseline. These metrics are defined as follows:

Precision. The ratio of true positive hits to all positive hits (including both true positives and false positives). In semantic caching, this measures how many of the queries matched to a cached query are correctly matched. $\text{Precision} = \frac{TP}{TP+FP}$ where TP represents true positives (true hits) and FP represents false positives (false hits).

Recall. The ratio of true positive hits to all relevant items (including both true positives and false negatives). In semantic caching, this assesses the proportion of correctly matched queries out of all queries that should have been matched to a cached query. $\text{Recall} = \frac{TP}{TP+FN}$ where FN represents false negatives (false misses).

F_β Score. A weighted harmonic mean of precision and recall, balancing the two based on

the value of β . $\beta > 1$ gives more weight to recall, while $\beta < 1$ emphasizes precision.

$$F_\beta = (1 + \beta^2) \cdot \frac{\text{Precision} \times \text{Recall}}{(\beta^2 \times \text{Precision}) + \text{Recall}}$$

Accuracy. The ratio of correctly identified queries (both true hits and true misses) to all queries. $\text{Accuracy} = \frac{TP+TN}{TP+TN+FP+FN}$ where TN represents true negatives (true misses).

6.4.2 MeanCache Comparison with Baseline

We evaluate MeanCache against GPTCache, a baseline semantic cache, to assess improvements in precision, recall, and F score. MeanCache FL model training is discussed in Section 6.4.5, and the optimal threshold is covered in Section 6.4.6.

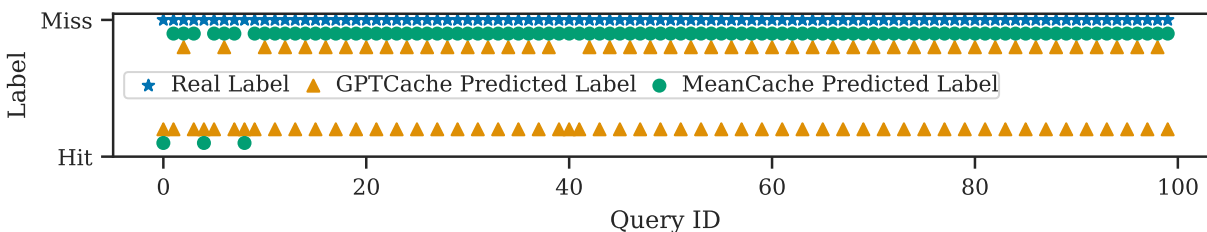
We select a sample of 1000 queries, 30% of which are repeated queries (*i.e.*, 300 queries are repeated), and load these 1000 queries as cached queries. Note that repeated queries are usually fewer than non-repeated queries. Thus, we use 30% as repeated queries, a similar percentage previously observed for web services [133].

Initially, we send a new set of one thousand queries to Llama 2 (*i.e.*, without any semantic cache) to establish a baseline for response times. We limit responses to 50 tokens to reflect practical response sizes, although actual sizes can be much larger. Note that MeanCache’s performance is not dependent on the response as it only matches the queries. Next, we send these queries to Llama 2 based local LLM service with MeanCache and GPTCache to measure the response times and performance metrics (e.g., precision, recall, F-score) respectively. An analysis of a random subset of 100 queries (70 non-duplicate and 30 duplicate) from the 1000 queries shows the impact of caching on response times in Figure 6.4 and cache hit and miss rates in Figure 6.5. The y-axis in Figure 6.4 shows the response time of each

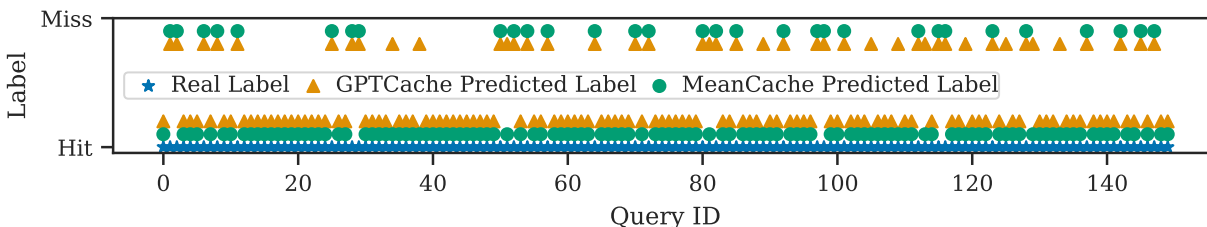
query without any cache (Llama 2), with GPTCache (Llama 2 + GPTCache), and with MeanCache (Llama 2 + MeanCache). In Figure 6.5, x-axis represents the query and the y-axis represents each semantic cache hit and miss alongside the real label. Figure 6.4 demonstrates that implementing a semantic cache does not impede the performance (queries ranging from 0 to 69) and improves the user experience as response times reduce on duplicate queries (queries 70 to 99).

However, Figure 6.5 shows that GPTCache produces significant false hits on non-duplicate queries (queries 0 to 69) compared to MeanCache. Each false hit means the user receives an incorrect query response from the cache and must resend the same query to LLM service, resulting in a poor user experience. To prioritize precision, we adjust the beta value in the F score to 0.5, valuing precision twice as highly as recall to ensure user satisfaction by avoiding false positives. This decision is driven by the need to minimize user inconvenience caused by incorrect cache hits. A false miss (low recall) does not require user intervention as the false miss query will be automatically routed to the LLM. Thus, precision in semantic caching is more important than recall.

Table 6.1 and the confusion matrices in Figure 6.6 highlight MeanCache’s superior performance over GPTCache on the 1000 user queries. Notably, MeanCache with MPNet achieves a precision of 0.72, significantly surpassing GPTCache’s 0.52. This superiority is evident in the lower false positive rates (i.e., false hits) shown in Figure 6.6a. The number of false hits for MeanCache is 89 (Figure 6.6a), while GPTCache has 233 false hits, as depicted in Figure 6.6b. In practical terms, this means that with GPTCache, the end user has to manually resend 233 queries to the LLM service to get the correct responses, compared to only 89 queries with MeanCache. While GPTCache’s recall is higher than MeanCache’s, as we discussed earlier, precision is significantly more important than recall, and MeanCache outperforms GPTCache in this regard. Overall, the F-score of MeanCache with MPNet is



(a) Ideally, all 100 queries should result in misses. However, GPTCache incorrectly produces 54 *false hits*, while MeanCache yields only 3.



(b) MeanCache yields 8% more *true hits* than the baseline.

Figure 6.7: Performance on Contextual Queries: MeanCache vs. Baseline. (a) reports MeanCache’s fewer false hits 3 vs. 54 of GPTCache. (b) reports higher true hits by MeanCache.

0.73 and 0.68 with the Albert embedding model, both of which outperform GPTCache’s F-score of 0.56.

Takeaway. In an end-to-end deployment, MeanCache significantly outperforms GPTCache. It demonstrates a 17% higher F score and a 20% increase in precision in optimal configuration. The substantial reduction in false cache hits enhances the end-user experience.

6.4.3 Contextual Queries

The end-user interaction with an LLM service mainly involves two types of user queries: standalone (*e.g.*, Q1 *Draw a line in Python?*) and follow-up or contextual queries (*e.g.*, Q2 *Change the color to red*). While standalone queries can be processed independently, contextual queries require prior context for accurate responses. Suppose that both Q1 and Q2 with their responses are cached. The user sends a new query, Q3 *Draw a circle?*, followed

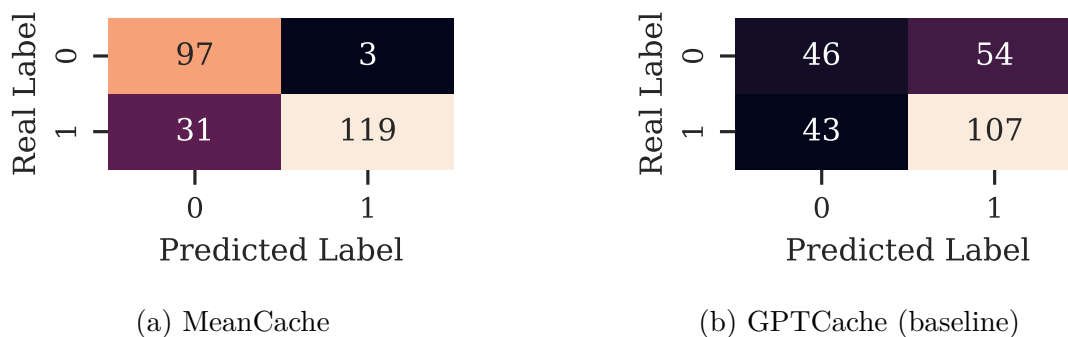


Figure 6.8: MeanCache reports three false hits compared to 54 false hits by GPTCache.

by Q4 *Change the color to red*. Q4 is semantically similar to Q2 in the cache but refers to a different context. GPTCache incorrectly identifies this as a cache hit, leading to an inaccurate response. MeanCache addresses this issue by verifying the context chain (Algorithm 6) of semantically matched queries. This approach ensures that contextual queries (Q4) result in a *true cache miss*, prompting a request to the LLM service for an appropriate response.

On the dataset of 450 contextual queries (see Section 6.4.1), first, we populate the cache (MeanCache and baseline) with 200 queries (100 standalone and 100 contextual queries of the standalone queries). Next, we send 150 duplicate queries (75 standalone + 75 contextual) and 100 non-duplicate queries (a total of 250 queries) to the cache-enabled LLM. Figure 6.7 shows the true label (whether the query should be returned from the cache or not) and the corresponding GPTCache and MeanCache performance (predicted label). Note that in Figure 6.7a, all the queries should be answered by the LLM; in other words, there should be no hits. However, GPTCache has 54 false hits, while MeanCache has only three false hits. This is also shown in the confusion matrix in Figure 6.8. Table 6.1 (Column-3) summarizes the comparative results. MeanCache outperforms GPTCache by over 25% in both F-Score and accuracy. Additionally, MeanCache achieves 32% higher precision compared to the baseline.

Takeaway. GPTCache’s low performance stems from its inability to consider contextual

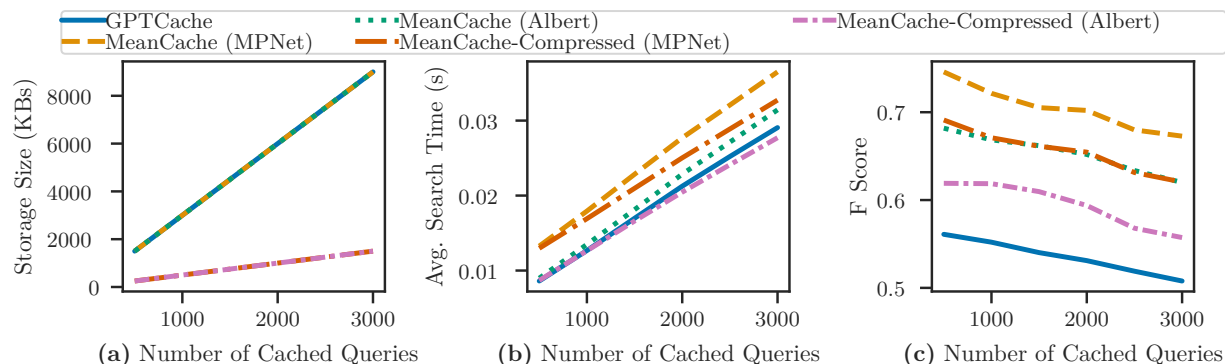


Figure 6.9: (a) MeanCache’s embedding compression reduces storage by 83% compared to GPTCache. (b) MeanCache’s semantic matching speed is 11% faster with compression enabled, while still outperforming GPTCache. (c) MeanCache’s F score is slightly lower with compression enabled, but it still outperforms GPTCache.

information, leading to high false hit rates. MeanCache demonstrates superior handling of contextual queries, with a 25% improvement in accuracy over the baseline.

6.4.4 Embedding Compression and Impact on Storage Space

Clients or users often face storage limitations compared to web servers. Storing embeddings in the local cache on the user side for semantic search demands memory storage. Various models yield embeddings with differing vector sizes; for example, the MPNet and Albert models produce an output embedding vector of 768 dimensions, whereas the Llama 2 model’s embeddings dimension size is 4096. The embedding vector size also affects semantic search speeds, where smaller vectors could enhance speed and lower resource demands.

Figure 6.9 illustrates the effects of MeanCache dimension reduction utility on the storage, semantic matching speed (overhead), and MeanCache’s performance (F score). The x-axis indicates the number of queries stored in the cache, while the y-axis shows storage size, average search time, and the F score in respective graphs. MeanCache-Compressed (MPNet) and MeanCache-Compressed (Albert) represent instances where MeanCache decreases

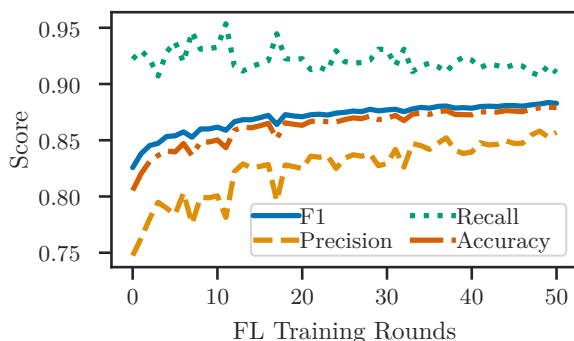


Figure 6.10: *MPNet*'s FL training helps generate high-quality embeddings.

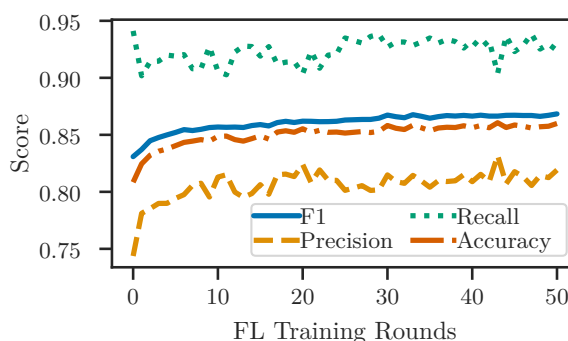


Figure 6.11: FL training boosts MeanCache's query matching precision with *Albert*.

the embedding dimensions from 768 to 64 by employing the compression, as detailed in MeanCache design (Section 6.3).

Figure 6.9 demonstrates that increased stored queries linearly raise storage needs. Yet MeanCache with compression enabled drastically lowers storage needs by 83% compared to GPTCache. Figure 6.9 also indicates that compression decreases the average search time, with MeanCache enabled compression approximately 11% faster. Moreover, despite a slight decrease in F score with compression enabled, MeanCache still surpasses GPTCache. Furthermore, given the evidence from Section 6.4.5 (Figures 6.10 and 6.11) that *MPNet* produces more precise embeddings, and it is also clear from Figure 6.9 that *MPNet*'s embeddings are particularly resilient to compression and excel in semantic matching.

Takeaway. The application of embedding compression optimization in MeanCache offers substantial benefits, including an 83% savings in storage and an 11% faster search process, while still outperforming the established baseline (GPTCache).

6.4.5 Privacy Preserving Embeddings Model Training

Storing clients queries on the server side presents a potential privacy risk. To address this, each client can retain its local data on its own device. The ensuing challenge is how to train an embedding model that can also utilize the distributed data from all clients. FL is recognized for training neural networks in a privacy-preserving manner. As such, MeanCache employs FL to train and fine-tune an embedding model, thereby preserving privacy and leveraging the dataset residing on the client’s side. In this section, our objective is to evaluate whether FL training can progressively enhance the embedding model to generate high-quality embeddings for user queries. To simulate this scenario, we distribute the training dataset among 20 clients. In each round, we sample 4 clients, conducting a total of 50 FL training rounds. Each client trains its embedding model for 6 epochs, operating on a dedicated A100 GPU. We conduct two experiments with the Albert and MPNet models. The batch size is set to 256 for the Albert model, and for MPNet, it is set to 128 during the local training by participating clients.

Figures 6.10 and 6.11 depict the performance of MeanCache as FL training progresses. The x-axis represents the training round, while the y-axis shows the performance metrics such as F-score, precision, recall, and accuracy of the global model (W_{global}^{t+1}). As illustrated in Figure 6.10, the F-score for MPNet increases from 0.82 to 0.88, and for Albert, it rises from 0.83 to 0.86, as shown in Figure 6.11. Similarly, precision for MPNet significantly increases from 0.74 to 0.85, as depicted in Figure 6.10, and for Albert, it increases from 0.74 to 0.81, as demonstrated in Figure 6.11. Given that MPNet is a more robust transformer architecture compared to Albert, it is also observed during our training that MPNet outperforms Albert, exhibiting superior learning in FL settings.

Takeaway. FL training increases 11% precision of MeanCache for MPNet and a 7% increase

for Albert. The performance of the embedding model to generate high-quality embeddings can improve in a privacy-preserving manner using FL training.

6.4.6 Cosine Similarity Threshold Impact on Semantic Matching

Semantic matching for a new user query begins by generating the embeddings of the user’s query (E_q) using the embedding model. The cosine similarity (θ) is computed with the cached embeddings. If the cosine similarity θ exceeds the threshold τ , the cache is hit and the response to the user query is returned from the cache. Therefore, the cosine similarity threshold τ is crucial in determining the similarity between a user query and cached entries. A low threshold value of τ can lead to false hits (incorrect matches), while a high threshold might overlook the appropriate matches (*i.e.*, false cache misses or false negatives).

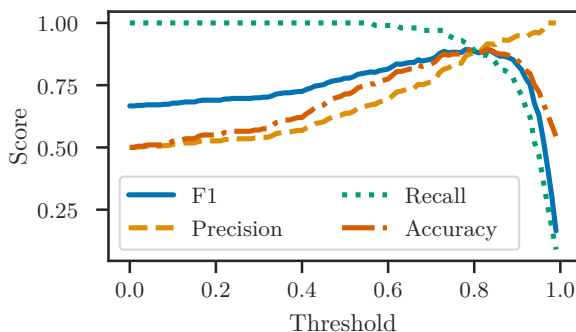


Figure 6.12: MeanCache automatically optimizes MPNet’s threshold.

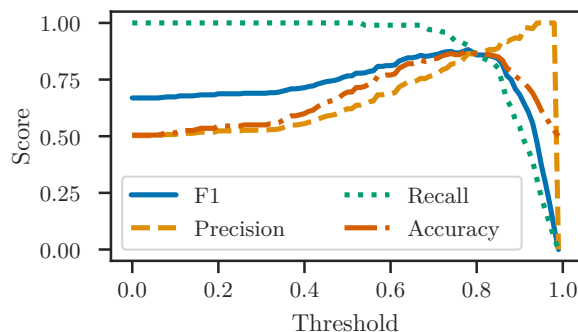


Figure 6.13: MeanCache identifies an optimal threshold of 0.78 for Albert.

To illustrate this, MeanCache varies the threshold τ from 0 to 9 and evaluates the performance metrics F-score, precision, recall, and accuracy with an equal distribution of duplicate and non-duplicate queries from the validation data to avoid bias. Figures 6.12 and 6.13 show how the cosine similarity threshold (τ) affects MeanCache’s performance. The x-axis represents the threshold τ values, and the y-axis denotes the performance metrics. For instance, at a 0.3 threshold, MeanCache’s semantic matching accuracy using MPNet is 57%, with a

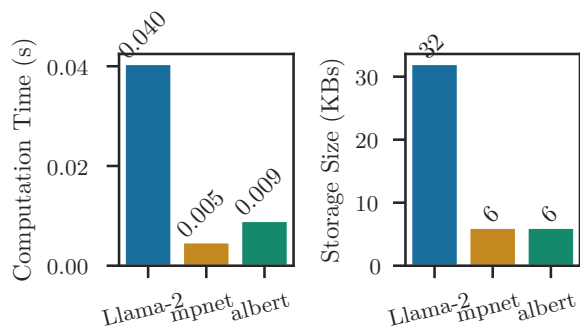


Figure 6.14: Llama 2 takes significantly longer to compute embeddings and requires substantially more storage space than Albert and MPNet.

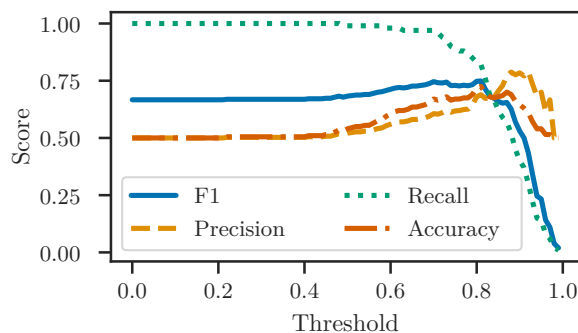


Figure 6.15: Llama 2 does not perform well in semantic matching, even at optimal thresholds.

precision of 54% as shown in Figure 6.12. Similarly, with Albert at the same threshold, the accuracy is 55%, and the precision is 53% (Figure 6.13). Precision typically improves with an increase in threshold. However, beyond a certain point, increasing the threshold τ leads to a decline in F score, accuracy, and recall.

For MPNet, the optimal threshold τ is identified at 0.83, achieving an F1 score of 0.89, precision of 0.92, and accuracy of 0.90 (Figure 6.12). For Albert, the optimal threshold is 0.78, with an F1 score of 0.88.

Takeaway. GPTCache’s suggested threshold of 0.7 will result in suboptimal performance during semantic matching. The optimal threshold τ values varies with the embedding model. MeanCache optimally adjusts the threshold based on user data, outperforming GPTCache’s suggested threshold by 16% in precision and 4% in F score for MPNet, and by 10% in precision and 2% in F score for Albert.

6.4.7 Infeasibility of Embedding Generation with Llama 2

GPTCache [29] recommend using Llama for generating embeddings to enhance GPTCache’s performance. However, Llama 2’s embedding computation is expensive in terms of inference time, requires substantial storage, and incurs considerable overhead during semantic searches. For example, Llama 2 with its 7 billion parameters, demands 30 GB of memory [3], whereas Albert and MPNet require only 43 MB and 420 MB, respectively.

To highlight the impracticality to generate query embeddings with Llama 2, we compare the embedding computation time, embedding storage space requirement of Llama 2, Albert and MPNet transformer models. Figure 6.14 shows the average time to compute the embeddings of a single query and storage requirements for the embeddings. Figure 6.14 shows that the average embedding computation time of Llama 2 is 0.04 seconds, while for Albert and MPNet the average computation time is 0.005 and 0.009 seconds respectively. Single query embeddings generated from Llama 2 takes approximately 32 KBs of space and embeddings generated by both MPNet and Albert only take 6 KBs, as shown in Figure 6.14.

Furthermore, we evaluate the performance of the Llama 2 on embedding generation and semantic matching. Figure 6.15 shows that the performance of Llama 2 with different cosine similarity threshold (τ) and corresponding performance metric. We can see that the performance of Llama 2 is not good even with the optimal cosine similarity threshold, as also noted by researchers [4]. The maximum F1 score achieved by Llama 2 is 0.75 which is quite low when compared with the optimal thresholds scores from the Figures 6.12 and Figure 6.13.

Smaller models tailored for specific tasks often surpass larger models in efficiency [57, 145, 152]. Thus, diverging from GPTCache’s approach, we advocate for adopting smaller yet efficient embedding models for semantic caching. These models not only ensure optimal

performance but also minimize the semantic cache’s overhead on users, featuring lower inference demands and reduced output embedding sizes, thereby facilitating deployment on edge devices.

Takeaway. Llama 2 is not viable for generating embeddings. Future enhancements might improve its performance to generate embeddings, but the computational demands, semantic search duration, and storage requirements will likely remain elevated.

6.5 Summary

MeanCache introduces the first user-centric semantic cache designed for LLM-based web services, such as ChatGPT. In MeanCache, clients collaboratively train a global embedding model using FL on their local data, ensuring user privacy. After aggregation, the global model produces high-quality embeddings for effective semantic matching. When a new query from the user matches a previous one, MeanCache semantically compares it with the user’s local cache and retrieves the most relevant results. This approach reduces the computational cost of LLM services, enhances bandwidth and latency, and conserves the user’s query quota. Even with compressed embeddings that save 83% of storage space, MeanCache outperforms existing baseline. With its distributed cache design, MeanCache offers a solution to reduce up to one-third of LLM query inference costs for semantically similar queries on the user side.

Chapter 7

Conclusion

Centralized machine learning (ML) benefits from a rich ecosystem of debugging and interpretability techniques that help developers understand model behavior, identify faulty training data, and attribute predictions to their sources, owing to transparent access to all training data. However, Federated Learning (FL), a distributed ML paradigm where multiple clients collaboratively train a global model without sharing raw data, removes this transparency (i.e., data access), rendering traditional debugging and interpretability techniques ineffective. As a result, debugging and interpretability remain open challenges in FL [97]. Unlike centralized ML, where developers can inspect training data, FL’s distributed architecture and strict privacy constraints limit access to client data. This makes it difficult to identify faulty clients, attribute predictions to their sources, and understand why a global model produces specific outputs.

The central hypothesis of this dissertation is that specialized, privacy-preserving methods can effectively address these challenges without accessing client data or without changing the aggregation protocol (e.g., FedAvg) or significantly impeding the FL training. The *three contributions* (FedDebug, TraceFL and ProToken) presented herein validate this hypothesis. The central insight threading through these contributions is that model parameters, activations, and gradients, information already shared or derivable in standard FL protocols (e.g., FedAvg, FedProx) contain sufficient signal to enable debugging and attribution keeping FL privacy principles intact.

FedDebug shows that systematic fault localization is possible by differentially testing neuron activations on auto-generated inputs. The key insight is that faulty clients yield models with neuron activations that diverge from benign clients on the same inputs. By adapting record-and-replay debugging to FL and comparing internal model behaviors instead of outputs, FedDebug lets developers inspect FL training states and localize faulty clients without server-side real-world test data. This method achieves high localization accuracy with minimal training time overhead.

TraceFL advances fault localization to fine-grained client attribution by introducing *neuron provenance*, which decouples data provenance from data access. By formalizing neuron provenance, TraceFL shows that data influence can be tracked even when the data itself is hidden. Leveraging the linearity of weighted averaging in FL aggregation, TraceFL makes individual client contributions traceable. For any prediction, it identifies influential neurons via gradient-based importance weighting and decomposes them to their client-specific origins, yielding an actionable, ranked list of responsible clients. This is valuable both when the global model is correct (for understanding and rewarding contributions) and when it mispredicts (for identifying likely sources of corruption). TraceFL is effective across architectures like CNNs and Transformers, and remains robust under non-IID data and differential privacy noise.

ProToken extends the provenance paradigm to federated LLMs, addressing the unique challenges of autoregressive generation and computational tractability at the billion-parameter scale. Attribution in federated LLMs is complicated by the difficulty of disentangling pre-trained knowledge from federated contributions, as well as by cascading dependencies between generated tokens. To resolve these challenges, ProToken introduces a token-level provenance mechanism. It computes per-token provenance at targeted transformer layers, specifically the output projection and final MLP layers. This approach is computationally ef-

ficient and achieves high attribution accuracy, enabling precise identification of which client is responsible for a generated textual response given an input. ProToken’s effectiveness holds across multiple LLM architectures and domains. *In each case (FedDebug, TraceFL, or ProToken), the solution operates entirely on information available at the aggregator, such as client model weights, and requires no client-side instrumentation or modifications to the FL protocol.*

7.1 Impact

These techniques have immediate *practical implications*. Integrating FedDebug into the Flower FL framework as a baseline makes systematic debugging accessible to practitioners deploying FL in production. TraceFL and ProToken enable accountability in collaborative learning, allowing organizations to trace predictions to contributing clients for fair reward allocation, quality assurance, or forensic analysis of mispredictions. All artifacts produced during this dissertation are public and fully executable on Google Colab with a single command, which fosters future research in this area. In addition, follow-up work on MeanCache leads to the development of a highly downloadable embedding model on Hugging Face that outperforms closed-source models on semantic caching tasks.

7.2 Future Work

Several promising directions remain unexplored. The techniques presented here focus primarily on horizontal FL with synchronous aggregation; extending neuron provenance to asynchronous or fully decentralized settings would broaden applicability. Current attribution methods identify responsible clients but not specific training samples within those

clients. Developing privacy-preserving mechanisms for finer-grained data attribution would provide more actionable debugging insights. As the field shifts toward multi-modal LLMs such as vision-language LLMs, extending provenance to trace how distributed responsibility across image and text inputs from different clients presents both technical and conceptual challenges. Additionally, ProToken assumes full model fine-tuning; its applicability to parameter-efficient fine-tuning methods such as LoRA, where only low-rank adapter weights are updated, remains unexplored.

The robustness of interpretability and debugging techniques under adversarial manipulation, where malicious clients might craft updates specifically to evade provenance detection or fault localization, warrants investigation for security-critical deployments. While TraceFL demonstrated initial robustness under a specific DP mechanism [135], comprehensive evaluation of FedDebug, TraceFL, and ProToken under diverse differential privacy settings remains an important direction for privacy-critical deployments.

The diagnostic tools presented in this dissertation operate in a human-in-the-loop manner, requiring developers to interpret signals and take corrective action. A natural evolution is toward a self-healing FL aggregator that automatically down-weights or unlearns contributions from clients identified as faulty, enabling autonomous resilience without halting training rounds. Additionally, while TraceFL and ProToken quantify client contributions accurately, translating attribution into economic valuation remains an open problem. Leveraging provenance scores to approximate fair contribution metrics could enable incentive mechanisms that reward clients based on the utility of their contributions rather than mere participation.

While investigating debugging and interpretability in federated learning, we observe that non-IID data distributions may not present the same challenges for federated LLMs as they do for CNN-based FL. In traditional FL scenarios, non-IID partitions, such as each client

holding data from only one class, pose significant difficulties for global model learning, especially in image classification tasks like CIFAR-10. In contrast, language data inherently exhibits structural regularities that transcend domain boundaries. Common tokenization patterns, shared grammatical structures, and ubiquitous words (e.g., “is”, “a”, “the”, spaces, punctuation) create a form of data symmetry across clients, even when their domains differ substantially. We hypothesize that this linguistic symmetry enables standard aggregation algorithms like FedAvg to remain effective for federated LLMs, even under extreme non-IID conditions where clients possess entirely distinct domain data, such as medical, financial, or legal. Exploring this hypothesis and identifying when simpler FL algorithms suffice for language models is a promising direction for expanding the deployment of FL in naturally siloed organizational environments.

Summary. This dissertation demonstrates that, although the debugging and interpretability challenges introduced by FL’s distributed nature are significant, they are addressable without violating FL training principles. By analyzing model parameters, activations, and gradients, and by leveraging the mathematical structure of FL aggregation, we can gain deep insights into global model behavior without accessing clients’ data. The techniques and insights established here lay the groundwork for building interpretable, debuggable, and accountable federated learning systems, paving the way for FL’s continued expansion into critical domains such as healthcare and finance.

Bibliography

- [1] Diskcache: Disk backed cache — diskcache 5.6.1 documentation. <https://grantjenks.com/docs/diskcache/>. (Accessed on 04/01/2024).
- [2] Nvidia dgx a100 | the universal system for ai infrastructure. <https://images.nvidia.com/aem-dam/Solutions/Data-Center/nvidia-dgx-a100-datasheet.pdf>.
- [3] Sizing guide - nvidia docs. <https://docs.nvidia.com/ai-enterprise/workflows-generative-ai/0.1.0/sizing-guide.html>. (Accessed on 01/25/2024).
- [4] [user] embedding doesn't seem to work? · issue #899 · ggerganov/llama.cpp. <https://github.com/ggerganov/llama.cpp/issues/899>. (Accessed on 01/18/2024).
- [5] Arc max is the popular browser's new suite of ai tools - the verge. <https://www.theverge.com/2023/10/3/23898907/arc-max-ai-browser-mac-ios>. (Accessed on 01/19/2024).
- [6] Google ai updates: Bard and new ai features in search. <https://blog.google/technology/ai/bard-google-ai-search-updates/>. (Accessed on 01/18/2024).
- [7] Microsoft edge and bing users engage in over 1.9 billion copilot chats in 2023. <https://www.msn.com/en-us/money/other/microsoft-edge-and-bing-users-engage-in-over-19-billion-copilot-chats-in-2023/ar-AA1mmrxZ>. (Accessed on 01/22/2024).
- [8] Introducing chatgpt. <https://openai.com/blog/chatgpt>. (Accessed on 01/18/2024).
- [9] Introducing claude 2.1 \ anthropic. <https://www.anthropic.com/news/claude-2-1>. (Accessed on 01/18/2024).

- [10] Gptcache/examples/benchmark at main · zilliztech/gptcache. <https://github.com/zilliztech/GPTCache/tree/main/examples/benchmark>. (Accessed on 03/04/2024).
- [11] Federated learning: Collaborative machine learning without centralized training data – google research blog. <https://blog.research.google/2017/04/federated-learning-collaborative.html>. (Accessed on 04/01/2024).
- [12] OpenAI Pricing. <https://openai.com/pricing>. (Accessed on 01/19/2024).
- [13] Perplexity pro. <https://www.perplexity.ai/pro>. (Accessed on 03/01/2024).
- [14] Learning human actions on computer applications. <https://www.rabbit.tech/research>. (Accessed on 01/19/2024).
- [15] How Apple personalizes Siri without hoovering up your data — technologyreview.com. <https://www.technologyreview.com/2019/12/11/131629/apple-ai-personalizes-siri-federated-learning/>. [Accessed 03-01-2025].
- [16] zilliztech/gptcache: Semantic cache for llms. fully integrated with langchain and llama_index. <https://github.com/zilliztech/gptcache>. (Accessed on 03/03/2024).
- [17] Reduan Achtibat, Sayed Mohammad Vakilzadeh Hatefi, Maximilian Dreyer, Aakriti Jain, Thomas Wiegand, Sebastian Lapuschkin, and Wojciech Samek. Attnlrp: attention-aware layer-wise relevance propagation for transformers. In *Proceedings of the 41st International Conference on Machine Learning, ICML'24*. JMLR.org, 2024.
- [18] Sherif Akoush, Ripduman Sohan, and Andy Hopper. HadoopProv: Towards Provenance as a First Class Citizen in MapReduce. In *TaPP*, 2013.
- [19] Haider Ali, Dian Chen, Matthew Harrington, Nathaniel Salazar, Mohannad Al Ameedi, Ahmad Faraz Khan, Ali R Butt, and Jin-Hee Cho. A survey on attacks and their

- countermeasures in deep learning: Applications in deep neural networks, federated, transfer, and deep reinforcement learning. *IEEE Access*, 11:120095–120130, 2023.
- [20] Loubna Ben allal, Anton Lozhkov, Elie Bakouch, Gabriel Martin Blazquez, Guilherme Penedo, Lewis Tunstall, Andrés Marafioti, Agustín Piqueres Lajarín, Hynek Kydlíček, Vaibhav Srivastav, Joshua Lochner, Caleb Fahlgren, Xuan Son NGUYEN, Ben Burtenshaw, Clémentine Fourrier, Haojun Zhao, Hugo Larcher, Mathieu Morlon, Cyril Zakka, Colin Raffel, Leandro Von Werra, and Thomas Wolf. SmolLM2: When smol goes big — data-centric training of a fully open small language model. In *Second Conference on Language Modeling*, 2025. URL <https://openreview.net/forum?id=3JiCl2A14H>.
- [21] Yael Amsterdamer, Susan B. Davidson, Daniel Deutch, Tova Milo, Julia Stoyanovich, and Val Tannen. Putting Lipstick on Pig: Enabling Database-Style Workflow Provenance. *Proc. VLDB Endow.*, 5(4):346–357, dec 2011. ISSN 2150-8097. doi: 10.14778/2095686.2095693.
- [22] M. Arnold, R. K. E. Bellamy, M. Hind, S. Houde, S. Mehta, A. Mojsilović, R. Nair, K. Natesan Ramamurthy, A. Olteanu, D. Piorkowski, D. Reimer, J. Richards, J. Tsay, and K. R. Varshney. Factsheets: Increasing trust in ai services through supplier’s declarations of conformity. *IBM Journal of Research and Development*, 63(4/5):6:1–6:13, 2019. doi: 10.1147/JRD.2019.2942288.
- [23] Hilal Asi, Vitaly Feldman, and Kunal Talwar. Optimal Algorithms for Mean Estimation under Local Differential Privacy. In *International Conference on Machine Learning*, pages 1046–1056. PMLR, 2022.
- [24] Dmitrii Avdiukhin and Shiva Kasiviswanathan. Federated learning under arbitrary communication patterns. In *International Conference on Machine Learning*, pages 425–435. PMLR, 2021.

- [25] Ricardo Baeza-Yates and Felipe Saint-Jean. A three level search engine index based in query log distribution. In *String Processing and Information Retrieval: 10th International Symposium, SPIRE 2003, Manaus, Brazil, October 8-10, 2003. Proceedings 10*, pages 56–65. Springer, 2003.
- [26] Ricardo Baeza-Yates, Aristides Gionis, Flavio Junqueira, Vanessa Murdock, Vassilis Plachouras, and Fabrizio Silvestri. The impact of caching on search engines. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 183–190, 2007.
- [27] Eugene Bagdasaryan, Andreas Veit, Yiqing Hua, Deborah Estrin, and Vitaly Shmatikov. How to backdoor federated learning. In *International Conference on Artificial Intelligence and Statistics*, pages 2938–2948. PMLR, 2020.
- [28] Borja Balle and Yu-Xiang Wang. Improving the Gaussian Mechanism for Differential Privacy: Analytical Calibration and Optimal Denoising. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 394–403. PMLR, 10–15 Jul 2018. URL <https://proceedings.mlr.press/v80/balle18a.html>.
- [29] Fu Bang. Gptcache: An open-source semantic cache for llm applications enabling faster answers and cost savings. In *Proceedings of the 3rd Workshop for Natural Language Processing Open Source Software (NLP-OSS 2023)*, pages 212–218, 2023.
- [30] Xianglin Bao, Cheng Su, Yan Xiong, Wenchao Huang, and Yifei Hu. FLChain: A Blockchain for Auditable Federated Learning with Trust and Incentive. In *2019 5th International Conference on Big Data Computing and Communications (BIGCOM)*, pages 151–159, 2019. doi: 10.1109/BIGCOM.2019.00030.

- [31] Nathalie Baracaldo, Bryant Chen, Heiko Ludwig, and Jaehoon Amir Safavi. Mitigating poisoning attacks on machine learning models: A data provenance based approach. In *Proceedings of the 10th ACM workshop on artificial intelligence and security*, pages 103–110, 2017.
- [32] Nathalie Baracaldo, Bryant Chen, Heiko Ludwig, Amir Safavi, and Rui Zhang. Detecting Poisoning Attacks on Machine Learning in IoT Environments. In *2018 IEEE International Congress on Internet of Things (ICIOT)*, pages 57–64, 2018. doi: 10.1109/ICIOT.2018.00015.
- [33] Emily M Bender and Batya Friedman. Data statements for natural language processing: Toward mitigating system bias and enabling better science. *Transactions of the Association for Computational Linguistics*, 6:587–604, 2018.
- [34] Daniel J Beutel, Taner Topal, Akhil Mathur, Xinchu Qiu, Titouan Parcollet, Pedro PB de Gusmão, and Nicholas D Lane. Flower: A friendly federated learning research framework. *arXiv preprint arXiv:2007.14390*, 2020.
- [35] Arjun Nitin Bhagoji, Supriyo Chakraborty, Prateek Mittal, and Seraphin Calo. Analyzing federated learning through an adversarial lens. In *International Conference on Machine Learning*, pages 634–643. PMLR, 2019.
- [36] Abhishek Bhowmick, John Duchi, Julien Freudiger, Gaurav Kapoor, and Ryan Rogers. Protection Against Reconstruction and Its Applications in Private Federated Learning. *arXiv preprint arXiv:1812.00984*, 2018.
- [37] Battista Biggio, Blaine Nelson, and Pavel Laskov. Poisoning attacks against support vector machines. In *Proceedings of the 29th International Conference on International Conference on Machine Learning*, pages 1467–1474, 2012.

- [38] Patrick Bilic, Patrick Christ, Hongwei Bran Li, Eugene Vorontsov, Avi Ben-Cohen, Georgios Kaissis, Adi Szeskin, Colin Jacobs, Gabriel Efrain Humpire Mamani, Gabriel Chartrand, et al. The liver tumor segmentation benchmark (lits). *Medical Image Analysis*, 84:102680, 2023.
- [39] Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnes, and Bernhard Seefeld. Prochlo: Strong Privacy for Analytics in the Crowd. In *Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17*, page 441–459, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350853. doi: 10.1145/3132747.3132769. URL <https://doi.org/10.1145/3132747.3132769>.
- [40] Danushka Bollegala, Shuichi Otake, Tomoya Machide, and Ken-ichi Kawarabayashi. A Neighbourhood-Aware Differential Privacy Mechanism for Static Word Embeddings. In *Findings of the Association for Computational Linguistics: IJCNLP-AACL 2023 (Findings)*, pages 65–79, 2023.
- [41] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konečný, Stefano Mazzocchi, Brendan McMahan, et al. Towards federated learning at scale: System design. *Proceedings of machine learning and systems*, 1:374–388, 2019.
- [42] Housseem Ben Braiek and Foutse Khomh. Deepevolution: A search-based testing approach for deep neural networks. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 454–458. IEEE, 2019.
- [43] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1-7):107–117, 1998.

- [44] Sebastian Caldas, Sai Meher Karthik Duddu, Peter Wu, Tian Li, Jakub Konečný, H Brendan McMahan, Virginia Smith, and Ameet Talwalkar. Leaf: A benchmark for federated settings. *arXiv preprint arXiv:1812.01097*, 2018.
- [45] Xiaoyu Cao, Minghong Fang, Jia Liu, and Neil Zhenqiang Gong. Fltrust: Byzantine-robust federated learning via trust bootstrapping. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2021.
- [46] Ricardo Silva Carvalho, Theodore Vasiloudis, and Oluwaseyi Feyisetan. BRR: Preserving Privacy of Text Data Efficiently on Device. *arXiv preprint arXiv:2107.07923*, 2021.
- [47] Sahil Chaudhary. Code alpaca: An instruction-following llama model for code generation. <https://github.com/sahil280114/codealpaca>, 2023.
- [48] Hila Chefer, Shir Gur, and Lior Wolf. Transformer interpretability beyond attention visualization. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 782–791, 2021. doi: 10.1109/CVPR46437.2021.00084.
- [49] Huili Chen, Jie Ding, Eric William Tramel, Shuang Wu, Anit Kumar Sahu, Salman Avestimehr, and Tao Zhang. Self-aware personalized federated learning. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=EqJ5_hZSqgy.
- [50] Yiqiang Chen, Xiaodong Yang, Xin Qin, Han Yu, Biao Chen, and Zhiqi Shen. Focus: Dealing with label quality disparity in federated learning. *arXiv preprint arXiv:2001.11359*, 2020.
- [51] Yae Jee Cho, Divyansh Jhunjunwala, Tian Li, Virginia Smith, and Gauri Joshi. To

- federate or not to federate: incentivizing client participation in federated learning. In *Workshop on Federated Learning: Recent Advances and New Challenges (in Conjunction with NeurIPS 2022)*, 2022.
- [52] Liam Collins, Hamed Hassani, Aryan Mokhtari, and Sanjay Shakkottai. Exploiting shared representations for personalized federated learning. In *International Conference on Machine Learning*, pages 2089–2099. PMLR, 2021.
- [53] Harsh Bimal Desai, Mustafa Safa Ozdayi, and Murat Kantarcioglu. Blockfla: Accountable federated learning via hybrid blockchain architecture. In *Proceedings of the eleventh ACM conference on data and application security and privacy*, pages 101–112, 2021.
- [54] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *North American Chapter of the Association for Computational Linguistics*, 2019.
- [55] Bolin Ding, Janardhan Kulkarni, and Sergey Yekhanin. Collecting Telemetry Data Privately. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/253614bbac999b38b5b60cae531c4969-Paper.pdf.
- [56] Minxin Du, Xiang Yue, Sherman SM Chow, and Huan Sun. Sanitizing Sentence Embeddings (and Labels) for Local Differential Privacy. In *Proceedings of the ACM Web Conference 2023*, pages 2349–2359, 2023.
- [57] Yilun Du and Leslie Kaelbling. Compositional generative modeling: A single model is not all you need. *arXiv preprint arXiv:2402.01103*, 2024.

- [58] Cynthia Dwork, Aaron Roth, et al. The Algorithmic Foundations of Differential Privacy. *Foundations and trends® in theoretical computer science*, 9(3–4):211–407, 2014.
- [59] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. RAPPOR: Randomized Aggregatable Privacy-Preserving Ordinal Response. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS '14*, page 1054–1067, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450329576. doi: 10.1145/2660267.2660348. URL <https://doi.org/10.1145/2660267.2660348>.
- [60] Tiziano Fagni, Raffaele Perego, Fabrizio Silvestri, and Salvatore Orlando. Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Transactions on Information Systems (TOIS)*, 24(1):51–78, 2006.
- [61] Minghong Fang, Xiaoyu Cao, Jinyuan Jia, and Neil Gong. Local model poisoning attacks to Byzantine-Robust federated learning. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1605–1622. USENIX Association, August 2020. ISBN 978-1-939133-17-5. URL <https://www.usenix.org/conference/usenixsecurity20/presentation/fang>.
- [62] Giulia Fanti, Vasyl Pihur, and Úlfar Erlingsson. Building a RAPPOR with the Unknown: Privacy-Preserving Learning of Associations and Data Dictionaries. *Proceedings on Privacy Enhancing Technologies*, 3:41–61, 2016.
- [63] Oluwaseyi Feyisetan and Shiva Kasiviswanathan. Private Release of Text Embedding Vectors. In *Proceedings of the First Workshop on Trustworthy Natural Language Processing*, pages 15–27, 2021.

- [64] Ferdinando Fioretto, Pascal Van Hentenryck, and Juba Ziani. Differential Privacy Overview and Fundamental Techniques. *arXiv preprint arXiv:2411.04710*, 2024.
- [65] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. OPTQ: Accurate Quantization for Generative Pre-trained Transformers. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.
- [66] Benoît Frénay and Michel Verleysen. Classification in the presence of label noise: a survey. *IEEE transactions on neural networks and learning systems*, 25(5):845–869, 2013.
- [67] Clement Fung, Chris J. M. Yoon, and Ivan Beschastnikh. The limitations of federated learning in sybil settings. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 301–316, San Sebastian, October 2020. USENIX Association. ISBN 978-1-939133-18-2. URL <https://www.usenix.org/conference/raid2020/presentation/fung>.
- [68] Tianyu Gao, Howard Yen, Jiatong Yu, and Danqi Chen. Enabling large language models to generate text with citations. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 6465–6488, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.398. URL <https://aclanthology.org/2023.emnlp-main.398/>.
- [69] Yan Gao, Massimo Roberto Scamarcia, Javier Fernandez-Marques, Mohammad Naseri, Chong Shen Ng, Dimitris Stripelis, Zexi Li, Tao Shen, Jiamu Bai, Daoyuan Chen, et al. Flowertune: A cross-domain benchmark for federated fine-tuning of large language models. *arXiv preprint arXiv:2506.02961*, 2025.

- [70] Timnit Gebru, Jamie Morgenstern, Briana Vecchione, Jennifer Wortman Vaughan, Hanna Wallach, Hal Daumé Iii, and Kate Crawford. Datasheets for datasets. *Communications of the ACM*, 64(12):86–92, 2021.
- [71] Simos Gerasimou, Hasan Ferit Eniser, Alper Sen, and Alper Cakan. Importance-driven deep learning system testing. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 702–713, 2020.
- [72] Aritra Ghosh, Himanshu Kumar, and P. S. Sastry. Robust loss functions under label noise for deep neural networks. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI’17, page 1919–1925. AAAI Press, 2017.
- [73] Waris Gill, Ali Anwar, and Muhammad Ali Gulzar. FedDebug: Systematic Debugging for Federated Learning Applications. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE ’23, page 512–523. IEEE Press, 2023. ISBN 9781665457019. doi: 10.1109/ICSE48619.2023.00053. URL <https://doi.org/10.1109/ICSE48619.2023.00053>.
- [74] Waris Gill, Ali Anwar, and Muhammad Ali Gulzar. FedDefender: Backdoor Attack Defense in Federated Learning. In *Proceedings of the 1st International Workshop on Dependability and Trustworthiness of Safety-Critical Systems with Machine Learned Components*, SE4SafeML 2023, page 6–9, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400703799. doi: 10.1145/3617574.3617858. URL <https://doi.org/10.1145/3617574.3617858>.
- [75] Waris Gill, Ali Anwar, and Muhammad Ali Gulzar. *TraceFL: Interpretability-Driven Debugging in Federated Learning via Neuron Provenance*, page 2264–2276. IEEE Press, 2025. ISBN 9798331505691. URL <https://doi.org/10.1109/ICSE55347.2025.00128>.

- [76] Waris Gill, Mohamed Elidrisi, Pallavi Kalapatapu, Ammar Ahmed, Ali Anwar, and Muhammad Ali Gulzar. MeanCache: User-Centric Semantic Caching for LLM Web Services . In *2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1298–1310, Los Alamitos, CA, USA, June 2025. IEEE Computer Society. doi: 10.1109/IPDPS64566.2025.00117. URL <https://doi.ieeecomputersociety.org/10.1109/IPDPS64566.2025.00117>.
- [77] Waris Gill, Natalie Isak, and Matthew Dressman. Cross-service threat intelligence in LLM services using privacy-preserving fingerprints. *CoRR*, abs/2509.05608, 2025. doi: 10.48550/ARXIV.2509.05608. URL <https://doi.org/10.48550/arXiv.2509.05608>.
- [78] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. Dlfuzz: Differential fuzzing testing of deep learning systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 739–743, 2018.
- [79] Tianyu Han, Lisa C Adams, Jens-Michalis Papaioannou, Paul Grundmann, Tom Oberhauser, Alexander Löser, Daniel Truhn, and Keno K Bressemer. MedAlpaca – An Open-Source Collection of Medical Conversational AI Models and Training Data. *arXiv preprint arXiv:2304.08247*, 2023.
- [80] Fabrice Harel-Canada, Lingxiao Wang, Muhammad Ali Gulzar, Quanquan Gu, and Miryung Kim. Is neuron coverage a meaningful measure for testing deep neural networks? In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 851–862, 2020.
- [81] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers:

- Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.
- [82] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
- [83] Matthew Henderson, Rami Al-Rfou, Brian Strope, Yun-Hsuan Sung, László Lukács, Ruiqi Guo, Sanjiv Kumar, Balint Miklos, and Ray Kurzweil. Efficient natural language response suggestion for smart reply. *arXiv preprint arXiv:1705.00652*, 2017.
- [84] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.
- [85] Dan Hendrycks, Mantas Mazeika, Duncan Wilson, and Kevin Gimpel. Using trusted data to train deep networks on labels corrupted by severe noise. *Advances in neural information processing systems*, 31, 2018.
- [86] Harold Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of educational psychology*, 24(6):417, 1933.
- [87] Justin Hsu, Marco Gaboardi, Andreas Haeberlen, Sanjeev Khanna, Arjun Narayan, Benjamin C Pierce, and Aaron Roth. Differential Privacy: An Economic Method for Choosing Epsilon. In *2014 IEEE 27th Computer Security Foundations Symposium*, pages 398–410. IEEE, 2014.
- [88] <https://www.amazon.science/author/huili.chen>. Personalized federated learning for a better customer experience — amazon.science. <https://www.amazon.science/blog/personalized-federated-learning-for-a-better-customer-experience>. [Accessed 03-01-2025].

- [89] Yuzheng Hu, Fan Wu, Qinbin Li, Yunhui Long, Gonzalo Munilla Garrido, Chang Ge, Bolin Ding, David Forsyth, Bo Li, and Dawn Song. SoK: Privacy-Preserving Data Synthesis. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4696–4713, 2024. doi: 10.1109/SP54263.2024.00002.
- [90] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.
- [91] Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd Millstein, and Tyson Condie. Titian: Data provenance support in spark. In *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, volume 9, page 216. NIH Public Access, 2015.
- [92] Ji Chu Jiang, Burak Kantarci, Sema Oktug, and Tolga Soyata. Federated learning in smart city sensing: Challenges and opportunities. *Sensors*, 20(21):6230, 2020.
- [93] Lu Jiang, Di Huang, Mason Liu, and Weilong Yang. Beyond synthetic noise: Deep learning on controlled noisy labels. In *International Conference on Machine Learning*, pages 4804–4815. PMLR, 2020.
- [94] Ian T Jolliffe and Jorge Cadima. Principal component analysis: a review and recent developments. *Philosophical transactions of the royal society A: Mathematical, Physical and Engineering Sciences*, 374(2065):20150202, 2016.
- [95] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering, ICSE '02*, pages 467–477, New York, NY, USA, 2002. ACM. ISBN 1-58113-472-X. doi: 10.1145/581339.581397. URL <http://doi.acm.org/10.1145/581339.581397>.

- [96] Severin Kacianka and Alexander Pretschner. Designing accountable systems. In *Proceedings of the 2021 ACM conference on fairness, accountability, and transparency*, pages 424–437, 2021.
- [97] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. Advances and Open Problems in Federated Learning. *Foundations and Trends® in Machine Learning*, 14(1-2):1–210, 2021.
- [98] Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, Louis Rouillard, Thomas Mesnard, Geoffrey Cideron, Jean-Bastien Grill, Sabela Ramos, Edouard Yvinec, Michelle Casbon, Etienne Pot, Ivo Penchev, Gaël Liu, Francesco Visin, Kathleen Kenealy, Lucas Beyer, Xiaohai Zhai, Anton Tsitsulin, Róbert Busa-Fekete, Alex Feng, Noveen Sachdeva, Benjamin Coleman, Yi Gao, Basil Mustafa, Iain Barr, Emilio Parisotto, David Tian, Matan Eyal, Colin Cherry, Jan-Thorsten Peter, Danila Sinopalnikov, Surya Bhupatiraju, Rishabh Agarwal, Mehran Kazemi, Dan Malkin, Ravin Kumar, David Vilar, Idan Brusilovsky, Jiaming Luo, Andreas Steiner, Abe Friesen, Abhanshu Sharma, Abheesht Sharma, Adi Mayrav Gilady, Adrian Goedeckemeyer, Alaa Saade, Alexander Kolesnikov, Alexei Bendebury, Alvin Abdagic, Amit Vadi, András György, André Susano Pinto, Anil Das, Ankur Bapna, Antoine Miech, Antoine Yang, Antonia Paterson, Ashish Shenoy, Ayan Chakrabarti, Bilal Piot, Bo Wu, Bobak Shahriari, Bryce Pettrini, Charlie Chen, Charline Le Lan, Christopher A. Choquette-Choo, CJ Carey, Cormac Brick, Daniel Deutsch, Danielle Eisenbud, Dee Cattle, Derek Cheng, Dimitris Paparas, Divyashree Shivakumar Sreepathihalli, Doug Reid, Dustin Tran, Dustin Zelle, Eric Noland, Erwin Huizenga, Eugene Kharitonov, Frederick Liu, Gagik Amirkhanyan, Glenn Cameron, Hadi Hashemi, Hanna Klimczak-Plucinska, Har-

- man Singh, Harsh Mehta, Harshal Tushar Lehri, Hussein Hazimeh, Ian Ballantyne, Idan Szpektor, and Ivan Nardini. Gemma 3 technical report. *CoRR*, abs/2503.19786, March 2025. URL <https://doi.org/10.48550/arXiv.2503.19786>.
- [99] Jiawen Kang, Zehui Xiong, Dusit Niyato, Han Yu, Ying-Chang Liang, and Dong In Kim. Incentive design for efficient federated learning in mobile networks: A contract theory approach. In *2019 IEEE VTS Asia Pacific Wireless Communications Symposium (APWCS)*, pages 1–5. IEEE, 2019.
- [100] Jiawen Kang, Zehui Xiong, Dusit Niyato, Yuze Zou, Yang Zhang, and Mohsen Guizani. Reliable federated learning for mobile networks. *IEEE Wireless Communications*, 27(2):72–80, 2020.
- [101] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. Dense passage retrieval for open-domain question answering. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6769–6781, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.550. URL <https://aclanthology.org/2020.emnlp-main.550>.
- [102] Jakob Nikolas Kather, Johannes Krisam, Pornpimol Charoentong, Tom Luedde, Esther Herpel, Cleo-Aron Weis, Timo Gaiser, Alexander Marx, Nektarios A Valous, Dyke Ferber, et al. Predicting survival from colorectal cancer histology slides using deep learning: A retrospective multicenter study. *PLoS medicine*, 16(1):e1002730, 2019.
- [103] Ahmad Faraz Khan, Azal Ahmad Khan, Ahmed M Abdelmoniem, Samuel Fountain, Ali R Butt, and Ali Anwar. FLOAT: Federated Learning Optimizations with Auto-

- mated Tuning. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 200–218, 2024.
- [104] Pang Wei Koh and Percy Liang. Understanding black-box predictions via influence functions. In *International conference on machine learning*, pages 1885–1894. PMLR, 2017.
- [105] Jakub Konečný, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. In *NeurIPS Workshop on Private Multi-Party Machine Learning*, 2016.
- [106] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research). URL <http://www.cs.toronto.edu/~kriz/cifar.html>.
- [107] Weirui Kuang, Bingchen Qian, Zitao Li, Daoyuan Chen, Dawei Gao, Xuchen Pan, Yuexiang Xie, Yaliang Li, Bolin Ding, and Jingren Zhou. Federatedscope-llm: A comprehensive package for fine-tuning large language models in federated learning. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5260–5271, 2024.
- [108] Fan Lai, Xiangfeng Zhu, Harsha V Madhyastha, and Mosharaf Chowdhury. Oort: Efficient federated learning via guided participant selection. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 19–35, 2021.
- [109] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. ALBERT: A lite BERT for self-supervised learning of language representations. *CoRR*, abs/1909.11942, 2019. URL <http://arxiv.org/abs/1909.11942>.

- [110] Tra Huong Thi Le, Nguyen H Tran, Yan Kyaw Tun, Minh NH Nguyen, Shashi Raj Pandey, Zhu Han, and Choong Seon Hong. An incentive mechanism for federated learning in wireless cellular networks: An auction approach. *IEEE Transactions on Wireless Communications*, 20(8):4874–4887, 2021.
- [111] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54–72, 2011.
- [112] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.
- [113] Ronny Lempel and Shlomo Moran. Predictive caching and prefetching of query results in search engines. In *Proceedings of the 12th international conference on World Wide Web*, pages 19–28, 2003.
- [114] Junnan Li, Yongkang Wong, Qi Zhao, and Mohan S Kankanhalli. Learning to learn from noisy labeled data. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5051–5059, 2019.
- [115] Junnan Li, Richard Socher, and Steven C.H. Hoi. Dividemix: Learning with noisy labels as semi-supervised learning. In *International Conference on Learning Representations*, 2020. URL <https://openreview.net/forum?id=HJgExaVtwr>.
- [116] Junnan Li, Caiming Xiong, and Steven CH Hoi. Learning from noisy data with robust representation learning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 9485–9494, 2021.
- [117] Qinbin Li, Yiqun Diao, Quan Chen, and Bingsheng He. Federated Learning on Non-IID Data Silos: An Experimental Study. In *IEEE International Conference on Data Engineering*, 2022.

- [118] Tian Li, Maziar Sanjabi, Ahmad Beirami, and Virginia Smith. Fair Resource Allocation in Federated Learning. In *International Conference on Learning Representations*, 2020.
- [119] Yige Li, Hanxun Huang, Yunhan Zhao, Xingjun Ma, and Jun Sun. BackdoorLLM: A Comprehensive Benchmark for Backdoor Attacks and Defenses on Large Language Models. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2025.
- [120] Tao Lin, Lingjing Kong, Sebastian U Stich, and Martin Jaggi. Ensemble distillation for robust model fusion in federated learning. *Advances in Neural Information Processing Systems*, 33:2351–2363, 2020.
- [121] Junxu Liu, Jian Lou, Li Xiong, Jinfei Liu, and Xiaofeng Meng. Projected federated averaging with heterogeneous differential privacy. *Proceedings of the VLDB Endowment*, 15(4):828–840, 2021.
- [122] Yang Liu, Yan Kang, Liping Li, Xinwei Zhang, Yong Cheng, Tianjian Chen, Mingyi Hong, and Qiang Yang. A communication efficient vertical federated learning framework. *Scanning Electron Microsc Meet at*, 2019.
- [123] Yeja Liu, Weiyuan Wu, Lampros Flokas, Jiannan Wang, and Eugene Wu. Enabling sql-based training data debugging for federated learning. *arXiv preprint arXiv:2108.11884*, 2021.
- [124] Dionysios Logothetis, Soumyarupa De, and Kenneth Yocum. Scalable Lineage Capture for Debugging DISC Analytics. In *Proceedings of the 4th Annual Symposium on Cloud Computing, SOCC '13*, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450324281. doi: 10.1145/2523616.2523619.

- [125] Guodong Long, Yue Tan, Jing Jiang, and Chengqi Zhang. Federated learning for open banking. In *Federated learning*, pages 240–254. Springer, 2020.
- [126] Xiaohui Long and Torsten Suel. Three-level caching for efficient query processing in large web search engines. In *Proceedings of the 14th international conference on World Wide Web*, pages 257–266, 2005.
- [127] Heiko Ludwig, Nathalie Baracaldo, Gegi Thomas, Yi Zhou, Ali Anwar, Shashank Rajamoni, Yuya Ong, Jayaram Radhakrishnan, Ashish Verma, Mathieu Sinn, et al. IBM Federated Learning: an Enterprise Framework White Paper V0. 1. *arXiv preprint arXiv:2007.10987*, 2020.
- [128] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. *Advances in neural information processing systems*, 30, 2017.
- [129] Lingjuan Lyu, Xinyi Xu, Qian Wang, and Han Yu. Collaborative fairness in federated learning. In *Federated Learning*, pages 189–204. Springer, 2020.
- [130] Lingjuan Lyu, Jiangshan Yu, Karthik Nandakumar, Yitong Li, Xingjun Ma, Jiong Jin, Han Yu, and Kee Siong Ng. Towards fair and privacy-preserving federated deep models. *IEEE Transactions on Parallel and Distributed Systems*, 31(11):2524–2541, 2020.
- [131] Shuming Ma, Hongyu Wang, Lingxiao Ma, Lei Wang, Wenhui Wang, Shaohan Huang, Li Dong, Ruiping Wang, Jilong Xue, and Furu Wei. The era of 1-bit llms: All large language models are in 1.58 bits. *arXiv preprint arXiv:2402.17764*, 2024.
- [132] Xingjun Ma, Yisen Wang, Michael E Houle, Shuo Zhou, Sarah Erfani, Shutao Xia, Sudanthi Wijewickrema, and James Bailey. Dimensionality-driven learning with noisy

- labels. In *International Conference on Machine Learning*, pages 3355–3364. PMLR, 2018.
- [133] Evangelos P. Markatos. On caching search engine query results. *Computer Communications*, 24(2):137–143, 2001.
- [134] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*, pages 1273–1282. PMLR, 2017.
- [135] H Brendan McMahan, Daniel Ramage, Kunal Talwar, and Li Zhang. Learning Differentially Private Recurrent Language Models. In *International Conference on Learning Representations*, 2018.
- [136] Casey Meehan, Khalil Mrini, and Kamalika Chaudhuri. Sentence-level Privacy for Document Embeddings. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3367–3380, 2022.
- [137] Meta AI. Llama 3.2: Revolutionizing edge ai and vision with open, customizable models. Meta AI Blog, 2024. URL <https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices/>.
- [138] Vaikkunth Mugunthan, Ravi Rahman, and Lalana Kagal. Blockflow: An accountable and privacy-preserving solution for federated learning. *arXiv preprint arXiv:2007.03856*, 2020.
- [139] Vinod Nair and Geoffrey E. Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning, ICML’10*, page 807–814, Madison, WI, USA, 2010. Omnipress. ISBN 9781605589077.

- [140] Joseph P. Near and Chiké Abuah. *Programming Differential Privacy*, volume 1. 2021. URL <https://programming-dp.com/>.
- [141] Thien Duc Nguyen, Phillip Rieger, Huili Chen, Hossein Yalame, Helen Möllering, Hossein Fereidooni, Samuel Marchal, Markus Miettinen, Azalia Mirhoseini, Shaza Zeitouni, Farinaz Koushanfar, Ahmad-Reza Sadeghi, and Thomas Schneider. FLAME: Taming backdoors in federated learning. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1415–1432, Boston, MA, August 2022. USENIX Association. ISBN 978-1-939133-31-1. URL <https://www.usenix.org/conference/usenixsecurity22/presentation/nguyen>.
- [142] Jianmo Ni, Gustavo Hernandez Abrego, Noah Constant, Ji Ma, Keith Hall, Daniel Cer, and Yinfei Yang. Sentence-t5: Scalable sentence encoders from pre-trained text-to-text models. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio, editors, *Findings of the Association for Computational Linguistics: ACL 2022*, pages 1864–1874, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.findings-acl.146. URL <https://aclanthology.org/2022.findings-acl.146>.
- [143] Augustus Odena, Catherine Olsson, David Andersen, and Ian Goodfellow. Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. In *International Conference on Machine Learning*, pages 4901–4911. PMLR, 2019.
- [144] Chris Olah, Arvind Satyanarayan, Ian Johnson, Shan Carter, Ludwig Schubert, Katherine Ye, and Alexander Mordvintsev. The building blocks of interpretability. *Distill*, 3(3):e10, 2018.
- [145] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training

- language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.
- [146] Mustafa Safa Ozdayi, Murat Kantarcioglu, and Yulia R Gel. Defending against backdoors in federated learning with robust learning rate. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 9268–9276, 2021.
- [147] Michela Paganini and Jessica Zosa Forde. dagger: A python framework for reproducible machine learning experiment orchestration. *CoRR*, abs/2006.07484, 2020.
- [148] Hyunjung Park, Robert Ikeda, and Jennifer Widom. RAMP: A System for Capturing and Tracing Provenance in MapReduce Workflows. *Proc. VLDB Endow.*, 4(12):1351–1354, jun 2020. ISSN 2150-8097. doi: 10.14778/3402755.3402768.
- [149] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.
- [150] Karl Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11): 559–572, 1901. doi: 10.1080/14786440109462720. URL <https://doi.org/10.1080/14786440109462720>.
- [151] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18, 2017.

- [152] Guilherme Penedo, Quentin Malartic, Daniel Hesslow, Ruxandra Cojocaru, Hamza Alobeidli, Alessandro Cappelli, Baptiste Pannier, Ebtesam Almazrouei, and Julien Launay. The refinedweb dataset for falcon llm: Outperforming curated corpora with web data only. *Advances in Neural Information Processing Systems*, 36, 2024.
- [153] Matthew E. Peters, Mark Neumann, Luke Zettlemoyer, and Wen-tau Yih. Dissecting contextual word embeddings: Architecture and representation. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun’ichi Tsujii, editors, *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1499–1509, Brussels, Belgium, October–November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1179. URL <https://aclanthology.org/D18-1179/>.
- [154] Stefan Podlipnig and Laszlo Böszörményi. A survey of web cache replacement strategies. *ACM Computing Surveys (CSUR)*, 35(4):374–398, 2003.
- [155] Qwen Team. Qwen2.5-llm: Extending the boundary of llms. Alibaba Cloud Community Blog, 2024. URL https://www.alibabacloud.com/blog/qwen2-5-llm-extending-the-boundary-of-llms_601786.
- [156] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.
- [157] Hannah Rashkin, Vitaly Nikolaev, Matthew Lamm, Lora Aroyo, Michael Collins, Dipanjan Das, Slav Petrov, Gaurav Singh Tomar, Iulia Turc, and David Reitter. Measuring attribution in natural language generation models. *Computational Linguistics*, 49(4):777–840, December 2023. doi: 10.1162/coli_a_00486. URL <https://aclanthology.org/2023.cl-4.2/>.
- [158] Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun

- Wan, editors, *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3982–3992, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1410. URL <https://aclanthology.org/D19-1410>.
- [159] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. ” why should i trust you?” explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144, 2016.
- [160] Nicola Rieke, Jonny Hancox, Wenqi Li, Fausto Milletari, Holger R Roth, Shadi Albarqouni, Spyridon Bakas, Mathieu N Galtier, Bennett A Landman, Klaus Maier-Hein, et al. The future of digital health with federated learning. *NPJ digital medicine*, 3(1): 1–7, 2020.
- [161] Lukas Rupprecht, James C Davis, Constantine Arnold, Yaniv Gur, and Deepavali Bhagwat. Improving reproducibility of data science pipelines through transparent provenance capture. *Proceedings of the VLDB Endowment*, 13(12):3354–3368, 2020.
- [162] Noveen Sachdeva, Benjamin Coleman, Wang-Cheng Kang, Jianmo Ni, Lichan Hong, Ed H Chi, James Caverlee, Julian McAuley, and Derek Zhiyuan Cheng. How to train data-efficient llms. *arXiv preprint arXiv:2402.09668*, 2024.
- [163] Sheeba Samuel, Frank Löffler, and Birgitta König-Ries. Machine learning pipelines: Provenance, reproducibility and fair data principles. In *Provenance and Annotation of Data and Processes: 8th and 9th International Provenance and Annotation Workshop, IPAW 2020+ IPAW 2021, Virtual Event, July 19–22, 2021, Proceedings 8*, pages 226–230. Springer, 2021.

- [164] Lorenzo Sani, Alex Jacob, Zeyu Cao, Bill Marino, Yan Gao, Tomas Paulik, Wanru Zhao, William F Shen, Preslav Aleksandrov, Xinchu Qiu, et al. The Future of Large Language Model Pre-training is Federated. *arXiv preprint arXiv:2405.10853*, 2024.
- [165] Patricia Correia Saraiva, Edleno Silva de Moura, Nivio Ziviani, Wagner Meira, Rodrigo Fonseca, and Berthier Ribeiro-Neto. Rank-preserving two-level caching for scalable search engines. In *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 51–58, 2001.
- [166] Yunus Sarikaya and Ozgur Ercetin. Motivating workers in federated learning: A stackelberg game perspective. *IEEE Networking Letters*, 2(1):23–27, 2019.
- [167] David Saxton, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. Analysing Mathematical Reasoning Abilities of Neural Models. In *International Conference on Learning Representations*, 2019. URL <https://openreview.net/forum?id=H1gR5iR5FX>.
- [168] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision*, pages 618–626, 2017.
- [169] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. Learning important features through propagating activation differences. In *International conference on machine learning*, pages 3145–3153. PMLR, 2017.
- [170] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL <http://arxiv.org/abs/1409.1556>.

- [171] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps. In Yoshua Bengio and Yann LeCun, editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Workshop Track Proceedings*, 2014.
- [172] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. Mpnet: Masked and permuted pre-training for language understanding. *Advances in Neural Information Processing Systems*, 33:16857–16867, 2020.
- [173] Renan Souza, Leonardo Azevedo, Vítor Lourenço, Elton Soares, Raphael Thiago, Rafael Brandao, Daniel Civitarese, Emilio Brazil, Marcio Moreno, Patrick Valduriez, et al. Provenance data in the machine learning lifecycle in computational science and engineering. In *2019 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*, pages 1–10. IEEE, 2019.
- [174] Jack W Stokes, Paul England, and Kevin Kane. Preventing Machine Learning Poisoning Attacks using Authentication and Provenance. In *MILCOM 2021-2021 IEEE Military Communications Conference (MILCOM)*, pages 181–188. IEEE, 2021.
- [175] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in nlp. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3645–3650, 2019.
- [176] Bing Sun, Jun Sun, Long H Pham, and Jie Shi. Causality-based neural network repair. In *Proceedings of the 44th International Conference on Software Engineering*, pages 338–349, 2022.
- [177] Jingwei Sun, Ziyue Xu, Hongxu Yin, Dong Yang, Daguang Xu, Yudong Liu, Zhixu Du, Yiran Chen, and Holger R. Roth. Fedbpt: efficient federated black-box prompt

- tuning for large language models. In *Proceedings of the 41st International Conference on Machine Learning, ICML'24*. JMLR.org, 2024.
- [178] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70, ICML'17*, page 3319–3328. JMLR.org, 2017.
- [179] Canh T Dinh, Nguyen Tran, and Josh Nguyen. Personalized federated learning with moreau envelopes. *Advances in Neural Information Processing Systems*, 33:21394–21405, 2020.
- [180] Ammar Tahir, Yongzhou Chen, and Prashanti Nilayam. FedSS: Federated learning with smart selection of clients. In *Federated Learning Systems (FLSys) Workshop @ MLSys 2023*, 2023. URL <https://openreview.net/forum?id=kSIJ3ScQ-e>.
- [181] Shunpu Tang, Wenqi Zhou, Lunyuan Chen, Lijia Lai, Junjuan Xia, and Liseng Fan. Battery-constrained federated edge learning in uav-enabled iot for b5g/6g networks. *Physical Communication*, 47:101381, 2021.
- [182] Chuanqi Tao, Yali Tao, Hongjing Guo, Zhiqiu Huang, and Xiaobing Sun. Dregion: coverage-guided fuzz testing of deep neural networks with region-based neuron selection strategies. *Information and Software Technology*, 162:107266, 2023.
- [183] Ian Tenney, Dipanjan Das, and Ellie Pavlick. BERT rediscovers the classical NLP pipeline. In Anna Korhonen, David Traum, and Lluís Màrquez, editors, *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4593–4601, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1452. URL <https://aclanthology.org/P19-1452/>.

- [184] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.
- [185] Albert Tseng, Jerry Chee, Qingyao Sun, Volodymyr Kuleshov, and Christopher De Sa. Quip#: Even better llm quantization with hadamard incoherence and lattice codebooks. *arXiv preprint arXiv:2402.04396*, 2024.
- [186] Muhammad Habib ur Rehman, Ahmed Mukhtar Dirir, Khaled Salah, Ernesto Damiani, and Davor Svetinovic. Trustfed: a framework for fair and trustworthy cross-device federated learning in iiot. *IEEE Transactions on Industrial Informatics*, 17(12):8485–8494, 2021.
- [187] Muhammad Usman, Divya Gopinath, Youcheng Sun, Yannic Noller, and Corina S Păsăreanu. Nnrepair: Constraint-based repair of neural network classifiers. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I 33*, pages 3–25. Springer, 2021.
- [188] Muhammad Usman, Yannic Noller, Corina S Păsăreanu, Youcheng Sun, and Divya Gopinath. Neurospf: A tool for the symbolic analysis of neural networks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 25–28. IEEE, 2021.
- [189] Salil Vadhan and Tianhao Wang. Concurrent Composition of Differential Privacy. In *Theory of Cryptography Conference*, pages 582–604, 2021.
- [190] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

- [191] Hongyi Wang, Mikhail Yurochkin, Yuekai Sun, Dimitris Papailiopoulos, and Yasaman Khazaeni. Federated Learning with Matched Averaging. In *International Conference on Learning Representations*, 2020.
- [192] Hongyu Wang, Shuming Ma, Li Dong, Shaohan Huang, Huaijie Wang, Lingxiao Ma, Fan Yang, Ruiping Wang, Yi Wu, and Furu Wei. Bitnet: Scaling 1-bit transformers for large language models. *arXiv preprint arXiv:2310.11453*, 2023.
- [193] Jianyu Wang, Qinghua Liu, Hao Liang, Gauri Joshi, and H Vincent Poor. Tackling the objective inconsistency problem in heterogeneous federated optimization. *Advances in neural information processing systems*, 33:7611–7623, 2020.
- [194] Tianhao Wang, Jeremiah Blocki, Ninghui Li, and Somesh Jha. Locally Differentially Private Protocols for Frequency Estimation. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 729–745, Vancouver, BC, August 2017. USENIX Association. ISBN 978-1-931971-40-9. URL <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang-tianhao>.
- [195] Mohammad Wardat, Wei Le, and Hridesh Rajan. Deeplocalize: fault localization for deep neural networks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 251–262. IEEE, 2021.
- [196] Stanley L Warner. Randomized Response: A Survey Technique for Eliminating Evasive Answer Bias. *Journal of the American statistical association*, 60(309):63–69, 1965.
- [197] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M.

- Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45. Association for Computational Linguistics, 2020. URL <https://www.aclweb.org/anthology/2020.emnlp-demos.6>.
- [198] Feijie Wu, Zitao Li, Yaliang Li, Bolin Ding, and Jing Gao. Fedbiot: Llm local fine-tuning in federated learning without full model. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD '24*, page 3345–3355, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704901. doi: 10.1145/3637528.3671897. URL <https://doi.org/10.1145/3637528.3671897>.
- [199] Marvin Xhemrishi, Johan Östman, Antonia Wachter-Zeh, et al. Fedgt: Identification of malicious clients in federated learning with secure aggregation. *arXiv preprint arXiv:2305.05506*, 2023.
- [200] Chulin Xie, Zinan Lin, Arturs Backurs, Sivakanth Gopi, Da Yu, Huseyin A Inan, Harsha Nori, Haotian Jiang, Huishuai Zhang, Yin Tat Lee, et al. Differentially Private Synthetic Data via Foundation Model APIs 2: Text. In *International Conference on Machine Learning*, pages 54531–54560. PMLR, 2024.
- [201] Xiaofei Xie, Lei Ma, Haijun Wang, Yuekang Li, Yang Liu, and Xiaohong Li. Diffchaser: Detecting disagreements for deep neural networks. In *IJCAI*, pages 5772–5778, 2019.
- [202] Xiaofei Xie, Tianlin Li, Jian Wang, Lei Ma, Qing Guo, Felix Juefei-Xu, and Yang Liu. NPC: Neuron Path Coverage via Characterizing Decision Logic of Deep Neural Networks. *ACM Trans. Softw. Eng. Methodol.*, 31(3), April 2022. ISSN 1049-331X. doi: 10.1145/3490489.

- [203] Yinglian Xie and David O’Hallaron. Locality in search engine queries and its implications for caching. In *Proceedings. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 3, pages 1238–1247. IEEE, 2002.
- [204] Doris Xin, Hui Miao, Aditya Parameswaran, and Neoklis Polyzotis. Production machine learning pipelines: Empirical analysis and optimization opportunities. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2639–2652, 2021.
- [205] Xinyi Xu, Lingjuan Lyu, Xingjun Ma, Chenglin Miao, Chuan Sheng Foo, and Bryan Kian Hsiang Low. Gradient driven rewards to guarantee fairness in collaborative machine learning. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 16104–16117. Curran Associates, Inc., 2021. URL https://proceedings.neurips.cc/paper_files/paper/2021/file/8682cc30db9c025ecd3fee433f8ab54c-Paper.pdf.
- [206] Xuanang Xu, Fugen Zhou, Bo Liu, Dongshan Fu, and Xiangzhi Bai. Efficient multiple organ localization in ct image using 3d region proposal network. *IEEE transactions on medical imaging*, 38(8):1885–1898, 2019.
- [207] Zichen Xu, Li Li, and Wenting Zou. Exploring federated learning on battery-powered devices. In *Proceedings of the ACM Turing Celebration Conference-China*, pages 1–6, 2019.
- [208] Hongyang Yang, Xiao-Yang Liu, and Christina Dan Wang. FinGPT: Open-Source Financial Large Language Models. *FinLLM at IJCAI*, 2023.
- [209] Jiancheng Yang, Rui Shi, and Bingbing Ni. MedMNIST Classification Decathlon: A

- Lightweight AutoML Benchmark for Medical Image Analysis. In *IEEE 18th International Symposium on Biomedical Imaging (ISBI)*, pages 191–195, 2021.
- [210] Jiancheng Yang, Rui Shi, Donglai Wei, Zequan Liu, Lin Zhao, Bilian Ke, Hanspeter Pfister, and Bingbing Ni. MedMNIST v2-A large-scale lightweight benchmark for 2D and 3D biomedical image classification. *Scientific Data*, 10(1):41, 2023.
- [211] Jiangchao Yao, Jiajie Wang, Ivor W Tsang, Ya Zhang, Jun Sun, Chengqi Zhang, and Rui Zhang. Deep learning from noisy image labels with quality embedding. *IEEE Transactions on Image Processing*, 28(4):1909–1922, 2018.
- [212] Dongdong Ye, Rong Yu, Miao Pan, and Zhu Han. Federated learning in vehicular edge computing: A selective model aggregation approach. *IEEE Access*, 8:23920–23935, 2020.
- [213] Sixing Yu, Phuong Nguyen, Waqwoya Abebe, Wei Qian, Ali Anwar, and Ali Jannesari. Spatl: Salient parameter aggregation and transfer learning for heterogeneous federated learning. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14. IEEE, 2022.
- [214] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6–12, 2014, Proceedings, Part I 13*, pages 818–833. Springer, 2014.
- [215] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *Software Engineering, IEEE Transactions on*, 28(2):183–200, 2002.
- [216] Rongfei Zeng, Shixun Zhang, Jiaqi Wang, and Xiaowen Chu. Fmore: An incentive scheme of multi-dimensional auction for federated learning in mec. In *2020 IEEE 40th*

- International Conference on Distributed Computing Systems (ICDCS)*, pages 278–288. IEEE, 2020.
- [217] Rongfei Zeng, Chao Zeng, Xingwei Wang, Bo Li, and Xiaowen Chu. A comprehensive survey of incentive mechanism for federated learning. *arXiv preprint arXiv:2106.15406*, 2021.
- [218] Yufeng Zhan, Jie Zhang, Zicong Hong, Leijie Wu, Peng Li, and Song Guo. A survey of incentive mechanism design for federated learning. *IEEE Transactions on Emerging Topics in Computing*, 10(2):1035–1044, 2021.
- [219] Chaoning Zhang, Philipp Benz, Dawit Mureja Argaw, Seokju Lee, Junsik Kim, Francois Rameau, Jean-Charles Bazin, and In So Kweon. Resnet or densenet? introducing dense shortcuts to resnet. In *Proceedings of the IEEE/CVF winter conference on applications of computer vision*, pages 3550–3559, 2021.
- [220] Jiangong Zhang, Xiaohui Long, and Torsten Suel. Performance of compressed inverted list caching in search engines. In *Proceedings of the 17th international conference on World Wide Web*, pages 387–396, 2008.
- [221] Weishan Zhang, Qinghua Lu, Qiuyu Yu, Zhaotong Li, Yue Liu, Sin Kit Lo, Shiping Chen, Xiwei Xu, and Liming Zhu. Blockchain-based federated learning for device failure detection in industrial iot. *IEEE Internet of Things Journal*, 8(7):5926–5937, 2020.
- [222] Xiang Zhang, Junbo Zhao, and Yann LeCun. Character-level convolutional networks for text classification. *Advances in neural information processing systems*, 28, 2015.
- [223] Zhaohua Zheng, Yize Zhou, Yilong Sun, Zhang Wang, Boyi Liu, and Keqiu Li. Ap-

- plications of federated learning in smart cities: recent advances, taxonomy, and open challenges. *Connection Science*, pages 1–28, 2021.
- [224] Banghua Zhu, Ying Sheng, Lianmin Zheng, Clark Barrett, Michael Jordan, and Jiantao Jiao. Towards optimal caching and model selection for large model inference. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL <https://openreview.net/forum?id=gd20oaZqqF>.