CS240 Project Report — Fall 2025

Author(s): Abdulrahman Alshahrani
Email(s): abdulrahman.alshahrani@kaust.edu.sa
Type: Regular Project
Target Traits: Reliability

December 7, 2025

1. Context and Motivation

Federated Learning (FL) enables the training of a centralized machine learning model across distributed clients without collecting their personal data. This approach help companies train big, performative models without infringing on the consumers' privacy, which is a growing problem in today's world. This field is also valuable for applications involving sensitive data like healthcare records and private banking data. However, these FL systems face two critical challenges: Byzantine failures and operational costs. While operational costs are outside the scope of this project, I decided to investigate Byzantine failures, where malicious or faulty clients submit corrupted model updates that can degrade the global model's performance.

There are many research papers and experiments around Byzantine-robust aggregation strategies for federated learning. My main research questions are:

- How do these different aggregation methods perform under various ratios of malicious clients and system configurations?
- How does my proposed Variational Auto Encoder[4] (VAE) anomaly detection approach compares against other aggregation methods?
- How does the real distributed deployments differ from the simulated results?

2. Design and Architecture

The system implements a federated learning framework built on the Flower[1] framework, with both simulation and real deployment capabilities. The architecture consists of three main layers: client implementation, aggregation strategies, and orchestration.

In figure 1, you can see an overview of the code architecture of the project. We have 3 primary sections: Experiment Orchestrator, FL Server, and Logging & Plotting. The Experiment Orchestrator accepts a given experiment environment (e.g. what datasets to use, what models to use, what server aggregator to run, for how many rounds, etc.), then it tells the FL server to starts with that environment. The Logging & Analysis section logs all the relevant results of the experiment and produces various plots.

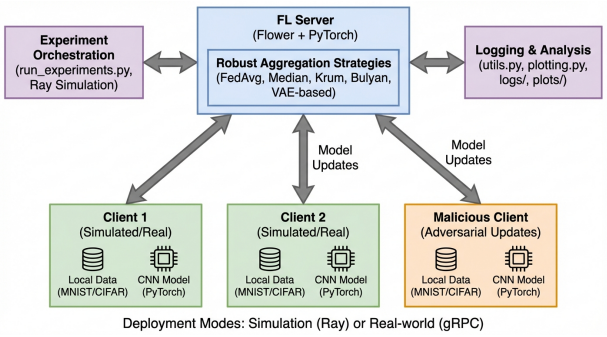


Figure 1: A diagram describing the architecture of the code project.

2.1. Client Implementation

In Flower, you can think of each client as an idempotent function `fit(parameters, config)` that takes in model parameters, trains them on its local dataset, and returns the gradients back. Each `config` object notes to the client which data partition it should use and whether it is malicious or not.

The client implementation is agnostic to whether it runs in simulation or as a distributed process. For real deployments, clients use gRPC to communicate with the server.

2.2. Aggregation Strategies

The server implements six aggregation strategies: FedAvg[6] (Baseline), FedMedian, Krum[7] (Multi-Krum), Bulyan[2], and my VAE-based approach.

Table 1: System Configuration Summary

Component	Specification
Language	Python 3.12+
FL Framework	Flower 1.22.0+ (federated learning primitives)
ML Framework	PyTorch[8] 2.9.1+ (model training)
Communication	gRPC (real deployment), Ray (simulation)
Data Processing	NumPy, torchvision (CIFAR-10[5] loading)
Visualization	Matplotlib, Seaborn, Pandas
Environment (Real Deployment)	Google Cloud Platform[3] e2-medium VMs (1–2 vCPUs, 4GB RAM)
Environment (Simulation)	x86 AMD 16-core CPU, 48GB RAM, Nvidia 5080 GPU

2.2.1 VAE-based Aggregator

Learns the distribution of honest updates using a Variational Autoencoder. Computes reconstruction error for each client update and assigns confidence scores. Updates with high reconstruction error receive lower weights in aggregation. Uses PCA to reduce parameter dimensionality before VAE processing.

2.3. Concurrency and Failure Handling

In simulation mode, Flower's Ray-based backend handles concurrent client execution. Each client runs in a separate Ray actor, allowing parallel training. The system enforces synchronous rounds—the server waits for all selected clients before aggregating.

For real deployment, the server listens for gRPC connections and manages client registration. The implementation includes retry logic for failed connections but does not handle client dropouts mid-round. I assumed synchronous, reliable communication, however, this can be easily adjusted by making the server train on a subset of the clients, moving to other clients if some of them timed out or failed.

3. Implementation Details

3.1. Design Decision

Model Architecture: The CIFAR-10 model uses a standard CNN with three convolutional layers (32, 64, 128 filters), max pooling, and three fully connected layers (256, 128, 10 units). Dropout (0.5) is applied after the first two FC layers. This architecture is simple enough to train quickly while being sufficiently complex to observe degradation under attack.

Byzantine Client Behavior: Gradient sign flipping was chosen as the attack model because it is simple to implement, computationally cheap, and effective at degrading model performance. It basically flips the signs of each component in the gradient update of the client. More sophisticated attacks (e.g., label flipping, model poisoning, random gradients) exist, but they are beyond the scope of this project since they need further intricate testing to measure their effects. I argue that if a gradient is the best vector to minimize loss, then the opposite vector is the 'worst-case' vector that will likely sway the global model away from convergence the most. This is way I chose the sign flipping as the attack model.

4. Evaluation

4.1. Simulation Results

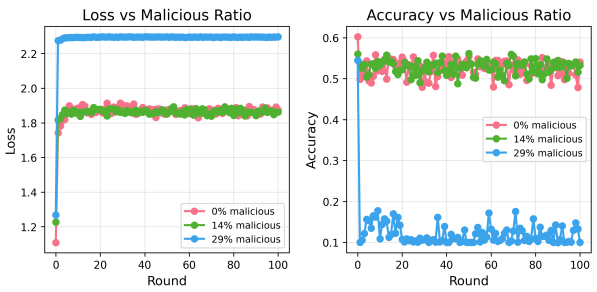
In the simulated environment, several aggregation methods were tested under different ratios of malicious clients and various setups.

In Fig. 3a, the performance of **FedAvg** collapses rapidly even with a small fraction of malicious clients. At only 10% malicious participants, accuracy drops to around 10%, and convergence effectively fails. This shows how easily the aggregation process can be corrupted.

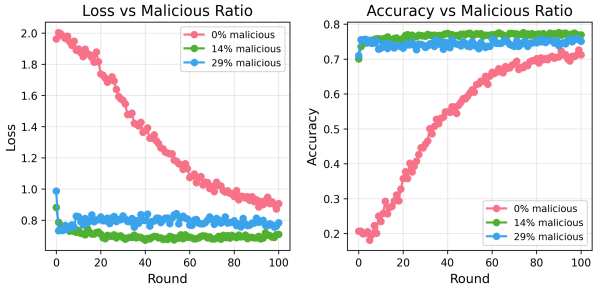
Fig. 3b provides the ideal baseline, where all methods converge smoothly without malicious clients, confirming that each performs as expected under clean conditions.

With one malicious client (Fig. 3c), method differences become clear. **FedAvg** performs worst, while **FedMedian** shows the most robust convergence. Both **Krum** and **Bulyan** converge quickly before stabilizing. The proposed **VAE-based** aggregator starts weakly, resembling FedAvg, but improves gradually and surpasses it. This indicates that it learns to down-weight Byzantine clients over time, though it converges slower and demands more computation.

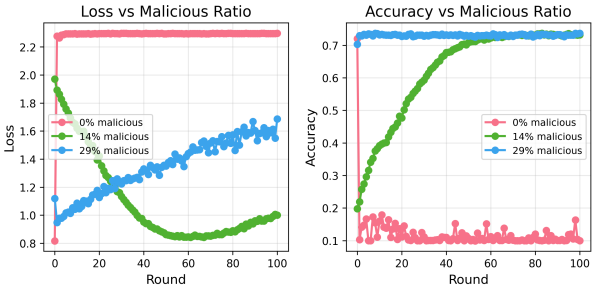
Fig. 3d shows how varying the number of clients affects FedAvg without malicious behavior. Fewer clients yield faster convergence and higher accuracy, aligning with the idea that larger per-client batch sizes lead to smoother and quicker training.



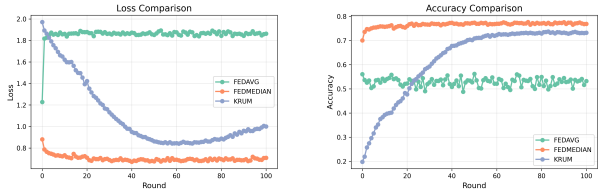
(a) Real setup comparing loss and accuracy across various percentages of malicious clients using FedAvg (7 clients total).



(b) Real setup comparing loss and accuracy across various percentages of malicious clients using FedMedian (7 clients total).

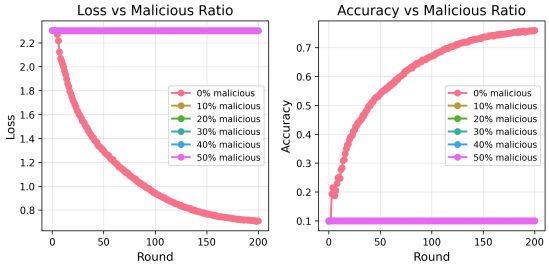


(c) Real setup comparing loss and accuracy across various percentages of malicious clients using Krum (7 clients total).

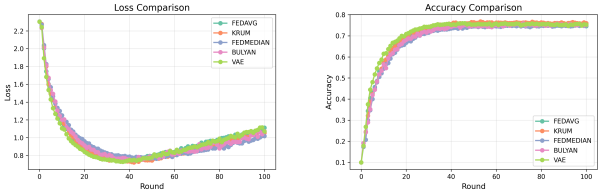


(d) Real strategy comparison with 7 clients and 1 malicious client. Shows different strategies' effects on loss and accuracy over rounds.

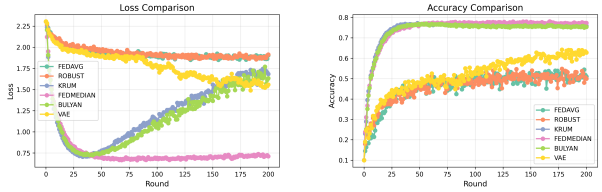
Figure 2: Comparison of real experimental setups showing performance under different aggregation strategies and malicious client ratios.



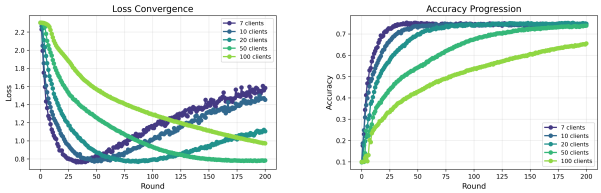
(a) Simulation setup comparing loss and accuracy across various percentages of malicious clients using FedAvg (10 clients total).



(b) Simulation setup comparing strategies with 7 clients and 0 malicious clients. Shows different strategies' effects on loss and accuracy over rounds.



(c) Simulation setup comparing strategies with 7 clients and 1 malicious client. Shows different strategies' effects on loss and accuracy over rounds.



(d) Simulation setup comparing different number of clients (0% malicious) on FedAvg. Shows the effect of different number of clients on loss and accuracy over rounds.

Figure 3: Comparison of simulation setups analyzing the impact of malicious clients and strategies on learning performance.

4.2. Real-World Experiments

In the real distributed experiments, overall patterns from simulations remained but appeared noisier with occasional outliers.

As seen in Fig. 2b, **FedMedian** consistently converges best, regardless of malicious client ratios. **Krum** (Fig. 2c) shows a comparable trend, though the 0% malicious run had unexpectedly low accuracy, possibly due to GCP resource throttling.

The **FedAvg** results (Fig. 2a) confirm its sensitivity: accuracy degrades steadily with more malicious clients. Notably, the 0/7 and 1/7 malicious setups performed similarly, while 2/7 reduced accuracy to about 10%.

A combined view (Fig. 2d) with one malicious client reinforces these findings: FedMedian converges fastest and achieves highest accuracy, followed by Krum. FedAvg remains the weakest with low accuracy.

Though consistent with simulations, real tests show greater noise and runtime variation. On average, distributed runs took about **4.7 times** longer per round vs the simulation runs (42s vs 9s), likely due to virtualization and network overhead within the same GCP zone. Using different zones or regions for the VMs would've likely resulted in even bigger differences.

4.3. Summary of Findings

Performance Across Aggregators. In both simulations and real-world experiments, **FedMedian** consistently achieved the best results under malicious conditions. **Krum** and **Bulyan** showed strong convergence and robustness to noise and Byzantine behavior, performing similarly as expected. The superior performance of FedMedian is natural since the coordinate-wise median is inherently robust to outliers, which correspond to the corrupted client updates in our results.

Effectiveness of the VAE-Based Method. The proposed **VAE-based** anomaly detection method showed promising but limited results. It began like FedAvg but gradually reduced loss and surpassed it by down-weighting byzantine contributions, indicating effective learning of latent anomaly features. However, it has slower convergence and higher computational cost, due to its combination of deep learning and PCA, so it remains a disadvantage compared to other methods.

Differences Between Simulation and Real Deployment. Real-world results supported simulation findings but exhibited more noise and hardware-related variability. Resource limitations prevented full testing of some methods such as **Bulyan** and the **VAE-based** approach. Still, both environments consistently showed that **FedMedian** is the most robust and efficient under malicious conditions, while **FedAvg** remains highly sensitive to even a few adversarial clients.

5. Discussion and Reflection

5.1. What Worked Well

The modular design made running the experiments surprisingly smooth. I could test new strategies without touching the client or orchestration code. The re-use of previous runs feature saved hours by skipping redundant re-runs, and the automated plotting tools handled most of the visualization work with little manual effort.

The Byzantine simulation using gradient sign flipping worked well as a consistent, reproducible attack, and CIFAR-10 served as a solid benchmark: simple enough to manage but complex enough to reveal meaningful differences.

Even with the GCP issues, transitioning from simulation to real deployment was straightforward. The system ran with minimal code changes, and it was rewarding to see similar convergence patterns appear in real distributed conditions.

5.2. What Didn't Work

The main limitation was the GCP environment. Hidden quotas frequently caused "exceeded quota" errors, restricting the real experiments to 7 clients. After extended use, GCP throttled network and CPU resources to about 5% of normal performance, making further tests with **VAE** or **Bulyan** impractical. With more time, switching to Kubernetes or other cloud providers could resolve these issues.

Some runs of **Krum** and **Bulyan** showed unexpected loss divergence. This likely occurred because the evaluation used the full dataset, while the original papers used the selected clients' partitions. The methods may have converged locally even though the global loss trends suggested otherwise.

References

[1] Daniel J Beutel, Taner Topal, Akhil Mathur, Xinchi Qiu, Javier Fernandez-Marques, Yan Gao, Lorenzo Sani, Hei Li Kwing, Titouan Parcollet, Pedro PB de Gusmão, and Nicholas D Lane. Flower: A friendly federated learning research framework. *arXiv preprint arXiv:2007.14390*, 2020.

[2] Peva Blanchard, El Mahdi El Mhamdi, Rachid Guerraoui, and Julien Stainer. Byzantine-tolerant machine learning. *CoRR*, abs/1703.02757, 2017.

[3] Google. Google Cloud Platform. urlhttps://cloud.google.com/, 2025. Accessed: 2025.

[4] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2022.

[5] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

[6] H. Brendan McMahan, Eider Moore, Daniel Ramage, and Blaise Agüera y Arcas. Federated learning of deep networks using model averaging. *CoRR*, abs/1602.05629, 2016.

[7] El Mahdi El Mhamdi, Rachid Guerraoui, and Sébastien Rouault. The hidden vulnerability of distributed learning in byzantium, 2018.

[8] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zach 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. *CoRR*, abs/1912.01703, 2019.