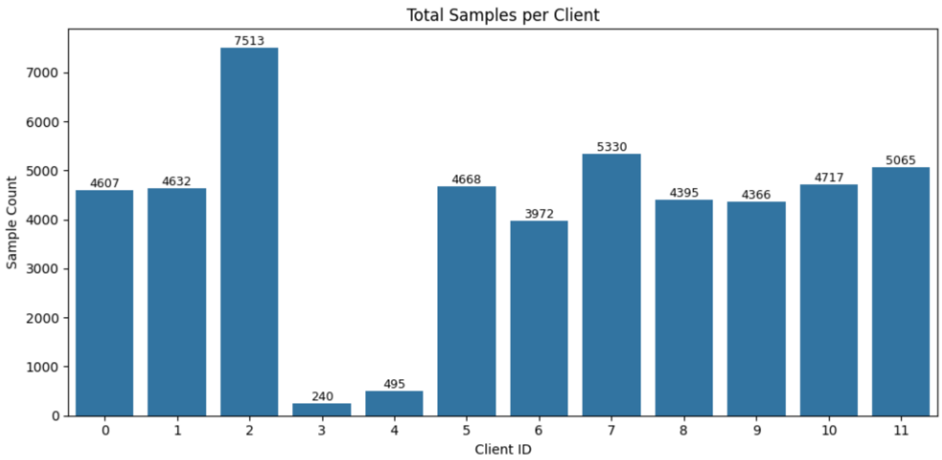
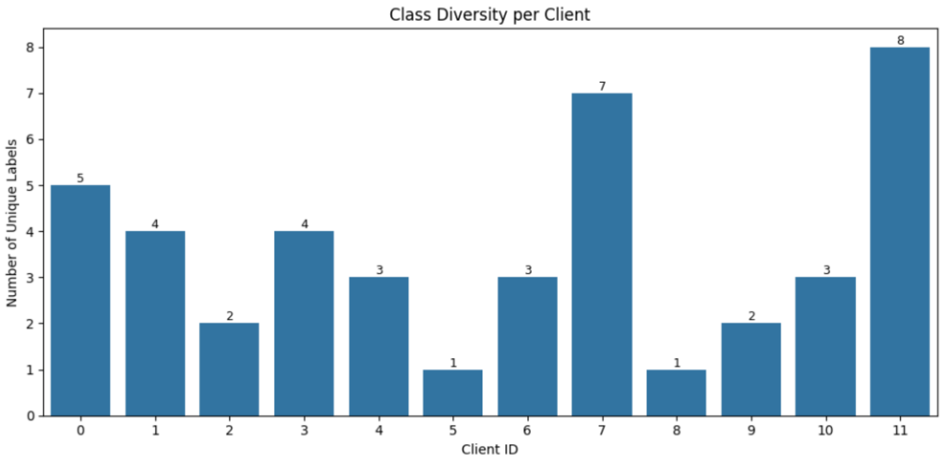
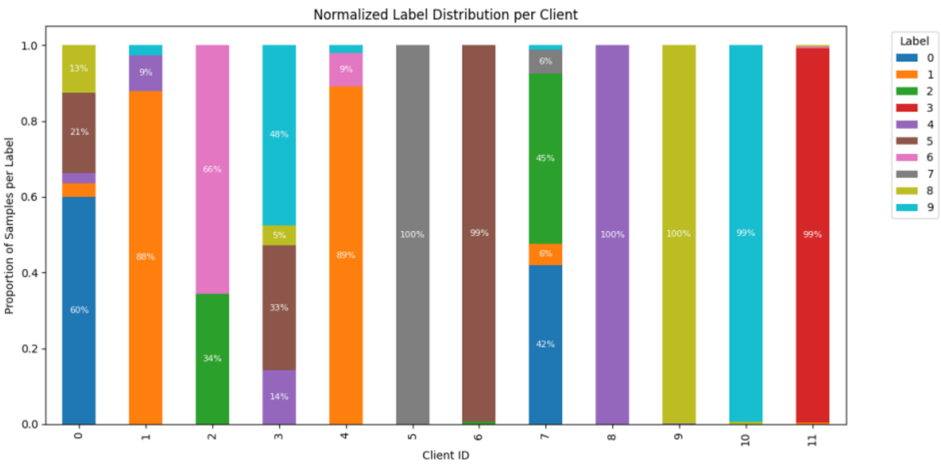
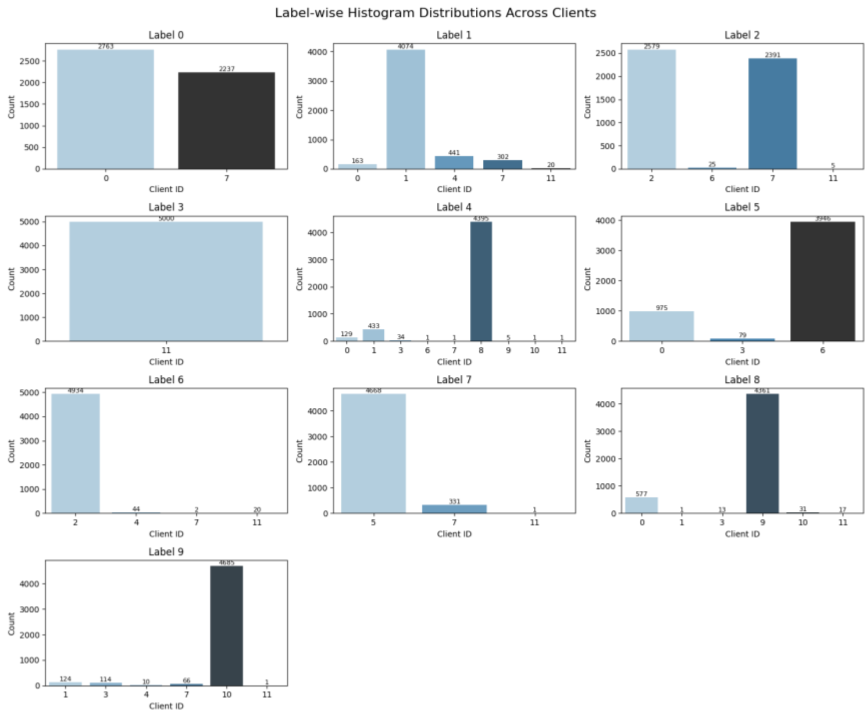
ACM Workshop on Systems for ML (ACM_WS_SYSML) – Pi0

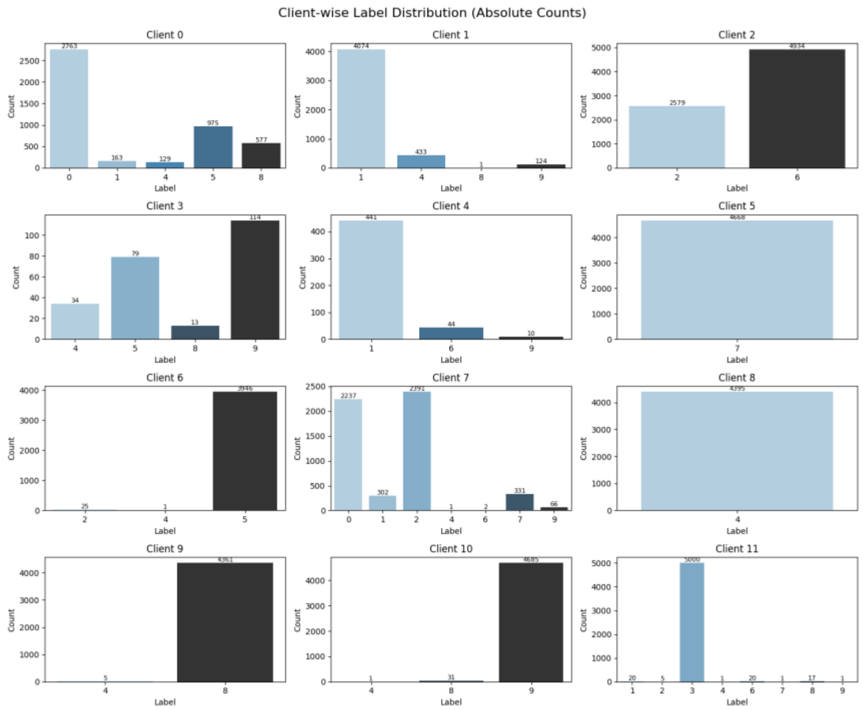
Task 1











Sample Size Heterogeneity

From the "Total Samples per Client" chart, we see that client datasets vary wildly in size. The largest shard, Client 2, holds 7,513 examples, and the next-largest (Clients 7, 11, 10, 5, 0, 1) each have between roughly 4,300 and 5,300 samples. By contrast, Clients 4 and 3 contain only 495 and 240 samples respectively—an order of magnitude less than the largest client. Clients 6, 8, and 9 fall in a mid-range around 3,900–4,400 samples. This imbalance in shard size alone can heavily skew any federated aggregation if left unaddressed, as large clients will dominate parameter updates.

Class Diversity and Imbalance

The "Class Diversity per Client" plot shows that some clients see almost the full label space (Client 11 sees 8 of the 10 classes; Client 7 sees 7), whereas others see only 1 or 2 labels (Clients 5 and 8 each have just a single class; Clients 2 and 9 just two classes). Drilling down with the normalized-distribution chart:

- **Highly concentrated shards**:
    - Client 8 is 100% Label 4.
    - Client 9 is 100% Label 8.
    - Client 5 is 100% Label 7.
    - Client 10 is ~99% Label 9.
    - Client 11 is ~99% Label 3.
- **Moderately diverse shards**:
    - Client 0 spans five labels but is dominated by Label 0 (60%), with the next largest slice Label 5 (21%) and the remainder sparse.
    - Client 3 covers four labels but 81% of its data comes from Labels 9 and 5.
    - Client 7 and Client 1 each have four labels but are heavily skewed toward one or two (e.g. Client 7 is roughly half Label 2 and half Label 0; Client 1 is 88% Label 1).
- **Relatively balanced shards**:
    - Client 11, despite seeing eight classes, still carries severe imbalance toward Label 3 (≈99%).
    - No client has an approximately uniform distribution across more than three labels.

The histogram grids reinforce that each label is almost entirely "owned" by one or two clients (e.g. Label 3 is nearly all from Client 11, Label 4 from Client 8, etc.).

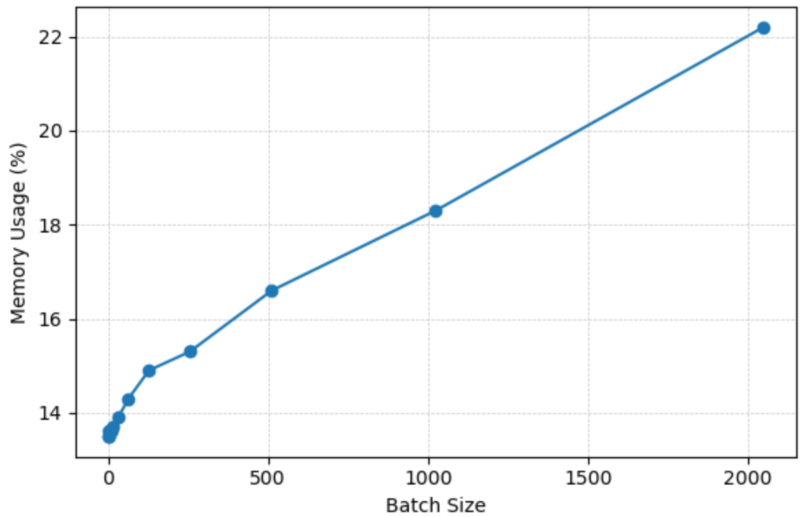
Implications of Size & Class Diversity Differences

Taken together, these plots reveal two stark sources of non-IID heterogeneity:

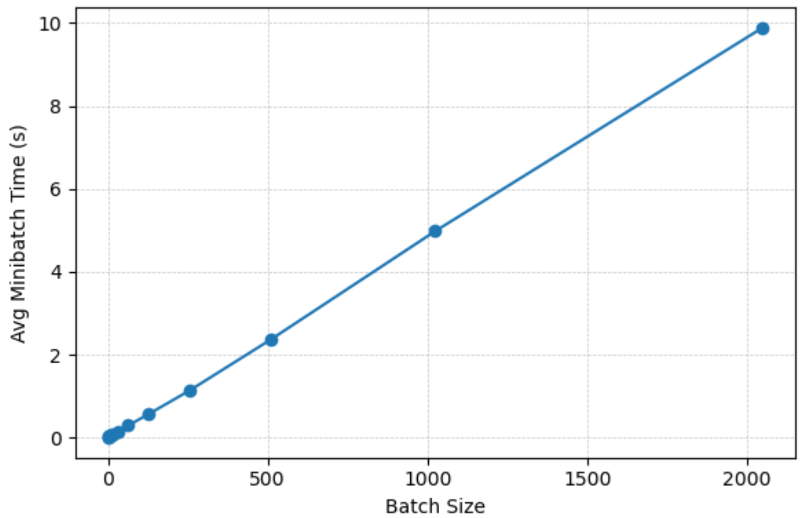
1. **Quantity imbalance**: A few clients (especially Client 2) will dominate global model updates if aggregation is unweighted, while tiny clients (3 and 4) risk being "washed out."
2. **Label imbalance**: Many clients have only one or two classes, and these classes rarely overlap—meaning the model may struggle to generalize across shards without careful reweighting or data-augmentation strategies.

In federated learning, such extreme shard skews typically necessitate techniques like class-balanced loss, shard-aware learning-rate adjustment, or sampling corrections during aggregation to ensure that small or minority-class shards still contribute meaningfully to the final global model.

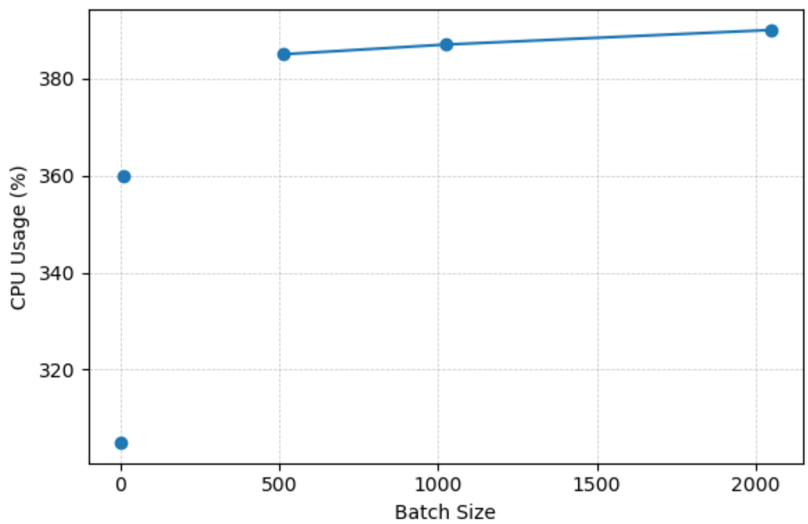
Task 2



22
20
18
16
14
0
500
1000
1500
2000
Batch Size
Memory Usage (%)



10
8
6
4
2
0
0
500
1000
1500
2000
Batch Size
Avg Minibatch Time (s)



380
360
340
320
0
500
1000
1500
2000
Batch Size
CPU Usage (%)

Minibatch-Time varies almost perfectly linearly

- **Observation**: As you double (or more) the batch size, the average time per minibatch grows in near-proportion—e.g. from ~0.017 s at **bs=1** to ~9.9 s at **bs=2048** (a ~580× increase in time for a 2048× increase in batch size).
- **Why**: Larger batches incur more forward/backward passes per step. The GPU (and any CPU preprocessing) must touch every sample, so minibatch time ≈ O(batch_size).

Memory-Footprint climbs steadily but more gently

- **Observation**: GPU-memory use rises from about **13.5%** at bs=1 up to **22.2%** at bs=2048—roughly a 1.6× increase over the entire range.
- **Why**: Memory scales with the size of activations (which grow with batch size), but you still reuse most buffers (so it grows sub-linearly compared to compute time).

CPU utilization plateaus on all cores

- **Observation**: Reported CPU usage (in percent across 4+ cores) climbs from ~305% at bs=1 to ~390% at bs=2048. Beyond bs≈500, the CPU is essentially saturated.
- **Why**: Data-loading, augmentation, and CPU-side bookkeeping saturate all available threads at moderate batch sizes. Pushing batch size higher mostly loads more work onto the GPU, so CPU% only edges up.

1	clients per round	train_bs	rounds	lr	avg minibatch time (s)	memory %	CPU %	diff time wrt prev
2	3	1	100	0.0001	0.017	13.5	305	0.017
3	3	2	100	0.0001	0.027	13.5		0.011
4	3	4	100	0.0001	0.036	13.6		0.009
5	3	8	100	0.0001	0.055	13.6		0.019
6	3	10	100	0.0001	0.063	13.6	360	0.007
7	3	16	100	0.0001	0.088	13.7		0.026
8	3	32	100	0.0001	0.160	13.9		0.071
9	3	64	100	0.0001	0.299	14.3		0.140
10	3	128	100	0.0001	0.574	14.9		0.274
11	3	256	100	0.0001	1.142	15.3		0.569
12	3	512	100	0.0001	2.381	16.6	385	1.239
13	3	1024	100	0.0001	4.982	18.3	387	2.600
14	3	2048	100	0.0001	9.885	22.2	390	4.904

Task 3 (With minibatch size = 10 - Training)

Client ID	# Times Selected	Avg Train Accuracy	Contribution (Assumption)
0	3	~73.6%	Moderate
1	4	~51.1%	Low
2	2	~99.6%	High
3	5	~58.8%	Low
4	4	~79.0%	Moderate
5	2	~98.3%	High
6	1	~98.2%	High (Limited Sample)
7	1	~83.3%	Moderate
8	4	~98.9%	High
9	1	~76.2%	Moderate
10	2	~95.9%	High
11	4	~98.5%	High

Task 4 (Efficient Client Selection Strategy – we think is best!)

Informed Client Selection Strategy

The client_selection_score_based function introduces an informed strategy for selecting clients based on multiple performance and contribution factors. It computes a composite score for each client, reflecting their effectiveness in improving the model relative to the time and data used.

First, for each client, it retrieves the number of times the client was selected (selection_count), the average test accuracy achieved (avg_accuracy), the number of training samples (num_samples), and the average training time per minibatch (avg_minibatch_time). Using this data, the function calculates a score that balances the client's contribution to model accuracy, efficiency (accuracy per unit time), and the quantity of data they hold. Each score is normalized with a small epsilon to avoid division by zero and ensure numerical stability.

Once scores are computed for all clients, the function ranks them and selects the top k clients as determined by args["num_clients_per_round"]. This ensures the server always engages with the most promising subset of clients per round, instead of relying on chance.

Comparative Analysis (Random Selection Vs Our Strategy!)

Aspect	Random Selection	Score-Based Selection (Proposed)
Selection Mechanism	Uniform random sampling	Top-k selection based on composite client score
Metric Awareness	None	Uses accuracy, data size, training speed, and contribution
Historical Data Usage	No	Yes
Client Utility Consideration	Ignored	Explicitly considered via scoring
Training Efficiency	Variable, often suboptimal	Higher, due to prioritization of efficient and impactful clients
Model Convergence	Slower and stochastic	Faster and more stable
Fairness Across Clients	Equal opportunity	Biased toward high-performing clients
Complexity	Low (O(1) per round)	Moderate (O(n) scoring, O(n log n) sorting)
Scalability	Poor performance with large, diverse client sets	Better suited for heterogeneous environments