Recommender

Part 1: Model Training

Root causes

1. Memory blow-ups (17 GB):
- Entire user/restaurant tables were materialized as big NumPy/Torch tensors
2. Too big batch size:
- The required batch=1024 forced large physical batches that don't fit memory
3. Learning rate misconfigured (1e-8) → underfitting
- With only 5 epochs, 1e-8 barely moves the weights: validation accuracy may be below 0.7

Optimizations that respect the constraints

1. Stream the data (small RAM).
- Use an IterableDataset to read parquet per file and yield batches—no full dataset in memory.
2. Train with micro_batch=128 instead
3. Set LR to 1e-5, for increasing convergence speed
4. Use standardization for faster, stabler optimization. With similar feature scales, the loss surface is better conditioned → gradients aren't tiny/huge → fewer exploding/vanishing activations, less LR fiddling, quicker convergence.

Loader Progress: 52934it [00:44, 1176.37it/s] | 0/5 [00:00<?, ?it/s]

Epoch 1: loss=0.1191 acc=0.9519 time=45.0s (cap 45s)

Loader Progress: 59538it [00:45, 1323.04it/s] | 1/5 [00:45<03:00, 45.00s/it]

Epoch 2: loss=0.1140 acc=0.9537 time=45.0s (cap 45s)

Loader Progress: 57522it [00:45, 1278.22it/s]████ | 2/5 [01:30<02:15, 45.00s/it]

Epoch 3: loss=0.1127 acc=0.9541 time=45.0s (cap 45s)

Loader Progress: 63833it [00:45, 1418.48it/s]████████████████████ | 3/5 [02:15<01:30, 45.00s/it]

Epoch 4: loss=0.1121 acc=0.9544 time=45.0s (cap 45s)

Loader Progress: 63067it [00:45, 1401.45it/s]████████████████████████ | 4/5 [03:00<00:45, 45.00s/it]

Epoch 5: loss=0.1116 acc=0.9546 time=45.0s (cap 45s)

Epoch Progress: 100%|

| 5/5 [03:45<00:00, 45.00s/it]

Saved model to data/model.pt and scaler to data/scaler.npz

Part 2: Model Serving

SLO. Sustain **≥30 requests/sec for 60s** with **p95 < 100 ms**.

Runtime & packaging.

1. FastAPI + Uvicorn application.
2. Kubernetes (from Part 3) or Docker Compose locally.
3. Parallelism via multiple Uvicorn workers; one Redis client per worker created on startup.

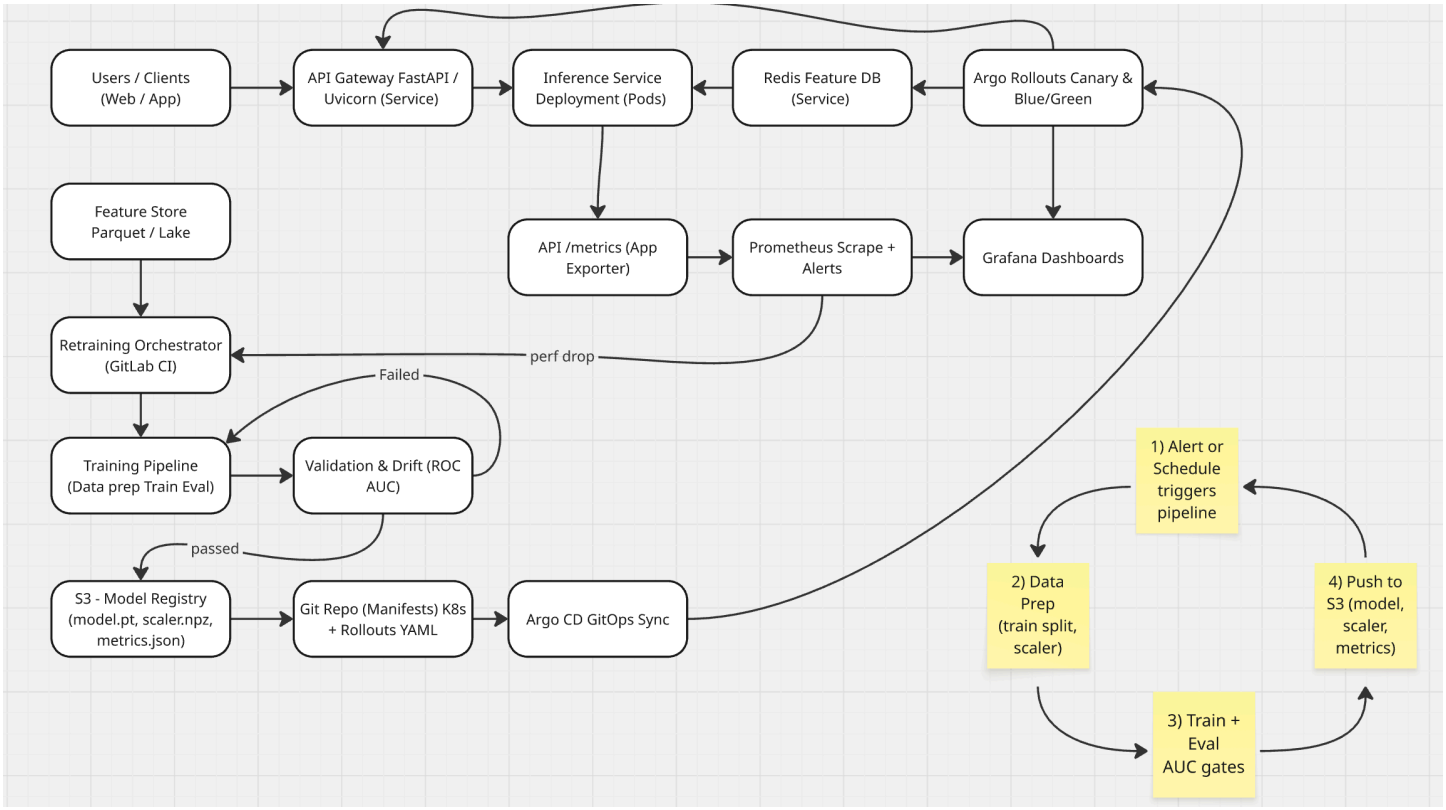
Feature store (database server)

1. Redis used as the online feature DB (server), not SQLite.
2. Loader pushes features from parquet to Redis once, loader_redis.py

Load Test

1. perf.py (with tqdm) drives RPS for a duration and reports p50/p95/p99.
   Configurable with OmegaConf
2. The result is `Requests: 1800 per min| p50=50.5ms p95=92.8ms p99=147.5ms`

Part 3: Model Serving



- API Gateway (FastAPI/Uvicorn): Receives requests, validates inputs, and forwards to the inference Deployment. Exposes /metrics for Prometheus scraping.
- Inference Service (Kubernetes): Scores requests in batches and emits compact prediction events. Uses features from Redis and the latest model from S3.
- Redis Feature DB (Service): Low-latency online store keyed by IDs. Keys are versioned to allow warm cutovers.
- Prometheus + Grafana: Scrape app metrics (latency, errors, throughput) and compute online AUC. Alerts fire when SLOs or accuracy regress.
- Feature Store: Durable parquet/lake tables for training and monitoring.
- Retraining Orchestrator (GitLab CI): Runs on a schedule and on Prometheus alerts. Executes data prep → train → eval, writing artifacts and metrics to S3 only
- S3 Model Registry: Stores model.pt, scaler.npz, metrics.json per version. Promotion writes the chosen S3 URI into the Git manifests
- Validation & Drift: Enforces thresholds (val ROC-AUC). If failed, the run is rejected; if passed, promotion continues
- Argo CD + Argo Rollouts: Syncs manifests and executes canary/blue-green. Auto-rollback if canary underperforms; otherwise flips traffic to 100%.