Project 1:
Federated Learning for Disease Prediction

Contents

1 Introduction

Federated learning (FL) enables collaborative model training across multiple data-owning entities without sharing raw data. In healthcare, this approach is particularly valuable, as patient privacy regulations (e.g., HIPAA, GDPR) prohibit pooling sensitive records into a central location. Instead, each hospital (client) trains a local model on its own patient data. Only model updates (weight vectors, gradients) are shared with a central server, which aggregates them (e.g., via Federated Averaging) to produce a global model. This document describes, in detail, a federated learning system designed to predict disease (diabetes or breast cancer) using healthcare datasets partitioned across multiple simulated hospitals. It covers every step from problem definition to final reporting, including data handling, architecture, training logic, privacy enhancements, and visualization. An architecture diagram using LaTeX `TikZ` is provided to illustrate the communication flow.

2 Step 1: Problem Definition & Dataset Selection

2.1 Objective

- Predict binary disease outcome (diabetes or breast cancer) using patient features.
- Respect privacy: no raw patient data leaves any hospital.
- Achieve high predictive performance (e.g., ROC-AUC ≥ 0.80, recall ≥ 0.75 on positive class).

2.2 Dataset Options

- **Diabetes:** Pima Indians Diabetes dataset (768 records, 8 numeric features, target `Outcome` $\in \{0, 1\}$).
- **Breast Cancer:** Wisconsin Diagnostic Breast Cancer dataset (569 records, 30 numeric features, target `Diagnosis` $\in \{\text{benign}, \text{malignant}\}$).

2.3 Privacy & Compliance

- Appoint a privacy officer to ensure HIPAA/GDPR compliance.
- Draft a Data Protection Impact Assessment (DPIA) covering:
  - Data flow diagrams
  - Threat model (e.g., malicious client attempting inversion attacks)
  - Mitigations (e.g., differential privacy, secure aggregation)
- Record dataset provenance in a register (source URL, download date, version).

2.4 Federated Partitioning

1. Decide on number of simulated hospitals (clients): typically 3–5.
2. Perform a *stratified split* of the entire dataset into N partitions:
   - Preserve class distribution (e.g., positive/negative ratio) across each shard.
   - Assign each patient ID to exactly one shard.
3. Reserve one shard (or a 10–20% hold-out) as a global test set for final evaluation.

4. Produce a mapping table:

$$\text{Patient ID} \longrightarrow \text{Hospital ID} \quad \text{(stored in metadata file)}$$

3 Step 2: Design the Federated Architecture

3.1 Actors & Roles

Client (Hospital)
- Trains a local model on its private shard.
- Evaluates locally on a validation split (optional).
- Exposes a small API (e.g., `/get_parameters`, `/fit`, `/evaluate`).
- Logs local training metrics (loss, accuracy, epoch times).

Server (Aggregator)
- Coordinates FL rounds: notifies clients to fit, collects updates, aggregates.
- Performs Federated Averaging (FedAvg) or alternative aggregation.
- Broadcasts the updated global model to all clients.
- Aggregates optional evaluation metrics (e.g., local validation accuracy) for a dashboard.
- Handles dropouts / stragglers (timeouts, retries).
- Logs per-round global metrics (global loss, global accuracy, participation count).

3.2 Communication & Coordination

- **Framework Choice:** Flower (Python-native ofr FL), with gRPC or HTTP under the hood.
- **Transport:**
  - *Synchronous mode:* Server waits for all N clients (or a predefined fraction) before aggregation.
  - *Asynchronous mode (optional):* Server aggregates as soon as a quorum (e.g., 80%) of clients have reported.
- **Security:**
  1. *Encryption in flight:* TLS for all endpoints (`HTTPS` or gRPC+`TLS`).
  2. *Authentication:* mTLS (mutual TLS) certificates per client server, or token-based (JWT).
  3. *Audit logs:* Each server round logs client IDs, timestamps, and model-version hashes.
  4. *Data at rest:* Each client stores its shard on an encrypted volume (e.g., LUKS).
- **Service Discovery:**
  - *Docker Compose:* Server and clients on the same bridge network, server hostname = `server`.
  - *Kubernetes:* Server exposed via a Service; clients locate server via `service-name.namespace` DNS.

3.3 Non-Functional Requirements

- **Scalability:** Support up to 20–50 clients with minimal latency increase.
- **Resilience:** Server handles up to 30% client drop-out per round by proceeding with available updates after a timeout.
- **Observability:** Every training round is logged, and a lightweight dashboard shows metrics in real time.

4 Step 3: Build Local Client Training Logic

4.1 Model Architecture

- **Baseline Model:** Logistic Regression (interpretable, convex, fast convergence).
- **Neural Network (optional):**
  - Input layer $\rightarrow$ Dense $(32, \text{ReLU})$ $\rightarrow$ Dense $(16, \text{ReLU})$ $\rightarrow$ Output $(1, \text{sigmoid})$.
  - Loss: Binary Cross-Entropy.
  - Optimizer: Adam (learning rate tunable, e.g. 10^{-3}).
- **Hyperparameters:**
  - Epochs per round: 1–5.
  - Batch size: 16–32.
  - Early stopping patience: 1 (stop if validation loss does not improve).
  - Learning rate scheduler (optional).

4.2 Data Handling on Client

- Each client loads its `train_client{i}.npz` from local storage.
- *Optional:* Split further into local training (e.g., 80%) and local validation (20%).
- Features are already standardized (zero mean, unit variance).
- No raw patient identifiers are present (anonymized at preprocessing).

4.3 Training Workflow

1. **Initialization:** Client receives initial global weights from server ($\mathbf{w}^{(0)}$).
2. **Local Fit:**
$$\mathbf{w}^{(t)} \leftarrow \text{LocalTrain}(\mathbf{w}^{(t-1)}, X_{\text{local}}, y_{\text{local}})$$
   - One epoch (or multiple) of training on X_{local}.
   - Compute local loss $\mathcal{L}_{\text{local}}$ and local accuracy.
3. **Return Update:** Client sends updated weights $\mathbf{w}^{(t)}$ and optionally local metrics $\{\mathcal{L}_{\text{local}}, \text{Acc}_{\text{local}}\}$.
4. **Logging:** Append to `client_i.log`:
   - Timestamp, round number, local loss, local accuracy, epoch durations.
5. **Wait for Next Round:** Block until server dispatches next global weights.

4.4 Client-Side Metrics

- **Training Loss per Epoch**
- **Validation Loss & Accuracy** (if a local hold-out is used)
- **Time per Epoch** (profiling for resource usage)
- **Number of Examples** n_i (used to weight aggregation)

4.5 Operational Checklist

- Ensure Docker image builds without errors.
- Verify that each client launches, reads its assigned shard, and trains for one epoch.
- Simulate a dropout: kill a client mid-round, ensure server times out and proceeds.
- Confirm logs are being written to `logs/client_i.log`.

5 Step 4: Implement Server Aggregation (FedAvg)

5.1 Federated Averaging Algorithm

$$\mathbf{w}_{\text{global}}^{(t)} = \frac{1}{N} \sum_{i=1}^{N} \frac{n_i}{n_{\text{total}}} \mathbf{w}_i^{(t)}$$

- N = number of participating clients in round t.
- n_i = number of local training examples at client i.
- $n_{\text{total}} = \sum_{i=1}^{N} n_i$.
- $\mathbf{w}_i^{(t)}$ = local weights from client i after local training.

5.2 Round Coordination

1. **Initial Round:**

   $$\mathbf{w}_{\text{global}}^{(0)} \sim \text{Random Initialization}$$

   Broadcast $\mathbf{w}_{\text{global}}^{(0)}$ to all clients.

2. **Each Round** t:

   (a) Send $\mathbf{w}_{\text{global}}^{(t-1)}$ to all clients with a *StartRound* message.

   (b) Wait (with timeout) for each client's $\mathbf{w}_i^{(t)}$ and $\text{metric}_i^{(t)}$.

   (c) Compute $\mathbf{w}_{\text{global}}^{(t)}$ via weighted average.

   (d) Aggregate any reported metrics (e.g., local validation accuracy).

   (e) Log:

   $$\text{Round } t: \; N_{\text{participated}}, \; \text{time-to-aggregate}, \; \{\text{Acc}_i^{(t)}\}$$

   (f) Broadcast $\mathbf{w}_{\text{global}}^{(t)}$ for next round.

3. **Termination:** After T rounds or if $\Delta \mathcal{L}_{\text{global}} < \varepsilon$.

5.3 Server-Side Logging & Metrics

- `server.log` captures for each round:
  - Timestamp, round index t.
  - Participating client IDs, number $N_{\text{participated}}$.
  - Aggregation time (in seconds).
  - Global training loss, global validation (if clients report).
- Derived metrics written to a CSV for dashboard ingestion:

$$\big[\text{Round},\, N_{\text{participated}},\, \mathcal{L}_{\text{global}},\, \text{Acc}_{\text{global}}\big].$$

5.4 Failure & Straggler Handling

- **Timeout policy:** Wait up to S seconds (e.g., 30 s) for each client. After S, exclude slow/noisy clients and aggregate with available updates.
- **Retry logic:** If a client fails due to transient network error, retry once before excluding.
- **Alerting:** If more than 30% clients drop out repeatedly, notify via email/Slack.

6 Step 5: Evaluate Global Model

6.1 Evaluation Strategies

- **Global Hold-Out Evaluation:**
  - Use `test_global.npz` (20% leftover) to compute: Accuracy, Precision, Recall, F1, ROC-AUC.
- **Leave-One-Client-Out (LOCO):**
  - Take one hospital's shard as an unseen test set; train on the other $N-1$ clients. Rotate through all clients.
  - Check generalization to a truly unseen client.

6.2 Evaluation Metrics

- Accuracy $= \frac{TP+TN}{TP+TN+FP+FN}$.
- Precision $= \frac{TP}{TP+FP}$.
- Recall $= \frac{TP}{TP+FN}$.
- F1-score $= 2 \cdot \frac{\text{Precision}\cdot\text{Recall}}{\text{Precision}+\text{Recall}}$.
- ROC-AUC: Area under the Receiver Operating Characteristic curve.
- **Confusion Matrix** (visualized as a heatmap).
- **Demographic Fairness Checks:** If demographic attributes exist, compare metrics across subpopulations.

6.3 Visualization & Reporting

- *ROC curves* overlaid for:
  - Final global model on `test_global`.
  - Each LOCO evaluation run.
- *Confusion matrix* heatmap for the final global model.
- *Metric summary table*:

Evaluation Set	Accuracy	Recall	ROC-AUC
Test_Global	0.81	0.74	0.85
LOCO (Client 1)	0.79	0.72	0.83
LOCO (Client 2)	0.80	0.75	0.84
LOCO (Client 3)	0.82	0.76	0.86

- *Loss/Accuracy curves* across FL rounds (plotted in dashboard).

7 Step 6: Privacy Enhancements

7.1 Differential Privacy (DP)

1. **Mechanism:** DP-SGD (add calibrated noise to gradients at each local step).
2. **Library:** Use Opacus (for PyTorch) or TensorFlow Privacy (for TensorFlow).
3. **Parameters:**
   - ε (privacy budget) target, e.g. $\varepsilon = 2.0$.
   - δ (probability of failure), typically 10^{-5} or 10^{-6}.
   - Gradient clipping norm C.
   - Noise multiplier σ chosen to meet (ε, δ).
4. **Workflow:**
   (a) Each client applies DP-SGD during its local training steps.
   (b) No changes to server aggregation logic—server still averages noisy updates.
   (c) Track cumulative privacy spend across rounds.
5. **Deliverable:** Privacy accounting report, e.g. "At $T = 5$ rounds, $\varepsilon = 1.8$, $\delta = 10^{-5}$."

7.2 Secure Aggregation

1. **Goal:** Prevent server from reconstructing or inspecting individual client updates.
2. **Techniques:**
   - *Additive Secret Sharing:* Clients split updates into shares; server sums shares to get aggregated update without seeing each.
   - *Homomorphic Encryption:* Clients send encrypted updates; server performs sum under encryption and decrypts only the final sum.
   - *Multi-Party Computation (MPC):* Clients and server jointly compute the aggregate without revealing individual inputs.
3. **Frameworks:**

- PySyft's Secure Aggregation API
- Google's SecAgg protocol (open source)

4. **Workflow:**
   - Each client runs a `SecretSharing` routine on its weight vector $\mathbf{w}_i$.
   - Transmit shares to the server.
   - Server reconstructs $\sum_i \mathbf{w}_i$ without seeing $\mathbf{w}_i$ individually.
5. **Deliverable:** Security proof sketch and runtime overhead benchmarks.

7.3 Compliance Artifacts

- Update DPIA to include DP parameters and secure aggregation design.
- Document key management policy for homomorphic encryption or secret-sharing keys.
- Store audit logs in an immutable ledger (append-only) for regulatory review.

8 Step 7: Dashboard & Reporting

8.1 Dashboard Requirements

- **Real-Time Metrics:**
  - Global loss/accuracy per round.
  - Number of clients participating each round.
  - Per-client local metrics (loss/accuracy) optionally displayed.
- **Technology Stack:**
  - Backend: Lightweight Flask/FastAPI serving metrics JSON from server logs.
  - Frontend: Plotly Dash or simple HTML+JavaScript (D3.js).
  - Hosting: Deployed alongside the server (same or separate container).
- **Visualizations:**
  - Line chart: global loss vs. FL round.
  - Line chart: global accuracy vs. FL round.
  - Bar chart: local training times per client per round.
  - Pie chart: participation fraction of each client over all rounds.
- **Deployment:** Dockerized; environment variables point to `server.log` path.

8.2 Reporting

- **Figures (stored in `reports/figures/`):**
  1. `loss_curve.png`: global loss vs. FL rounds.
  2. `accuracy_curve.png`: global accuracy vs. FL rounds.
  3. `confusion_matrix.png`: final model on `test_global`.
  4. `roc_curve.png`: ROC for global model.
  5. `per_client_bar.png`: local metrics per client.

- **Summary PDF (`reports/federated_summary.pdf`):**
  1. *Section 1: Problem Statement & Data*
     - Motivation for FL in healthcare.
     - Dataset descriptions.
     - Partition strategy summary.
  2. *Section 2: System Architecture*
     - Components: clients, server, communication.
     - Security measures: TLS, DP, secure aggregation.
  3. *Section 3: Modeling & Results*
     - Local model architecture.
     - FedAvg details.
     - Final global metrics vs. baseline (centralized training).
  4. *Section 4: Privacy Analysis*
     - DP accounting.
     - Secure aggregation overhead.
     - Compliance summary.
  5. *Section 5: Next Steps & Conclusions*
     - Potential extensions: multi-modal data (imaging, genomics).
     - Real deployment considerations.
     - Limitations and open questions.

9 Architecture Diagram

Below is a diagram illustrating the high-level federated learning workflow. Three clients (hospitals) train locally on private data and communicate weight updates to a central server, which aggregates and redistributes the global model.

10 Process Flow: Start to Finish

This section details every task and decision point from the project's inception to completion, in chronological order:

1. **Define Objective & Stakeholders**
   - Clarify clinical question: *"Predict diabetes (Outcome) or cancer (Diagnosis) from patient labs."*
   - Identify stakeholders: data scientists, clinicians, privacy officer, DevOps.
   - Set quantitative targets (ROC-AUC, recall).
2. **Assemble Raw Data**
   - Obtain `diabetes.csv` and/or `breast-cancer.csv` from UCI or Kaggle.
   - Verify schema (feature names, types, missing values).
   - Record dataset version and source in a provenance spreadsheet.
3. **Preprocessing & Shard Creation**

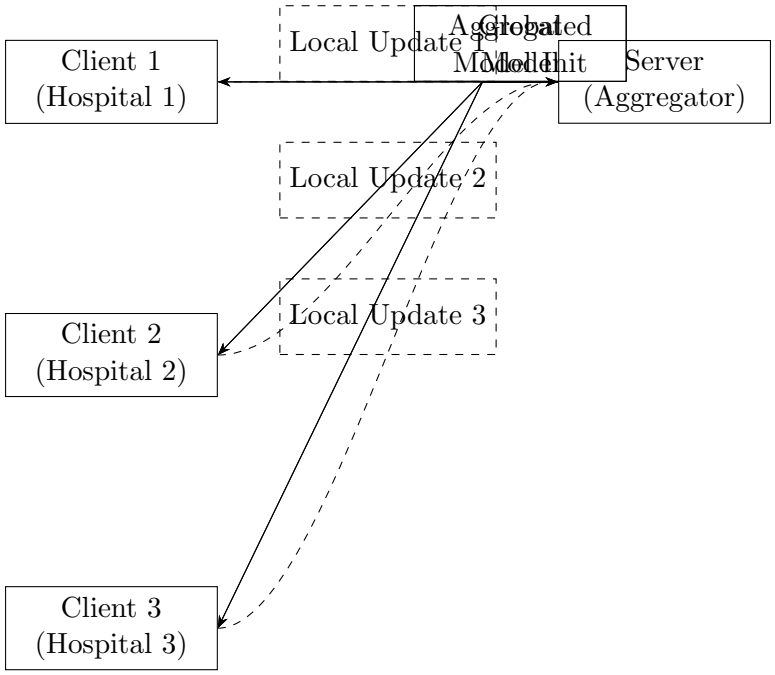

Figure 1: Federated Learning Architecture: Clients train locally, send updates to Server, Server aggregates and broadcasts back.

- Run `scripts/preprocess.py`:
  - Load CSV, drop missing or impute if necessary.
  - Standardize numeric features (zero mean, unit variance).
  - Save combined $\{X, y\}$ to `data/processed/diabetes.npz`.
- Run `scripts/split_data.py` with `--clients` $= N$:
  - Stratified-split indices into N shards.
  - Save each shard to `train_client{i}.npz`.
  - Save global test set to `test_global.npz`.
- Confirm each shard has roughly equal class distribution.

4. **Design FL Architecture**

- Choose Flower as FL framework for simplicity.
- Define client endpoints (`/get_parameters`, `/fit`, `/evaluate`).
- Define server orchestration loop (FedAvg, timeouts, logging).
- Decide on security: TLS, mTLS, or token-based auth.
- Draft architecture and sequence diagrams (see Figure 1).

5. **Implement Client Logic**

- In `client.py`:
  - Load `train_client{i}.npz` $\rightarrow (X_i, y_i)$.
  - Build local model (logistic regression or small NN).
  - Implement `get_parameters()`, `fit()`, `evaluate()` methods.
  - Log local metrics to `logs/client_i.log`.
- Containerize with `Dockerfile.client`:

  - Install dependencies from `requirements.txt`.
  - Copy in `client.py`, `utils.py`, and `data/processed/train_client{i}.npz`.
  - Entrypoint: `python client.py`.
- Validate by running a single client against a dummy server.

6. **Implement Server Logic**
   - In `server.py`:
     - Read environment variables (`MIN_CLIENTS`, `NUM_ROUNDS`).
     - Initialize FedAvg strategy with desired parameters.
     - Start server on host:port.
     - Log each round's metrics to `logs/server.log`.
   - Containerize with `Dockerfile.server`:
     - Install dependencies.
     - Copy `server.py`, `utils.py`.
     - Expose port (e.g., 8080).
   - Validate by running server and 2 clients locally via Docker Compose.
7. **Enable Privacy Enhancements (Optional)**
   - Implement DP-SGD at clients (inject noise into local gradient updates).
   - Integrate secure aggregation (e.g., PySyft or SecAgg) if needed.
   - Update privacy impact assessment with chosen parameters (ε, δ).
   - Benchmark overhead of DP and secure aggregation.
8. **Dashboard Setup**
   - Build a lightweight Flask API inside the server container to expose metrics.
   - Create a simple HTML/JavaScript frontend (or Plotly Dash) to query and plot:
     - Global loss, Global accuracy vs. FL rounds.
     - Client participation per round (bar chart).
     - Local metrics per client (line charts).
   - Containerize dashboard as part of server or as a separate service in Docker Compose.
9. **Run Full FL Experiment**
   - *Via Docker Compose:*
     - `docker-compose up --build` spawns: Server + Client_1 + Client_2 + ... + Client_N.
     - Watch logs in `logs/server.log`, `logs/client_i.log`.
     - Verify that each round's aggregation completes and global model improves.
   - *Via Kubernetes (optional):*
     - Apply `k8s/service.yaml`, `deployment-server.yaml`, `deployment-client.yaml`.
     - Scale clients via `kubectl scale deployment/fl-client --replicas=N`.
     - Monitor pods, ensure client pods mount their respective shards.
10. **Global Model Evaluation**
    - After final round T, save aggregated global weights to a file (e.g., `model_final.h5` or PyTorch checkpoint).

- Load `test_global.npz` $\rightarrow (X_{\text{global\_test}}, y_{\text{global\_test}})$.
- Compute:
$$\text{Accuracy, Precision, Recall, F1, ROC-AUC.}$$
- Generate: confusion matrix heatmap, ROC curve image.
- If LOCO evaluation desired: retrain with one client excluded and test on that client's shard (rotate).

11. **Reporting & Documentation**
    - Populate `reports/figures/` with all generated plots.
    - Compile `reports/federated_summary.pdf` containing:
      - Problem statement, data description, partition details.
      - System architecture diagrams (TikZ figure).
      - Model architecture and hyperparameters.
      - FL training results: loss/accuracy curves, final metrics.
      - Privacy analysis: DP budgets, secure aggregation overhead.
      - Next steps and conclusions.
    - Finalize `docs/architecture.md` and `docs/api_spec.md` with comprehensive diagrams and endpoint definitions.

11 Summary of All Components (Key Points)

Below is a consolidated list of every component, task, and deliverable:

- **Top-Level Files**
  - `README.md`: Project overview, prerequisites, setup, folder summary.
  - `.env.example`: Template for environment variables (`KAGGLE_USERNAME`, `SERVER_HOST`, ...).
  - `.gitignore`: Exclude logs, data/raw, data/processed, environment files.
  - `requirements.txt`: Pin versions of Flower, TensorFlow, scikit-learn, pandas, kaggle CLI.
  - `download_data.sh`: Shell script to fetch raw CSVs from Kaggle and unzip into `data/raw/`.
- **Data Directories**
  - `data/raw/`: Stores original CSVs (unaltered).
    Example: `diabetes.csv`, `breast_cancer.csv`.
  - `data/processed/`: Contains:
    * `diabetes.npz` or `breast_cancer.npz` from `preprocess.py`.
    * `train_client1.npz`, ..., `train_clientN.npz`: stratified shards.
    * `test_global.npz`: global hold-out test set.
- **Notebooks (`notebooks/`)**
  - `01_explore_data.ipynb`:
    * Data profiling, missing-value checks, correlation matrices.
    * Histograms, boxplots for each feature, class balance visualization.

  - `02_preprocess_and_split.ipynb`:
    * Demonstrates running `preprocess.py` and `split_data.py` step-by-step.
    * Verifies class distribution per shard.

- **Scripts** (`scripts/`)
  - `preprocess.py`:
    * Input: `data/raw/<dataset>.csv`.
    * Output: `data/processed/<dataset>.npz`.
    * Actions: drop missing if any, standardize features, save as compressed NumPy.
  - `split_data.py`:
    * Input: `data/processed/<dataset>.npz`, `--clients` $= N$.
    * Output: `train_client{i}.npz` for $i = 1, \ldots, N$, and `test_global.npz`.
    * Partition: stratified by class labels.
  - `evaluate_global_model.py`:
    * Input: final global model checkpoint, `test_global.npz`.
    * Output: metrics CSV, confusion matrix plot, ROC curve.

- **Utility Module** (`utils.py`)
  - `load_data(path)`: loads compressed `.npz`, returns (X, y).
  - `standardize_features(X)`: zero mean, unit variance.
  - `train_validation_split(X,y, val_fraction)`.

- **Federated Client** (`client.py`)
  - Loads `train_client{i}.npz`.
  - Builds local model (Logistic Regression or small NN).
  - Implements Flower's `NumPyClient` interface:
    * `get_parameters()`: return local weights.
    * `fit(parameters, config)`: set weights, train, return updated weights.
    * `evaluate(parameters, config)`: set weights, evaluate on local test split, return loss & accuracy.
  - Writes local logs to `logs/client_i.log`.

- **Federated Server** (`server.py`)
  - Reads environment variables: `MIN_CLIENTS`, `NUM_ROUNDS`, `SERVER_HOST`, `SERVER_PORT`.
  - Configures FedAvg strategy (fraction_fit = 1.0, etc.).
  - Runs `fl.server.start_server` on `0.0.0.0:8080`.
  - Logs per-round metrics to `logs/server.log`.
  - Exposes a small REST endpoint for dashboard metrics (optional).

- **Containerization**
  - `Dockerfile.server`:
    * Base: `python:3.9-slim`.
    * Copy `requirements.txt`, install packages, copy `server.py` and `utils.py`.
    * Expose port 8080, entrypoint `CMD [``python``, ``server.py``]`.

  - `Dockerfile.client`:
    * Base: `python:3.9-slim`.
    * Copy `requirements.txt`, install packages.
    * Copy `client.py`, `utils.py`, and its shard (e.g..
    * Entry-point: `CMD [‘‘python‘‘, ‘‘client.py‘‘]`.
  - `docker-compose.yml`:
    * Services:
      · `server`: build from `Dockerfile.server`, environment {`MIN_CLIENTS`, `NUM_ROUNDS`}.
      · `client1`, `client2`, ...: each builds from `Dockerfile.client`, mounts its own shard, sets `SERVER_ADDRESS`=`server:8080`.
    * Networks: `bridge`, so that `server` resolves as a hostname.

- **Kubernetes Manifests (`k8s/`)**
  - `service.yaml`: Expose server via `LoadBalancer` or `ClusterIP`.
  - `deployment-server.yaml`:
    * Pod with one replica, environment variables from `ConfigMap`/`Secret`.
    * Volume mount: share `utils.py` and logs to a persistent volume.
  - `deployment-client.yaml`:
    * Replica count = number of simulated clients.
    * Each pod mounts its own data shard (via a `ConfigMap` or PVC).
    * Environment variable: `SERVER_ADDRESS` points to `service-server`.

- **Logging & Monitoring (`logs/`)**
  - `server.log`: Each line records:
    1. `timestamp, round_id, n_participating, aggregation_time, global_loss, global_accuracy`
  - `client_i.log`: Each line records:
    1. `timestamp, round_id, local_train_loss, local_train_accuracy, epoch_time`
  - Optionally: integrate Prometheus exporters in server for real-time scraping.

- **Reports (`reports/`)**
  - `figures/`:
    * `loss_curve.png`, `accuracy_curve.png`, `roc_curve.png`, `confusion_matrix.png`, `per_client_bar.png`.
  - `federated_summary.pdf`:
    1. *Section 1: Problem & Data*
    2. *Section 2: Architecture & Security*
    3. *Section 3: Modeling & Results*
    4. *Section 4: Privacy Guarantees*
    5. *Section 5: Next Steps*

- **Documentation (`docs/`)**
  - `architecture.md`:
    * Detailed C4 diagrams: Context, Container, Component views.

    * Sequence diagram for one FL round.
  - `api_spec.md`:
    * Endpoint listing:
      · `GET /parameters`
      · `POST /fit` (payload: client update)
      · `POST /evaluate` (payload: global weights)
      · Health checks, metrics endpoint.
    * Request/response schemas (JSON).
    * Error codes (timeout, invalid payload, auth failure).

- **Privacy & Security Artifacts**
  - *DPIA Document*: Covers threat model, mitigations, data flows.
  - *DP Accounting Report*: Values of ε, δ per FL round.
  - *Secure Aggregation Proof Sketch*: Shows how server cannot invert individual updates.
  - *Key Management Policy*: Describes certificate rotation (for mTLS) and storage (HSM or KMS).

- **Continuous Integration / Continuous Deployment**
  - `.github/workflows/ci.yml` (or `Jenkinsfile`):
    * Lint Python code.
    * Run unit tests:
      · *Client unit tests:* Verify `get_parameters()` returns correct shape, `fit()` updates weights.
      · *Server unit tests:* Simulate 2 dummy clients, verify FedAvg correctness.
    * Build Docker images (`client`, `server`) and push to registry.
  - `.github/workflows/cd.yml`:
    * Deploy to Dev environment (Docker Compose or K8s staging).
    * Smoke test: launch server + 2 ephemeral clients, run one FL round, gather logs.
    * Notify team of success/failure.