# 140509\_37.md - Privacy-Preserving AI Training Framework

## README

**Summary:** Build a framework that enables AI model training while preserving data privacy through techniques like differential privacy, federated learning, and secure computation.

**Problem Statement:** Organizations need to train AI models on sensitive data while maintaining privacy and regulatory compliance. Your task is to create a framework that implements privacy-preserving techniques for AI training including differential privacy, federated learning, and secure multi-party computation. The system should provide privacy guarantees, enable collaborative learning without data sharing, and maintain model utility.

**Steps:** - Design differential privacy mechanisms for various machine learning algorithms - Create secure aggregation and communication protocols for multi-party computation - Build privacy tracking systems - Develop model utility assessment under different privacy constraints - Include compliance reporting and privacy audit capabilities

**Suggested Data Requirements:** - Sensitive training datasets with privacy requirements - Model performance benchmarks under privacy constraints

**Themes:** Responsible AI, AI design that assures Security, Legal and Privacy requirements

## PRD (Product Requirements Document)

### Product Vision

Create a comprehensive privacy-preserving AI training framework that enables organizations to collaborate on AI model development while maintaining strict data privacy and regulatory compliance.

### Target Users

* **Primary:** Data Scientists, ML Engineers, Privacy Officers
* **Secondary:** Healthcare Organizations, Financial Institutions, Government Agencies
* **Tertiary:** Research Institutions, Multi-party Collaborations

### Core Value Propositions

1. **Privacy Guarantees:** Mathematical privacy guarantees with configurable privacy budgets
2. **Collaborative Learning:** Multi-party AI training without data sharing
3. **Regulatory Compliance:** Built-in compliance with GDPR, HIPAA, CCPA
4. **Utility Preservation:** Minimal impact on model performance while ensuring privacy
5. **Scalable Architecture:** Support for large-scale distributed training

### Key Features

1. **Differential Privacy:** Automated DP mechanism design and implementation
2. **Federated Learning:** Secure aggregation and communication protocols
3. **Secure Multi-Party Computation:** Cryptographic protocols for joint computation
4. **Privacy Budget Management:** Automated tracking and optimization
5. **Utility-Privacy Trade-off Analysis:** Comprehensive analysis tools
6. **Compliance Dashboard:** Real-time privacy compliance monitoring

### Success Metrics

* Privacy guarantee strength: Configurable ε values from 0.1 to 10
* Model utility preservation: >90% accuracy retention under strong privacy
* Training efficiency: <2x overhead compared to non-private training
* Compliance coverage: 100% automated compliance with major privacy laws
* Adoption rate: 200+ organizations using framework within 12 months

## FRD (Functional Requirements Document)

### Core Functional Requirements

#### F1: Differential Privacy Implementation

* **F1.1:** Gaussian and Laplacian noise mechanisms for various data types
* **F1.2:** Advanced composition theorems and privacy accounting
* **F1.3:** Private gradient computation with clipping and noise addition
* **F1.4:** Adaptive privacy budget allocation across training epochs
* **F1.5:** Private hyperparameter tuning with privacy budget management

#### F2: Federated Learning Framework

* **F2.1:** Secure aggregation protocols with cryptographic guarantees
* **F2.2:** Client selection and sampling strategies for heterogeneous data
* **F2.3:** Communication-efficient protocols with compression
* **F2.4:** Byzantine-robust aggregation against malicious participants
* **F2.5:** Personalization techniques for non-IID data distributions

#### F3: Secure Multi-Party Computation

* **F3.1:** Secret sharing schemes for distributed computation
* **F3.2:** Homomorphic encryption for private arithmetic operations
* **F3.3:** Garbled circuits for complex private functions
* **F3.4:** Private set intersection for data alignment
* **F3.5:** Threshold cryptography for distributed key management

#### F4: Privacy Budget Management

* **F4.1:** Automated privacy accounting with composition theorems
* **F4.2:** Dynamic privacy budget allocation optimization
* **F4.3:** Privacy budget auditing and compliance reporting
* **F4.4:** Multi-level privacy budgets for hierarchical organizations
* **F4.5:** Privacy budget forecasting and planning tools

#### F5: Utility-Privacy Analysis

* **F5.1:** Comprehensive utility metrics under privacy constraints
* **F5.2:** Privacy-utility Pareto frontier analysis
* **F5.3:** Sensitivity analysis for privacy parameters
* **F5.4:** Model performance benchmarking across privacy levels
* **F5.5:** Automated privacy parameter optimization

#### F6: Compliance and Auditing

* **F6.1:** GDPR Article 25 compliance (Privacy by Design)
* **F6.2:** HIPAA Privacy Rule compliance for healthcare data
* **F6.3:** CCPA compliance for consumer data protection
* **F6.4:** Automated privacy audit trail generation
* **F6.5:** Regulatory reporting and documentation automation

## NFRD (Non-Functional Requirements Document)

### Privacy Requirements

* **NFR-PR1:** Differential privacy guarantees: ε ∈ [0.1, 10] with δ ≤ 10^-6
* **NFR-PR2:** Zero-knowledge proofs for computation correctness
* **NFR-PR3:** Information-theoretic security for secret sharing
* **NFR-PR4:** Semantic security for homomorphic encryption
* **NFR-PR5:** Privacy budget consumption tracking with 99.9% accuracy

### Performance Requirements

* **NFR-P1:** Training overhead: <2x compared to non-private training
* **NFR-P2:** Communication efficiency: <10MB per round in federated setting
* **NFR-P3:** Cryptographic operation latency: <100ms per operation
* **NFR-P4:** Privacy accounting computation: <1 second per update
* **NFR-P5:** Secure aggregation latency: <30 seconds for 1000 participants

### Scalability Requirements

* **NFR-S1:** Support 10,000+ federated learning participants
* **NFR-S2:** Handle datasets up to 1TB with differential privacy
* **NFR-S3:** Multi-party computation with up to 100 parties
* **NFR-S4:** Concurrent training sessions: 100+ simultaneous jobs
* **NFR-S5:** Privacy budget management for 1000+ privacy accounts

### Security Requirements

* **NFR-SE1:** End-to-end encryption for all communications
* **NFR-SE2:** Authenticated key exchange protocols
* **NFR-SE3:** Secure random number generation for noise
* **NFR-SE4:** Protection against timing and side-channel attacks
* **NFR-SE5:** Audit logging with integrity guarantees

## AD (Architecture Diagram)

graph TB  
 subgraph "Client Applications"  
 PYTHON\_SDK[Python SDK]  
 R\_SDK[R SDK]  
 WEB\_UI[Web Interface]  
 CLI[CLI Tools]  
 end  
   
 subgraph "API Gateway & Security"  
 GATEWAY[API Gateway]  
 AUTH[Authentication]  
 AUTHZ[Authorization]  
 AUDIT[Audit Logger]  
 end  
   
 subgraph "Core Privacy Services"  
 DP\_ENGINE[Differential Privacy Engine]  
 FL\_COORDINATOR[Federated Learning Coordinator]  
 SMPC\_ENGINE[Secure Multi-Party Computation]  
 PRIVACY\_BUDGET[Privacy Budget Manager]  
 UTILITY\_ANALYZER[Utility-Privacy Analyzer]  
 end  
   
 subgraph "Privacy Mechanisms"  
 NOISE\_GEN[Noise Generation]  
 GRADIENT\_CLIP[Gradient Clipping]  
 SECURE\_AGG[Secure Aggregation]  
 SECRET\_SHARE[Secret Sharing]  
 HE\_COMPUTE[Homomorphic Encryption]  
 end  
   
 subgraph "Communication Layer"  
 SECURE\_COMM[Secure Communication]  
 KEY\_MGMT[Key Management]  
 CRYPTO\_PROTO[Cryptographic Protocols]  
 P2P\_NETWORK[P2P Network Layer]  
 end  
   
 subgraph "Data Storage"  
 ENCRYPTED\_DB[Encrypted Database]  
 PRIVACY\_LOGS[Privacy Audit Logs]  
 KEY\_STORE[Secure Key Storage]  
 MODEL\_STORE[Model Repository]  
 end  
   
 subgraph "External Integrations"  
 COMPLIANCE\_SYS[Compliance Systems]  
 ML\_PLATFORMS[ML Platforms]  
 MONITORING[Privacy Monitoring]  
 ALERTS[Alert Systems]  
 end  
   
 PYTHON\_SDK --> GATEWAY  
 R\_SDK --> GATEWAY  
 WEB\_UI --> GATEWAY  
 CLI --> GATEWAY  
   
 GATEWAY --> AUTH  
 GATEWAY --> AUTHZ  
 GATEWAY --> AUDIT  
   
 GATEWAY --> DP\_ENGINE  
 GATEWAY --> FL\_COORDINATOR  
 GATEWAY --> SMPC\_ENGINE  
 GATEWAY --> PRIVACY\_BUDGET  
 GATEWAY --> UTILITY\_ANALYZER  
   
 DP\_ENGINE --> NOISE\_GEN  
 DP\_ENGINE --> GRADIENT\_CLIP  
 FL\_COORDINATOR --> SECURE\_AGG  
 SMPC\_ENGINE --> SECRET\_SHARE  
 SMPC\_ENGINE --> HE\_COMPUTE  
   
 FL\_COORDINATOR --> SECURE\_COMM  
 SMPC\_ENGINE --> KEY\_MGMT  
 SECURE\_COMM --> CRYPTO\_PROTO  
 CRYPTO\_PROTO --> P2P\_NETWORK  
   
 DP\_ENGINE --> ENCRYPTED\_DB  
 PRIVACY\_BUDGET --> PRIVACY\_LOGS  
 KEY\_MGMT --> KEY\_STORE  
 UTILITY\_ANALYZER --> MODEL\_STORE  
   
 AUDIT --> COMPLIANCE\_SYS  
 UTILITY\_ANALYZER --> ML\_PLATFORMS  
 PRIVACY\_BUDGET --> MONITORING  
 FL\_COORDINATOR --> ALERTS

## HLD (High Level Design)

### Differential Privacy Engine

class DifferentialPrivacyEngine:  
class HomomorphicEncryptionEngine:  
 def \_\_init\_\_(self):  
 self.key\_generator = HEKeyGenerator()  
 self.encryptor = HEEncryptor()  
 self.evaluator = HEEvaluator()  
 self.decryptor = HEDecryptor()  
   
 def private\_model\_training(self, encrypted\_data, model\_params, training\_config):  
 """Train model on homomorphically encrypted data"""  
   
 # Generate homomorphic encryption keys  
 public\_key, secret\_key, evaluation\_keys = self.key\_generator.generate\_keys(  
 security\_level=training\_config.security\_level  
 )  
   
 # Initialize encrypted model parameters  
 encrypted\_weights = self.encryptor.encrypt\_tensor(  
 model\_params.weights, public\_key  
 )  
   
 for epoch in range(training\_config.epochs):  
 encrypted\_gradients = self.compute\_encrypted\_gradients(  
 encrypted\_data, encrypted\_weights, evaluation\_keys  
 )  
   
 # Update weights homomorphically  
 encrypted\_weights = self.evaluator.subtract(  
 encrypted\_weights,  
 self.evaluator.multiply\_plain(  
 encrypted\_gradients,   
 training\_config.learning\_rate  
 )  
 )  
   
 # Decrypt final weights (only by authorized party)  
 final\_weights = self.decryptor.decrypt\_tensor(encrypted\_weights, secret\_key)  
   
 return HomomorphicTrainingResult(  
 trained\_weights=final\_weights,  
 privacy\_level='semantic\_security',  
 computational\_overhead=self.measure\_he\_overhead()  
 )  
   
 def compute\_encrypted\_gradients(self, encrypted\_data, encrypted\_weights, eval\_keys):  
 """Compute gradients on encrypted data"""  
 encrypted\_predictions = self.evaluator.matrix\_multiply(  
 encrypted\_data, encrypted\_weights, eval\_keys  
 )  
   
 # Compute encrypted loss gradients  
 encrypted\_gradients = self.evaluator.compute\_gradient\_approximation(  
 encrypted\_data, encrypted\_predictions, eval\_keys  
 )  
   
 return encrypted\_gradients  
  
class SecureAggregationProtocol:  
 def \_\_init\_\_(self):  
 self.threshold\_crypto = ThresholdCryptography()  
 self.verifiable\_secret\_sharing = VerifiableSecretSharing()  
   
 def secure\_federated\_aggregation(self, client\_updates, aggregation\_config):  
 """Perform secure aggregation of client model updates"""  
   
 num\_clients = len(client\_updates)  
 threshold = aggregation\_config.threshold or (num\_clients // 2 + 1)  
   
 # Phase 1: Masked model sharing  
 client\_masks = {}  
 shared\_updates = {}  
   
 for i, client\_update in enumerate(client\_updates):  
 # Generate random mask  
 client\_mask = self.generate\_random\_mask(client\_update.shape)  
 client\_masks[i] = client\_mask  
   
 # Mask the client update  
 masked\_update = client\_update + client\_mask  
   
 # Secret share the masked update  
 shares = self.verifiable\_secret\_sharing.share\_secret(  
 masked\_update, threshold, num\_clients  
 )  
 shared\_updates[i] = shares  
   
 # Phase 2: Mask cancellation  
 aggregated\_masks = np.zeros\_like(client\_updates[0])  
   
 for client\_id, mask in client\_masks.items():  
 # Share the mask for cancellation  
 mask\_shares = self.verifiable\_secret\_sharing.share\_secret(  
 mask, threshold, num\_clients  
 )  
   
 # Reconstruct and subtract mask  
 if self.can\_reconstruct\_secret(mask\_shares, threshold):  
 reconstructed\_mask = self.verifiable\_secret\_sharing.reconstruct\_secret(  
 mask\_shares, threshold  
 )  
 aggregated\_masks -= reconstructed\_mask  
   
 # Phase 3: Secure aggregation  
 aggregated\_update = np.zeros\_like(client\_updates[0])  
   
 for client\_id, shares in shared\_updates.items():  
 if self.can\_reconstruct\_secret(shares, threshold):  
 client\_contribution = self.verifiable\_secret\_sharing.reconstruct\_secret(  
 shares, threshold  
 )  
 aggregated\_update += client\_contribution  
   
 # Add back the aggregated masks (they should cancel out)  
 final\_aggregated\_update = aggregated\_update + aggregated\_masks  
   
 return SecureAggregationResult(  
 aggregated\_update=final\_aggregated\_update,  
 participating\_clients=len(client\_updates),  
 privacy\_guarantee='information\_theoretic'  
 )  
  
class PrivacyBudgetManager:  
 def \_\_init\_\_(self):  
 self.budget\_tracker = BudgetTracker()  
 self.composition\_accountant = AdvancedCompositionAccountant()  
   
 def allocate\_privacy\_budget(self, total\_budget, training\_phases):  
 """Optimally allocate privacy budget across training phases"""  
   
 # Analyze the sensitivity of each training phase  
 phase\_sensitivities = {}  
 for phase in training\_phases:  
 sensitivity = self.analyze\_phase\_sensitivity(phase)  
 phase\_sensitivities[phase.id] = sensitivity  
   
 # Solve optimization problem for budget allocation  
 budget\_allocation = self.solve\_budget\_optimization(  
 total\_budget, phase\_sensitivities  
 )  
   
 return budget\_allocation  
   
 def track\_privacy\_consumption(self, mechanism\_type, epsilon, delta):  
 """Track privacy budget consumption with advanced composition"""  
   
 # Update composition using Renyi DP or other advanced methods  
 self.composition\_accountant.compose(  
 mechanism=mechanism\_type,  
 epsilon=epsilon,  
 delta=delta  
 )  
   
 # Check if budget is exceeded  
 current\_budget = self.composition\_accountant.get\_current\_budget()  
   
 if current\_budget.epsilon > self.budget\_tracker.total\_epsilon:  
 raise PrivacyBudgetExceededException(  
 f"Privacy budget exceeded: {current\_budget.epsilon} > {self.budget\_tracker.total\_epsilon}"  
 )  
   
 return current\_budget  
   
 def optimize\_noise\_parameters(self, target\_epsilon, target\_delta, utility\_constraint):  
 """Optimize noise parameters for given privacy and utility constraints"""  
   
 def objective\_function(noise\_params):  
 # Compute expected utility loss  
 utility\_loss = self.estimate\_utility\_loss(noise\_params)  
   
 # Compute privacy guarantee  
 privacy\_epsilon = self.compute\_privacy\_epsilon(noise\_params)  
   
 # Penalize if privacy constraint violated  
 if privacy\_epsilon > target\_epsilon:  
 return float('inf')  
   
 # Penalize if utility constraint violated  
 if utility\_loss > utility\_constraint:  
 return float('inf')  
   
 return utility\_loss  
   
 # Use optimization algorithm to find optimal noise parameters  
 optimal\_params = self.optimize(objective\_function)  
   
 return optimal\_params

### Database Schema

-- Privacy training jobs  
CREATE TABLE privacy\_training\_jobs (  
 id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),  
 job\_name VARCHAR(255) NOT NULL,  
 privacy\_technique VARCHAR(50) NOT NULL, -- 'differential\_privacy', 'federated\_learning', 'secure\_mpc'  
 privacy\_parameters JSONB NOT NULL,  
 dataset\_metadata JSONB NOT NULL,  
 model\_config JSONB NOT NULL,  
 status VARCHAR(50) DEFAULT 'created',  
 created\_by UUID NOT NULL,  
 created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,  
 started\_at TIMESTAMP,  
 completed\_at TIMESTAMP,  
   
 CONSTRAINT valid\_privacy\_technique CHECK (  
 privacy\_technique IN ('differential\_privacy', 'federated\_learning', 'secure\_mpc', 'homomorphic\_encryption')  
 )  
);  
  
-- Privacy budget tracking  
CREATE TABLE privacy\_budgets (  
 id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),  
 training\_job\_id UUID REFERENCES privacy\_training\_jobs(id) ON DELETE CASCADE,  
 total\_epsilon DECIMAL(10, 6) NOT NULL,  
 total\_delta DECIMAL(15, 12) NOT NULL,  
 consumed\_epsilon DECIMAL(10, 6) DEFAULT 0,  
 consumed\_delta DECIMAL(15, 12) DEFAULT 0,  
 composition\_method VARCHAR(50) DEFAULT 'advanced\_composition',  
 budget\_allocation JSONB, -- Per-phase allocation  
 created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,  
 updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP  
);  
  
-- Privacy mechanism executions  
CREATE TABLE privacy\_mechanism\_executions (  
 id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),  
 budget\_id UUID REFERENCES privacy\_budgets(id) ON DELETE CASCADE,  
 mechanism\_type VARCHAR(100) NOT NULL,  
 mechanism\_parameters JSONB NOT NULL,  
 epsilon\_consumed DECIMAL(10, 6) NOT NULL,  
 delta\_consumed DECIMAL(15, 12) NOT NULL,  
 utility\_metrics JSONB,  
 execution\_time\_ms INTEGER,  
 executed\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP  
);  
  
-- Federated learning participants  
CREATE TABLE fl\_participants (  
 id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),  
 training\_job\_id UUID REFERENCES privacy\_training\_jobs(id) ON DELETE CASCADE,  
 participant\_id VARCHAR(255) NOT NULL,  
 participant\_type VARCHAR(50) NOT NULL, -- 'client', 'server', 'coordinator'  
 connection\_info JSONB NOT NULL,  
 data\_characteristics JSONB, -- Size, distribution info  
 participation\_rounds INTEGER[] DEFAULT '{}',  
 status VARCHAR(50) DEFAULT 'registered',  
 registered\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,  
 last\_seen\_at TIMESTAMP  
);  
  
-- Secure computation sessions  
CREATE TABLE secure\_computation\_sessions (  
 id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),  
 training\_job\_id UUID REFERENCES privacy\_training\_jobs(id) ON DELETE CASCADE,  
 session\_type VARCHAR(50) NOT NULL, -- 'secret\_sharing', 'homomorphic\_encryption', 'garbled\_circuits'  
 parties JSONB NOT NULL, -- List of participating parties  
 security\_parameters JSONB NOT NULL,  
 computation\_result\_hash VARCHAR(64),  
 session\_status VARCHAR(50) DEFAULT 'initialized',  
 created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,  
 completed\_at TIMESTAMP  
);  
  
-- Privacy compliance records  
CREATE TABLE privacy\_compliance\_records (  
 id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),  
 training\_job\_id UUID REFERENCES privacy\_training\_jobs(id) ON DELETE CASCADE,  
 regulation\_type VARCHAR(50) NOT NULL, -- 'gdpr', 'hipaa', 'ccpa'  
 compliance\_status VARCHAR(50) NOT NULL,  
 compliance\_details JSONB NOT NULL,  
 audit\_trail JSONB NOT NULL,  
 compliance\_officer UUID,  
 verified\_at TIMESTAMP,  
 expires\_at TIMESTAMP,  
 created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP  
);

## Pseudocode

### Differential Privacy Training Workflow

ALGORITHM PrivacyPreservingTraining  
INPUT: model, dataset, privacy\_config  
OUTPUT: private\_model, privacy\_report  
  
BEGIN  
 // Initialize privacy framework  
 privacy\_engine = SELECT\_PRIVACY\_ENGINE(privacy\_config.technique)  
 privacy\_accountant = PrivacyAccountant(privacy\_config.epsilon, privacy\_config.delta)  
   
 // Validate privacy parameters  
 IF NOT VALIDATE\_PRIVACY\_PARAMETERS(privacy\_config) THEN  
 RETURN ERROR("Invalid privacy parameters")  
 END IF  
   
 SWITCH privacy\_config.technique  
 CASE "differential\_privacy":  
 result = DIFFERENTIAL\_PRIVACY\_TRAINING(model, dataset, privacy\_config, privacy\_accountant)  
 CASE "federated\_learning":  
 result = FEDERATED\_LEARNING\_TRAINING(model, dataset, privacy\_config, privacy\_accountant)  
 CASE "secure\_mpc":  
 result = SECURE\_MPC\_TRAINING(model, dataset, privacy\_config, privacy\_accountant)  
 CASE "homomorphic\_encryption":  
 result = HOMOMORPHIC\_ENCRYPTION\_TRAINING(model, dataset, privacy\_config, privacy\_accountant)  
 DEFAULT:  
 RETURN ERROR("Unsupported privacy technique")  
 END SWITCH  
   
 // Generate privacy compliance report  
 compliance\_report = GENERATE\_COMPLIANCE\_REPORT(result, privacy\_config)  
   
 // Validate model utility  
 utility\_metrics = EVALUATE\_MODEL\_UTILITY(result.private\_model, dataset)  
   
 RETURN PrivacyPreservingResult(  
 private\_model = result.private\_model,  
 privacy\_spent = privacy\_accountant.get\_total\_privacy\_spent(),  
 utility\_metrics = utility\_metrics,  
 compliance\_report = compliance\_report  
 )  
END  
  
FUNCTION DIFFERENTIAL\_PRIVACY\_TRAINING(model, dataset, config, accountant)  
BEGIN  
 dp\_engine = DifferentialPrivacyEngine()  
   
 // Allocate privacy budget across epochs  
 per\_epoch\_budget = ALLOCATE\_PRIVACY\_BUDGET(  
 total\_epsilon = config.epsilon,  
 num\_epochs = config.num\_epochs,  
 allocation\_strategy = config.budget\_allocation  
 )  
   
 private\_model = model.copy()  
   
 FOR epoch IN RANGE(config.num\_epochs) DO  
 epoch\_epsilon = per\_epoch\_budget[epoch]  
   
 // Compute gradient sensitivity  
 gradient\_sensitivity = COMPUTE\_GRADIENT\_SENSITIVITY(private\_model, dataset)  
   
 // Process batches with differential privacy  
 epoch\_gradients = []  
   
 FOR batch IN dataset.batches(config.batch\_size) DO  
 // Compute per-sample gradients  
 per\_sample\_gradients = []  
 FOR sample IN batch DO  
 gradient = private\_model.compute\_gradient(sample)  
 per\_sample\_gradients.APPEND(gradient)  
 END FOR  
   
 // Clip gradients to bound sensitivity  
 clipped\_gradients = CLIP\_GRADIENTS(  
 per\_sample\_gradients,  
 max\_norm = config.max\_gradient\_norm  
 )  
   
 // Add calibrated Gaussian noise  
 noise\_scale = COMPUTE\_NOISE\_SCALE(  
 sensitivity = config.max\_gradient\_norm,  
 epsilon = epoch\_epsilon / dataset.num\_batches,  
 delta = config.delta / (config.num\_epochs \* dataset.num\_batches)  
 )  
   
 noisy\_gradient = ADD\_GAUSSIAN\_NOISE(  
 MEAN(clipped\_gradients),  
 noise\_scale  
 )  
   
 epoch\_gradients.APPEND(noisy\_gradient)  
   
 // Update privacy accountant  
 accountant.add\_mechanism(  
 mechanism = "gaussian\_mechanism",  
 epsilon = epoch\_epsilon / dataset.num\_batches,  
 delta = config.delta / (config.num\_epochs \* dataset.num\_batches)  
 )  
 END FOR  
   
 // Update model with noisy gradients  
 aggregated\_gradient = MEAN(epoch\_gradients)  
 private\_model.update\_weights(aggregated\_gradient, config.learning\_rate)  
   
 // Monitor privacy consumption  
 current\_privacy = accountant.get\_current\_privacy()  
 IF current\_privacy.epsilon > config.epsilon THEN  
 BREAK // Stop training if privacy budget exceeded  
 END IF  
 END FOR  
   
 RETURN DifferentialPrivacyResult(  
 private\_model = private\_model,  
 privacy\_consumed = accountant.get\_current\_privacy(),  
 training\_metrics = GET\_TRAINING\_METRICS()  
 )  
END  
  
FUNCTION FEDERATED\_LEARNING\_TRAINING(global\_model, client\_datasets, config, accountant)  
BEGIN  
 coordinator = FederatedLearningCoordinator()  
 secure\_aggregator = SecureAggregator()  
   
 // Initialize federated learning setup  
 participants = INITIALIZE\_PARTICIPANTS(client\_datasets, config)  
 global\_model\_params = global\_model.get\_parameters()  
   
 FOR round\_num IN RANGE(config.num\_rounds) DO  
 // Select participants for this round  
 selected\_participants = SELECT\_PARTICIPANTS(  
 participants,  
 selection\_fraction = config.client\_fraction,  
 selection\_strategy = config.selection\_strategy  
 )  
   
 // Distribute current global model to selected participants  
 participant\_updates = []  
   
 FOR participant IN selected\_participants DO  
 // Send model to participant  
 SEND\_MODEL\_TO\_PARTICIPANT(global\_model\_params, participant)  
   
 // Participant performs local training with privacy  
 local\_update = PARTICIPANT\_LOCAL\_TRAINING(  
 participant,  
 global\_model\_params,  
 config.local\_training\_config,  
 config.local\_privacy\_config  
 )  
   
 participant\_updates.APPEND(local\_update)  
 END FOR  
   
 // Secure aggregation of participant updates  
 IF config.use\_secure\_aggregation THEN  
 aggregated\_update = secure\_aggregator.secure\_aggregate(  
 participant\_updates,  
 aggregation\_threshold = config.aggregation\_threshold  
 )  
 ELSE  
 aggregated\_update = SIMPLE\_FEDERATED\_AVERAGING(participant\_updates)  
 END IF  
   
 // Apply differential privacy to aggregated update if configured  
 IF config.server\_side\_dp THEN  
 dp\_aggregated\_update = ADD\_DP\_NOISE\_TO\_UPDATE(  
 aggregated\_update,  
 config.server\_epsilon / config.num\_rounds,  
 config.server\_delta / config.num\_rounds  
 )  
   
 accountant.add\_mechanism(  
 mechanism = "gaussian\_mechanism",  
 epsilon = config.server\_epsilon / config.num\_rounds,  
 delta = config.server\_delta / config.num\_rounds  
 )  
 ELSE  
 dp\_aggregated\_update = aggregated\_update  
 END IF  
   
 // Update global model  
 global\_model\_params = UPDATE\_GLOBAL\_MODEL(  
 global\_model\_params,  
 dp\_aggregated\_update,  
 config.server\_learning\_rate  
 )  
   
 // Evaluate global model periodically  
 IF round\_num % config.evaluation\_frequency == 0 THEN  
 global\_metrics = EVALUATE\_GLOBAL\_MODEL(  
 global\_model\_params,  
 config.validation\_data  
 )  
 LOG\_FEDERATED\_METRICS(round\_num, global\_metrics)  
 END IF  
 END FOR  
   
 // Finalize global model  
 final\_global\_model = CREATE\_MODEL\_FROM\_PARAMETERS(global\_model\_params)  
   
 RETURN FederatedLearningResult(  
 global\_model = final\_global\_model,  
 total\_rounds = config.num\_rounds,  
 participating\_clients = participants.length,  
 privacy\_consumed = accountant.get\_current\_privacy(),  
 convergence\_metrics = GET\_CONVERGENCE\_METRICS()  
 )  
END  
  
FUNCTION PARTICIPANT\_LOCAL\_TRAINING(participant, global\_params, local\_config, privacy\_config)  
BEGIN  
 // Initialize local model with global parameters  
 local\_model = CREATE\_MODEL\_FROM\_PARAMETERS(global\_params)  
 local\_dataset = participant.get\_local\_dataset()  
   
 // Apply local differential privacy if configured  
 IF privacy\_config.use\_local\_dp THEN  
 local\_accountant = PrivacyAccountant(  
 privacy\_config.local\_epsilon,  
 privacy\_config.local\_delta  
 )  
   
 FOR local\_epoch IN RANGE(local\_config.local\_epochs) DO  
 // Compute private gradients  
 private\_gradient = COMPUTE\_PRIVATE\_LOCAL\_GRADIENT(  
 local\_model,  
 local\_dataset,  
 privacy\_config.local\_epsilon / local\_config.local\_epochs,  
 privacy\_config.local\_delta / local\_config.local\_epochs  
 )  
   
 // Update local model  
 local\_model.update\_weights(private\_gradient, local\_config.local\_learning\_rate)  
   
 local\_accountant.add\_mechanism(  
 mechanism = "gaussian\_mechanism",  
 epsilon = privacy\_config.local\_epsilon / local\_config.local\_epochs,  
 delta = privacy\_config.local\_delta / local\_config.local\_epochs  
 )  
 END FOR  
 ELSE  
 // Standard local training without privacy  
 FOR local\_epoch IN RANGE(local\_config.local\_epochs) DO  
 gradient = local\_model.compute\_gradient(local\_dataset)  
 local\_model.update\_weights(gradient, local\_config.local\_learning\_rate)  
 END FOR  
 END IF  
   
 // Compute model update (difference from global model)  
 model\_update = COMPUTE\_MODEL\_UPDATE(global\_params, local\_model.get\_parameters())  
   
 RETURN ParticipantUpdate(  
 update = model\_update,  
 data\_size = local\_dataset.size,  
 training\_loss = local\_model.evaluate\_loss(local\_dataset),  
 privacy\_consumed = local\_accountant.get\_current\_privacy() IF privacy\_config.use\_local\_dp ELSE None  
 )  
END  
  
FUNCTION SECURE\_MPC\_TRAINING(model, datasets, config, accountant)  
BEGIN  
 mpc\_engine = SecureMultiPartyComputationEngine()  
   
 // Initialize secure computation protocol  
 parties = INITIALIZE\_MPC\_PARTIES(datasets, config)  
 protocol = SELECT\_MPC\_PROTOCOL(config.security\_model) // secret\_sharing, homomorphic\_encryption, garbled\_circuits  
   
 // Set up secure communication channels  
 secure\_channels = ESTABLISH\_SECURE\_CHANNELS(parties, config.communication\_config)  
   
 SWITCH protocol  
 CASE "secret\_sharing":  
 result = SECRET\_SHARING\_TRAINING(model, parties, config, secure\_channels)  
 CASE "homomorphic\_encryption":  
 result = HOMOMORPHIC\_ENCRYPTION\_TRAINING(model, parties, config, secure\_channels)  
 CASE "garbled\_circuits":  
 result = GARBLED\_CIRCUITS\_TRAINING(model, parties, config, secure\_channels)  
 END SWITCH  
   
 RETURN SecureMPCResult(  
 trained\_model = result.model,  
 computation\_transcript = result.transcript,  
 security\_guarantees = result.security\_level,  
 performance\_metrics = result.performance  
 )  
END  
  
FUNCTION SECRET\_SHARING\_TRAINING(model, parties, config, channels)  
BEGIN  
 secret\_sharing\_scheme = SELECT\_SECRET\_SHARING\_SCHEME(config.threshold, parties.length)  
   
 // Phase 1: Share training data  
 shared\_datasets = {}  
 FOR party IN parties DO  
 data\_shares = secret\_sharing\_scheme.share\_data(  
 party.dataset,  
 threshold = config.threshold,  
 num\_parties = parties.length  
 )  
 shared\_datasets[party.id] = data\_shares  
 END FOR  
   
 // Phase 2: Initialize shared model parameters  
 shared\_model\_params = secret\_sharing\_scheme.share\_data(  
 model.get\_parameters(),  
 threshold = config.threshold,  
 num\_parties = parties.length  
 )  
   
 // Phase 3: Secure training loop  
 FOR epoch IN RANGE(config.num\_epochs) DO  
 // Compute gradients on shared data  
 shared\_gradients = COMPUTE\_GRADIENTS\_ON\_SHARES(  
 shared\_model\_params,  
 shared\_datasets,  
 secret\_sharing\_scheme  
 )  
   
 // Update shared model parameters  
 shared\_model\_params = UPDATE\_SHARED\_PARAMETERS(  
 shared\_model\_params,  
 shared\_gradients,  
 config.learning\_rate,  
 secret\_sharing\_scheme  
 )  
 END FOR  
   
 // Phase 4: Reconstruct final model  
 IF PARTIES\_AGREE\_TO\_RECONSTRUCT(parties, config.threshold) THEN  
 final\_model\_params = secret\_sharing\_scheme.reconstruct\_secret(  
 shared\_model\_params,  
 threshold = config.threshold  
 )  
 final\_model = CREATE\_MODEL\_FROM\_PARAMETERS(final\_model\_params)  
 ELSE  
 final\_model = NULL // Cannot reconstruct without sufficient parties  
 END IF  
   
 RETURN SecretSharingResult(  
 model = final\_model,  
 privacy\_level = "information\_theoretic",  
 reconstruction\_successful = (final\_model IS NOT NULL)  
 )  
END

This completes the comprehensive documentation for Problem Statements 36 and 37. Both solutions provide enterprise-grade architectures for responsible AI, covering bias detection/mitigation and privacy-preserving training frameworks with advanced cryptographic protocols and regulatory compliance.): self.noise\_generator = NoiseGenerator() self.privacy\_accountant = PrivacyAccountant() self.gradient\_clipper = GradientClipper() self.composition\_tracker = CompositionTracker()

def private\_training(self, model, dataset, privacy\_config):  
 # Initialize privacy parameters  
 epsilon = privacy\_config.epsilon  
 delta = privacy\_config.delta  
 epochs = privacy\_config.epochs  
   
 # Allocate privacy budget across epochs  
 per\_epoch\_epsilon = self.privacy\_accountant.allocate\_budget(  
 total\_epsilon=epsilon,  
 total\_epochs=epochs,  
 allocation\_strategy=privacy\_config.allocation\_strategy  
 )  
   
 private\_model = model.copy()  
   
 for epoch in range(epochs):  
 # Compute private gradients  
 private\_gradients = self.compute\_private\_gradients(  
 private\_model,   
 dataset,   
 per\_epoch\_epsilon[epoch],  
 delta / epochs  
 )  
   
 # Update model with private gradients  
 private\_model.update(private\_gradients)  
   
 # Track privacy consumption  
 self.composition\_tracker.add\_mechanism(  
 epsilon=per\_epoch\_epsilon[epoch],  
 delta=delta / epochs  
 )  
   
 # Monitor utility degradation  
 utility\_loss = self.evaluate\_utility\_loss(private\_model, model, dataset)  
   
 return PrivateTrainingResult(  
 model=private\_model,  
 privacy\_spent=self.composition\_tracker.get\_total\_privacy(),  
 utility\_metrics=self.compute\_utility\_metrics(private\_model, dataset)  
 )  
  
def compute\_private\_gradients(self, model, dataset, epsilon, delta):  
 batch\_size = len(dataset) // 10 # Reasonable batch size  
 sensitivity = self.compute\_gradient\_sensitivity(model, dataset)  
   
 private\_gradients = []  
   
 for batch in dataset.batch(batch\_size):  
 # Compute per-sample gradients  
 per\_sample\_grads = model.compute\_per\_sample\_gradients(batch)  
   
 # Clip gradients to bound sensitivity  
 clipped\_grads = self.gradient\_clipper.clip\_gradients(  
 per\_sample\_grads,   
 clip\_norm=sensitivity  
 )  
   
 # Add calibrated noise  
 noise\_scale = self.compute\_noise\_scale(sensitivity, epsilon, delta)  
 noisy\_grads = self.noise\_generator.add\_gaussian\_noise(  
 clipped\_grads,   
 noise\_scale  
 )  
   
 private\_gradients.append(noisy\_grads)  
   
 return np.mean(private\_gradients, axis=0)

class FederatedLearningCoordinator: def **init**(self): self.secure\_aggregator = SecureAggregator() self.client\_manager = ClientManager() self.communication\_protocol = SecureCommunicationProtocol()

def coordinate\_federated\_training(self, global\_model, clients, fl\_config):  
 for round\_num in range(fl\_config.num\_rounds):  
 # Select clients for this round  
 selected\_clients = self.client\_manager.select\_clients(  
 clients,   
 selection\_strategy=fl\_config.client\_selection,  
 fraction=fl\_config.client\_fraction  
 )  
   
 # Distribute global model to selected clients  
 client\_updates = []  
   
 for client in selected\_clients:  
 # Send model to client  
 client\_model = self.communication\_protocol.send\_model(  
 global\_model, client  
 )  
   
 # Client performs local training  
 local\_update = client.local\_training(  
 client\_model,   
 fl\_config.local\_epochs,  
 fl\_config.local\_privacy\_config  
 )  
   
 client\_updates.append(local\_update)  
   
 # Secure aggregation of client updates  
 aggregated\_update = self.secure\_aggregator.aggregate\_updates(  
 client\_updates,  
 aggregation\_weights=self.compute\_aggregation\_weights(selected\_clients)  
 )  
   
 # Update global model  
 global\_model.apply\_update(aggregated\_update)  
   
 # Evaluate global model  
 if round\_num % fl\_config.eval\_frequency == 0:  
 global\_metrics = self.evaluate\_global\_model(global\_model, fl\_config.test\_data)  
   
 return FederatedLearningResult(  
 global\_model=global\_model,  
 training\_metrics=self.get\_training\_history(),  
 privacy\_metrics=self.compute\_privacy\_metrics()  
 )

class SecureMultiPartyComputation: def **init**(self): self.secret\_sharing = SecretSharingScheme() self.homomorphic\_encryption = HomomorphicEncryption() self.garbled\_circuits = GarbledCircuits()

def secure\_joint\_training(self, parties, training\_function, security\_config):  
 # Initialize secure computation protocol  
 if security\_config.protocol == 'secret\_sharing':  
 return self.secret\_sharing\_protocol(parties, training\_function)  
 elif security\_config.protocol == 'homomorphic\_encryption':  
 return self.homomorphic\_encryption\_protocol(parties, training\_function)  
 elif security\_config.protocol == 'garbled\_circuits':  
 return self.garbled\_circuits\_protocol(parties, training\_function)  
   
def secret\_sharing\_protocol(self, parties, training\_function):  
 # Share data using secret sharing  
 shared\_data = []  
 for party in parties:  
 shares = self.secret\_sharing.share\_data(  
 party.data,   
 threshold=len(parties)//2 + 1,  
 num\_parties=len(parties)  
 )  
 shared\_data.append(shares)  
   
 # Perform secure computation on shared data  
 result\_shares = training\_function.compute\_on\_shares(shared\_data)  
   
 # Reconstruct final result  
 final\_result = self.secret\_sharing.reconstruct(  
 result\_shares,   
 threshold=len(parties)//2 + 1  
 )  
   
 return SecureComputationResult(  
 result=final\_result,  
 privacy\_guarantees='information\_theoretic',  
 computation\_overhead=self.measure\_overhead()  
 )

```

## LLD (Low Level Design)

### Advanced Differential Privacy Mechanisms

```python class AdvancedDPMechanisms: def **init**(self): self.composition\_accountant = RenyiDPAccountant() self.sensitivity\_analyzer = SensitivityAnalyzer()

def private\_sgd(self, model, dataset, privacy\_params):  
 """Private Stochastic Gradient Descent with advanced composition"""  
 epsilon, delta = privacy\_params.epsilon, privacy\_params.delta  
   
 # Compute sensitivity  
 l2\_sensitivity = self.sensitivity\_analyzer.compute\_l2\_sensitivity(model)  
   
 # Initialize privacy accountant  
 self.composition\_accountant.initialize(epsilon, delta)  
   
 for epoch in range(privacy\_params.epochs):  
 for batch in dataset.batches(privacy\_params.batch\_size):  
 # Compute per-example gradients  
 per\_example\_grads = []  
 for example in batch:  
 grad = model.compute\_gradient(example)  
 per\_example\_grads.append(grad)  
   
 # Clip gradients  
 clipped\_grads = self.clip\_gradients\_per\_example(  
 per\_example\_grads,   
 privacy\_params.max\_grad\_norm  
 )  
   
 # Compute noise scale using current privacy budget  
 current\_epsilon = self.composition\_accountant.get\_current\_epsilon()  
 noise\_multiplier = self.compute\_noise\_multiplier(  
 target\_epsilon=current\_epsilon,  
 target\_delta=delta,  
 sensitivity=privacy\_params.max\_grad\_norm  
 )  
   
 # Add Gaussian noise  
 noisy\_grad = self.add\_gaussian\_noise(  
 np.mean(clipped\_grads, axis=0),  
 noise\_multiplier \* privacy\_params.max\_grad\_norm  
 )  
   
 # Update model  
 model.update\_weights(noisy\_grad)  
   
 # Update privacy accountant  
 self.composition\_accountant.step(noise\_multiplier)  
   
 return model, self.composition\_accountant.get\_privacy\_spent()  
  
def private\_query\_release(self, dataset, queries, privacy\_budget):  
 """Private query release using exponential mechanism"""  
 query\_results = []  
 epsilon\_per\_query = privacy\_budget / len(queries)  
   
 for query in queries:  
 # Compute utility function  
 def utility\_function(answer):  
 return -abs(query.true\_answer(dataset) - answer)  
   
 # Sample from exponential mechanism  
 private\_answer = self.exponential\_mechanism(  
 dataset,   
 query.answer\_domain,  
 utility\_function,  
 epsilon\_per\_query / 2 # Global sensitivity of utility  
 )  
   
 query\_results.append(private\_answer)  
   
 return query\_results

class HomomorphicEncryptionEngine: def **init**(self