# 140509\_39.md - Synthetic Data Generation Platform

## README

**Summary:** Develop a platform that generates high-quality synthetic data for AI training while preserving statistical properties and privacy characteristics of original datasets.

**Problem Statement:** Organizations need high-quality training data but face privacy, cost, and availability constraints. Your task is to create a synthetic data generation platform that produces realistic datasets maintaining statistical properties, relationships, and patterns of original data while ensuring privacy protection. The system should support various data types, provide quality assessment, and enable data augmentation for improved model training.

**Steps:** - Design generative models for different data types (tabular, time-series, images, text) - Implement statistical property preservation and relationship modeling - Create privacy-preserving synthetic data generation with differential privacy - Build quality assessment metrics and validation frameworks - Develop data augmentation capabilities for improving model performance - Include integration with popular ML training pipelines and data platforms

**Suggested Data Requirements:** - Original datasets for synthesis reference and validation - Statistical property specifications and relationship definitions - Privacy requirements and synthetic data quality criteria - ML model training requirements and performance benchmarks

**Themes:** AI for Data & Data for AI, Synthetic data generation

## PRD (Product Requirements Document)

### Product Vision

Create a comprehensive synthetic data generation platform that enables organizations to generate high-fidelity, privacy-preserving synthetic datasets for AI/ML training while maintaining statistical integrity and regulatory compliance.

### Target Users

* **Primary:** Data Scientists, ML Engineers, Data Engineers
* **Secondary:** Privacy Officers, Research Teams, Product Managers
* **Tertiary:** Healthcare Organizations, Financial Services, Government Agencies

### Core Value Propositions

1. **Privacy Protection:** Generate synthetic data without exposing sensitive information
2. **Statistical Fidelity:** Preserve complex relationships and statistical properties
3. **Scale and Speed:** Generate unlimited synthetic data at enterprise scale
4. **Multi-Modal Support:** Handle tabular, time-series, images, and text data
5. **Regulatory Compliance:** Built-in privacy guarantees and audit trails

### Key Features

1. **Multi-Type Data Generation:** Tabular, time-series, images, text, and graph data
2. **Advanced GANs and Diffusion Models:** State-of-the-art generative architectures
3. **Privacy-Preserving Techniques:** Differential privacy and federated synthesis
4. **Quality Assessment Suite:** Comprehensive evaluation metrics and validation
5. **Relationship Preservation:** Complex dependency and correlation modeling
6. **Integration APIs:** Seamless integration with ML pipelines and data platforms

### Success Metrics

* Statistical fidelity: >95% preservation of original data distributions
* Privacy protection: Formal privacy guarantees with configurable ε-δ bounds
* Generation speed: 10x faster than traditional data collection methods
* Model performance: <5% accuracy loss when training on synthetic vs real data
* Adoption rate: 500+ datasets generated across 100+ organizations

## FRD (Functional Requirements Document)

### Core Functional Requirements

#### F1: Multi-Modal Data Generation

* **F1.1:** Tabular data synthesis with complex relationships and constraints
* **F1.2:** Time-series generation with temporal dependencies and seasonality
* **F1.3:** Image synthesis with style consistency and label preservation
* **F1.4:** Text generation maintaining semantic meaning and style
* **F1.5:** Graph data generation preserving network structure and properties

#### F2: Advanced Generative Models

* **F2.1:** Tabular GANs (CTGAN, CopulaGAN, TableGAN) implementation
* **F2.2:** Variational autoencoders for continuous data generation
* **F2.3:** Diffusion models for high-fidelity image and text synthesis
* **F2.4:** Transformer-based models for sequential data generation
* **F2.5:** Hybrid architectures combining multiple generative approaches

#### F3: Privacy-Preserving Generation

* **F3.1:** Differential privacy mechanisms for all data types
* **F3.2:** Federated synthetic data generation across organizations
* **F3.3:** Membership inference attack protection
* **F3.4:** Attribute inference attack mitigation
* **F3.5:** Privacy budget management and accounting

#### F4: Quality Assessment and Validation

* **F4.1:** Statistical similarity metrics (KS test, Chi-square, correlation)
* **F4.2:** Machine learning efficacy evaluation (train on synthetic, test on real)
* **F4.3:** Privacy risk assessment and re-identification analysis
* **F4.4:** Domain-specific quality metrics and validation rules
* **F4.5:** Automated quality reporting and recommendations

#### F5: Relationship and Constraint Modeling

* **F5.1:** Complex dependency modeling between variables
* **F5.2:** Conditional relationships and hierarchical structures
* **F5.3:** Business rule and constraint enforcement
* **F5.4:** Referential integrity maintenance across related tables
* **F5.5:** Custom constraint specification and validation

#### F6: Integration and Deployment

* **F6.1:** REST APIs for synthetic data generation requests
* **F6.2:** Python/R SDKs for seamless workflow integration
* **F6.3:** Integration with popular ML platforms (MLflow, Kubeflow, SageMaker)
* **F6.4:** Batch and streaming data generation capabilities
* **F6.5:** Export to multiple formats (CSV, Parquet, JSON, HDF5)

## NFRD (Non-Functional Requirements Document)

### Performance Requirements

* **NFR-P1:** Generation speed: 1M synthetic records per hour for tabular data
* **NFR-P2:** Model training time: <24 hours for datasets up to 10M records
* **NFR-P3:** API response time: <30 seconds for small batch requests (<1K records)
* **NFR-P4:** Memory efficiency: Generate datasets 10x larger than original
* **NFR-P5:** Concurrent jobs: Support 100+ simultaneous generation tasks

### Quality Requirements

* **NFR-Q1:** Statistical fidelity: >95% similarity in key statistical measures
* **NFR-Q2:** ML utility preservation: <5% accuracy degradation on downstream tasks
* **NFR-Q3:** Privacy guarantee: Configurable ε-differential privacy (ε ∈ [0.1, 10])
* **NFR-Q4:** Relationship preservation: >90% correlation structure maintenance
* **NFR-Q5:** Constraint satisfaction: 100% compliance with specified business rules

### Scalability Requirements

* **NFR-S1:** Handle datasets up to 1TB in size
* **NFR-S2:** Support 1000+ concurrent users across multi-tenant platform
* **NFR-S3:** Horizontal scaling across GPU clusters for model training
* **NFR-S4:** Auto-scaling based on generation queue length
* **NFR-S5:** Global deployment with data residency compliance

### Privacy and Security Requirements

* **NFR-PR1:** Formal privacy guarantees with mathematical proofs
* **NFR-PR2:** Zero original data leakage in synthetic outputs
* **NFR-PR3:** Secure multi-party computation for federated synthesis
* **NFR-PR4:** End-to-end encryption for all data processing
* **NFR-PR5:** Audit trails for all generation activities

## AD (Architecture Diagram)

graph TB  
 subgraph "Client Interfaces"  
 WEB\_UI[Web Interface]  
 PYTHON\_SDK[Python SDK]  
 R\_SDK[R SDK]  
 REST\_API[REST API]  
 end  
   
 subgraph "API Gateway & Security"  
 API\_GW[API Gateway]  
 AUTH[Authentication]  
 RATE\_LIMIT[Rate Limiter]  
 PRIVACY\_GUARD[Privacy Guard]  
 end  
   
 subgraph "Core Generation Services"  
 DATA\_PROFILER[Data Profiler]  
 MODEL\_SELECTOR[Model Selector]  
 GENERATION\_ENGINE[Generation Engine]  
 QUALITY\_ASSESSOR[Quality Assessor]  
 PRIVACY\_ENGINE[Privacy Engine]  
 end  
   
 subgraph "Generative Models"  
 TABULAR\_GAN[Tabular GANs]  
 VAE[Variational Autoencoders]  
 DIFFUSION[Diffusion Models]  
 TRANSFORMERS[Transformer Models]  
 HYBRID\_MODELS[Hybrid Architectures]  
 end  
   
 subgraph "Quality & Privacy Modules"  
 STAT\_VALIDATOR[Statistical Validator]  
 ML\_EVALUATOR[ML Utility Evaluator]  
 PRIVACY\_ANALYZER[Privacy Risk Analyzer]  
 CONSTRAINT\_CHECKER[Constraint Checker]  
 RELATIONSHIP\_VALIDATOR[Relationship Validator]  
 end  
   
 subgraph "Training Infrastructure"  
 GPU\_CLUSTER[GPU Training Cluster]  
 MODEL\_TRAINER[Model Training Service]  
 HYPERPARAMETER\_OPT[Hyperparameter Optimizer]  
 EXPERIMENT\_TRACKER[Experiment Tracker]  
 end  
   
 subgraph "Data Storage"  
 POSTGRES[PostgreSQL - Metadata]  
 MONGODB[MongoDB - Configurations]  
 S3[Object Storage - Data & Models]  
 REDIS[Redis - Cache]  
 VECTOR\_DB[Vector Database - Embeddings]  
 end  
   
 subgraph "External Integrations"  
 ML\_PLATFORMS[ML Platforms]  
 DATA\_SOURCES[Data Sources]  
 CLOUD\_STORAGE[Cloud Storage APIs]  
 MONITORING[Monitoring Systems]  
 end  
   
 WEB\_UI --> API\_GW  
 PYTHON\_SDK --> API\_GW  
 R\_SDK --> API\_GW  
 REST\_API --> API\_GW  
   
 API\_GW --> AUTH  
 API\_GW --> RATE\_LIMIT  
 API\_GW --> PRIVACY\_GUARD  
   
 API\_GW --> DATA\_PROFILER  
 API\_GW --> MODEL\_SELECTOR  
 API\_GW --> GENERATION\_ENGINE  
 API\_GW --> QUALITY\_ASSESSOR  
 API\_GW --> PRIVACY\_ENGINE  
   
 MODEL\_SELECTOR --> TABULAR\_GAN  
 MODEL\_SELECTOR --> VAE  
 MODEL\_SELECTOR --> DIFFUSION  
 MODEL\_SELECTOR --> TRANSFORMERS  
 MODEL\_SELECTOR --> HYBRID\_MODELS  
   
 QUALITY\_ASSESSOR --> STAT\_VALIDATOR  
 QUALITY\_ASSESSOR --> ML\_EVALUATOR  
 PRIVACY\_ENGINE --> PRIVACY\_ANALYZER  
 GENERATION\_ENGINE --> CONSTRAINT\_CHECKER  
 GENERATION\_ENGINE --> RELATIONSHIP\_VALIDATOR  
   
 GENERATION\_ENGINE --> GPU\_CLUSTER  
 MODEL\_SELECTOR --> MODEL\_TRAINER  
 MODEL\_TRAINER --> HYPERPARAMETER\_OPT  
 MODEL\_TRAINER --> EXPERIMENT\_TRACKER  
   
 DATA\_PROFILER --> POSTGRES  
 MODEL\_SELECTOR --> MONGODB  
 GENERATION\_ENGINE --> S3  
 QUALITY\_ASSESSOR --> REDIS  
 PRIVACY\_ENGINE --> VECTOR\_DB  
   
 GENERATION\_ENGINE --> ML\_PLATFORMS  
 DATA\_PROFILER --> DATA\_SOURCES  
 GENERATION\_ENGINE --> CLOUD\_STORAGE  
 QUALITY\_ASSESSOR --> MONITORING

## HLD (High Level Design)

### Core Generation Architecture

class SyntheticDataPlatform:  
 def \_\_init\_\_(self):  
 self.data\_profiler = DataProfiler()  
 self.model\_selector = ModelSelector()  
 self.generation\_engine = GenerationEngine()  
 self.quality\_assessor = QualityAssessor()  
 self.privacy\_engine = PrivacyEngine()  
   
 async def generate\_synthetic\_data(self, generation\_request):  
 # Step 1: Profile original data  
 data\_profile = await self.data\_profiler.profile\_dataset(  
 generation\_request.source\_data  
 )  
   
 # Step 2: Select optimal generative model  
 model\_config = await self.model\_selector.select\_model(  
 data\_profile, generation\_request.requirements  
 )  
   
 # Step 3: Apply privacy constraints  
 privacy\_config = await self.privacy\_engine.configure\_privacy(  
 generation\_request.privacy\_requirements, data\_profile  
 )  
   
 # Step 4: Generate synthetic data  
 synthetic\_data = await self.generation\_engine.generate(  
 source\_data=generation\_request.source\_data,  
 model\_config=model\_config,  
 privacy\_config=privacy\_config,  
 generation\_params=generation\_request.parameters  
 )  
   
 # Step 5: Quality assessment  
 quality\_report = await self.quality\_assessor.assess\_quality(  
 original\_data=generation\_request.source\_data,  
 synthetic\_data=synthetic\_data,  
 assessment\_criteria=generation\_request.quality\_criteria  
 )  
   
 # Step 6: Privacy risk analysis  
 privacy\_report = await self.privacy\_engine.analyze\_privacy\_risks(  
 original\_data=generation\_request.source\_data,  
 synthetic\_data=synthetic\_data  
 )  
   
 return SyntheticDataResult(  
 synthetic\_data=synthetic\_data,  
 quality\_report=quality\_report,  
 privacy\_report=privacy\_report,  
 generation\_metadata=self.extract\_generation\_metadata()  
 )  
  
class DataProfiler:  
 def \_\_init\_\_(self):  
 self.statistical\_analyzer = StatisticalAnalyzer()  
 self.relationship\_detector = RelationshipDetector()  
 self.constraint\_extractor = ConstraintExtractor()  
   
 async def profile\_dataset(self, dataset):  
 # Basic statistical profiling  
 statistical\_profile = self.statistical\_analyzer.analyze(dataset)  
   
 # Relationship detection  
 relationships = self.relationship\_detector.detect\_relationships(dataset)  
   
 # Constraint extraction  
 constraints = self.constraint\_extractor.extract\_constraints(dataset)  
   
 # Data type analysis  
 column\_types = self.analyze\_column\_types(dataset)  
   
 return DataProfile(  
 statistical\_summary=statistical\_profile,  
 relationships=relationships,  
 constraints=constraints,  
 column\_types=column\_types,  
 data\_shape=dataset.shape,  
 missing\_patterns=self.analyze\_missing\_patterns(dataset)  
 )  
  
class GenerationEngine:  
 def \_\_init\_\_(self):  
 self.model\_factory = ModelFactory()  
 self.constraint\_enforcer = ConstraintEnforcer()  
 self.relationship\_preserver = RelationshipPreserver()  
   
 async def generate(self, source\_data, model\_config, privacy\_config, generation\_params):  
 # Initialize generative model  
 model = self.model\_factory.create\_model(model\_config)  
   
 # Apply privacy-preserving training if needed  
 if privacy\_config.use\_differential\_privacy:  
 model = await self.apply\_differential\_privacy(  
 model, source\_data, privacy\_config  
 )  
   
 # Train generative model  
 trained\_model = await self.train\_generative\_model(  
 model, source\_data, generation\_params.training\_config  
 )  
   
 # Generate synthetic samples  
 raw\_synthetic\_data = await trained\_model.generate(  
 num\_samples=generation\_params.num\_samples,  
 seed=generation\_params.seed  
 )  
   
 # Apply constraint enforcement  
 constrained\_data = self.constraint\_enforcer.enforce\_constraints(  
 raw\_synthetic\_data, model\_config.constraints  
 )  
   
 # Preserve relationships  
 final\_synthetic\_data = self.relationship\_preserver.preserve\_relationships(  
 constrained\_data, model\_config.relationships  
 )  
   
 return final\_synthetic\_data  
  
class TabularGAN:  
 def \_\_init\_\_(self, model\_type='CTGAN'):  
 self.model\_type = model\_type  
 self.generator = None  
 self.discriminator = None  
 self.transformer = DataTransformer()  
   
 async def train(self, data, training\_config):  
 # Transform data for GAN training  
 transformed\_data = self.transformer.fit\_transform(data)  
   
 # Initialize generator and discriminator  
 self.generator = self.build\_generator(transformed\_data.shape[1])  
 self.discriminator = self.build\_discriminator(transformed\_data.shape[1])  
   
 # Training loop  
 for epoch in range(training\_config.epochs):  
 # Train discriminator  
 d\_loss = self.train\_discriminator\_step(transformed\_data)  
   
 # Train generator  
 g\_loss = self.train\_generator\_step(transformed\_data.shape[0])  
   
 if epoch % 100 == 0:  
 print(f"Epoch {epoch}: D\_loss={d\_loss:.4f}, G\_loss={g\_loss:.4f}")  
   
 return self  
   
 async def generate(self, num\_samples, seed=None):  
 if seed is not None:  
 torch.manual\_seed(seed)  
   
 # Generate synthetic samples  
 with torch.no\_grad():  
 noise = torch.randn(num\_samples, self.generator.input\_dim)  
 synthetic\_data = self.generator(noise)  
   
 # Inverse transform to original space  
 synthetic\_df = self.transformer.inverse\_transform(synthetic\_data)  
   
 return synthetic\_df  
  
class DifferentialPrivacySynthesis:  
 def \_\_init\_\_(self):  
 self.privacy\_accountant = PrivacyAccountant()  
 self.noise\_mechanisms = {  
 'gaussian': GaussianMechanism(),  
 'laplacian': LaplacianMechanism()  
 }  
   
 async def generate\_private\_synthetic\_data(self, data, epsilon, delta, mechanism='gaussian'):  
 # Initialize privacy accounting  
 self.privacy\_accountant.initialize(epsilon, delta)  
   
 # Compute data sensitivity  
 sensitivity = self.compute\_data\_sensitivity(data)  
   
 # Select noise mechanism  
 noise\_mechanism = self.noise\_mechanisms[mechanism]  
   
 # Apply noise to sufficient statistics  
 noisy\_statistics = noise\_mechanism.add\_noise(  
 self.compute\_sufficient\_statistics(data),  
 sensitivity,  
 epsilon,  
 delta  
 )  
   
 # Generate synthetic data from noisy statistics  
 synthetic\_data = self.reconstruct\_from\_statistics(  
 noisy\_statistics, data.shape[0] \* 2 # Generate 2x samples  
 )  
   
 # Update privacy accounting  
 self.privacy\_accountant.spend\_budget(epsilon, delta)  
   
 return PrivateSyntheticResult(  
 synthetic\_data=synthetic\_data,  
 privacy\_spent=self.privacy\_accountant.get\_spent\_budget(),  
 privacy\_guarantee=f"({epsilon}, {delta})-differential privacy"  
 )

## LLD (Low Level Design)

### Advanced Quality Assessment

class ComprehensiveQualityAssessor:  
 def \_\_init\_\_(self):  
 self.statistical\_tests = StatisticalTestSuite()  
 self.ml\_evaluator = MLUtilityEvaluator()  
 self.privacy\_analyzer = PrivacyRiskAnalyzer()  
   
 async def comprehensive\_assessment(self, original\_data, synthetic\_data, assessment\_config):  
 quality\_results = {}  
   
 # Statistical fidelity assessment  
 statistical\_results = await self.assess\_statistical\_fidelity(  
 original\_data, synthetic\_data  
 )  
 quality\_results['statistical'] = statistical\_results  
   
 # ML utility assessment  
 ml\_results = await self.assess\_ml\_utility(  
 original\_data, synthetic\_data, assessment\_config.ml\_tasks  
 )  
 quality\_results['ml\_utility'] = ml\_results  
   
 # Privacy risk assessment  
 privacy\_results = await self.assess\_privacy\_risks(  
 original\_data, synthetic\_data  
 )  
 quality\_results['privacy'] = privacy\_results  
   
 # Constraint satisfaction assessment  
 constraint\_results = await self.assess\_constraint\_satisfaction(  
 synthetic\_data, assessment\_config.constraints  
 )  
 quality\_results['constraints'] = constraint\_results  
   
 # Overall quality score  
 overall\_score = self.calculate\_overall\_quality\_score(quality\_results)  
   
 return QualityAssessmentResult(  
 overall\_score=overall\_score,  
 detailed\_results=quality\_results,  
 recommendations=self.generate\_quality\_recommendations(quality\_results)  
 )  
   
 async def assess\_statistical\_fidelity(self, original\_data, synthetic\_data):  
 results = {}  
   
 # Univariate distribution comparison  
 for column in original\_data.columns:  
 if original\_data[column].dtype in ['int64', 'float64']:  
 # KS test for continuous variables  
 ks\_stat, ks\_pvalue = stats.kstest(  
 original\_data[column].dropna(),  
 synthetic\_data[column].dropna()  
 )  
 results[f'{column}\_ks\_test'] = {  
 'statistic': ks\_stat,  
 'p\_value': ks\_pvalue,  
 'similar': ks\_pvalue > 0.05  
 }  
 else:  
 # Chi-square test for categorical variables  
 orig\_counts = original\_data[column].value\_counts()  
 synth\_counts = synthetic\_data[column].value\_counts()  
   
 # Align categories  
 all\_categories = set(orig\_counts.index) | set(synth\_counts.index)  
 orig\_aligned = [orig\_counts.get(cat, 0) for cat in all\_categories]  
 synth\_aligned = [synth\_counts.get(cat, 0) for cat in all\_categories]  
   
 chi2\_stat, chi2\_pvalue = stats.chisquare(synth\_aligned, orig\_aligned)  
 results[f'{column}\_chi2\_test'] = {  
 'statistic': chi2\_stat,  
 'p\_value': chi2\_pvalue,  
 'similar': chi2\_pvalue > 0.05  
 }  
   
 # Correlation structure comparison  
 orig\_corr = original\_data.select\_dtypes(include=[np.number]).corr()  
 synth\_corr = synthetic\_data.select\_dtypes(include=[np.number]).corr()  
   
 correlation\_similarity = self.calculate\_correlation\_similarity(orig\_corr, synth\_corr)  
 results['correlation\_preservation'] = correlation\_similarity  
   
 return StatisticalFidelityResult(  
 univariate\_tests=results,  
 correlation\_similarity=correlation\_similarity,  
 overall\_fidelity=self.calculate\_statistical\_fidelity\_score(results)  
 )  
   
 async def assess\_ml\_utility(self, original\_data, synthetic\_data, ml\_tasks):  
 """Assess ML utility by training models on synthetic data and testing on real data"""  
 utility\_results = {}  
   
 for task in ml\_tasks:  
 if task.type == 'classification':  
 utility\_result = await self.evaluate\_classification\_utility(  
 original\_data, synthetic\_data, task  
 )  
 elif task.type == 'regression':  
 utility\_result = await self.evaluate\_regression\_utility(  
 original\_data, synthetic\_data, task  
 )  
   
 utility\_results[task.name] = utility\_result  
   
 return MLUtilityResult(  
 task\_results=utility\_results,  
 average\_utility\_preservation=np.mean([r.utility\_score for r in utility\_results.values()])  
 )  
   
 async def evaluate\_classification\_utility(self, original\_data, synthetic\_data, task):  
 """Train classifier on synthetic data, test on real data"""  
 from sklearn.model\_selection import train\_test\_split  
 from sklearn.ensemble import RandomForestClassifier  
 from sklearn.metrics import accuracy\_score, f1\_score  
   
 # Prepare data  
 X\_real = original\_data.drop(columns=[task.target\_column])  
 y\_real = original\_data[task.target\_column]  
 X\_synth = synthetic\_data.drop(columns=[task.target\_column])  
 y\_synth = synthetic\_data[task.target\_column]  
   
 # Split real data for testing  
 X\_real\_train, X\_real\_test, y\_real\_train, y\_real\_test = train\_test\_split(  
 X\_real, y\_real, test\_size=0.2, random\_state=42  
 )  
   
 # Train on real data (baseline)  
 baseline\_model = RandomForestClassifier(random\_state=42)  
 baseline\_model.fit(X\_real\_train, y\_real\_train)  
 baseline\_accuracy = accuracy\_score(y\_real\_test, baseline\_model.predict(X\_real\_test))  
   
 # Train on synthetic data  
 synthetic\_model = RandomForestClassifier(random\_state=42)  
 synthetic\_model.fit(X\_synth, y\_synth)  
 synthetic\_accuracy = accuracy\_score(y\_real\_test, synthetic\_model.predict(X\_real\_test))  
   
 # Calculate utility preservation  
 utility\_score = synthetic\_accuracy / baseline\_accuracy  
   
 return ClassificationUtilityResult(  
 baseline\_accuracy=baseline\_accuracy,  
 synthetic\_accuracy=synthetic\_accuracy,  
 utility\_score=utility\_score,  
 utility\_preservation=utility\_score >= 0.95  
 )  
  
class PrivacyEngine:  
 def \_\_init\_\_(self):  
 self.privacy\_accountant = AdvancedPrivacyAccountant()  
 self.attack\_simulators = {  
 'membership\_inference': MembershipInferenceAttack(),  
 'attribute\_inference': AttributeInferenceAttack(),  
 'model\_inversion': ModelInversionAttack()  
 }  
   
 async def analyze\_privacy\_risks(self, original\_data, synthetic\_data):  
 privacy\_results = {}  
   
 # Membership inference attack simulation  
 mia\_result = await self.attack\_simulators['membership\_inference'].simulate\_attack(  
 original\_data, synthetic\_data  
 )  
 privacy\_results['membership\_inference'] = mia\_result  
   
 # Attribute inference attack simulation   
 aia\_result = await self.attack\_simulators['attribute\_inference'].simulate\_attack(  
 original\_data, synthetic\_data  
 )  
 privacy\_results['attribute\_inference'] = aia\_result  
   
 # Distance-based privacy metrics  
 distance\_metrics = await self.calculate\_distance\_based\_privacy(  
 original\_data, synthetic\_data  
 )  
 privacy\_results['distance\_metrics'] = distance\_metrics  
   
 # Overall privacy risk score  
 privacy\_risk\_score = self.calculate\_privacy\_risk\_score(privacy\_results)  
   
 return PrivacyAnalysisResult(  
 individual\_attacks=privacy\_results,  
 overall\_risk\_score=privacy\_risk\_score,  
 privacy\_guaranteed=privacy\_risk\_score < 0.3,  
 recommendations=self.generate\_privacy\_recommendations(privacy\_results)  
 )  
  
# Database Schema  
"""  
-- Synthetic data generation jobs  
CREATE TABLE generation\_jobs (  
 id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),  
 job\_name VARCHAR(255) NOT NULL,  
 user\_id UUID NOT NULL,  
 source\_data\_path TEXT NOT NULL,  
 data\_type VARCHAR(50) NOT NULL, -- 'tabular', 'time\_series', 'image', 'text'  
 generation\_config JSONB NOT NULL,  
 privacy\_config JSONB,  
 status VARCHAR(50) DEFAULT 'pending',  
 created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,  
 started\_at TIMESTAMP,  
 completed\_at TIMESTAMP,  
 error\_message TEXT  
);  
  
-- Generated datasets  
CREATE TABLE generated\_datasets (  
 id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),  
 generation\_job\_id UUID REFERENCES generation\_jobs(id) ON DELETE CASCADE,  
 dataset\_name VARCHAR(255) NOT NULL,  
 file\_path TEXT NOT NULL,  
 file\_size\_bytes BIGINT,  
 num\_records INTEGER,  
 quality\_score DECIMAL(4,3),  
 privacy\_score DECIMAL(4,3),  
 generation\_method VARCHAR(100) NOT NULL,  
 model\_config JSONB NOT NULL,  
 created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP  
);  
  
-- Quality assessments  
CREATE TABLE quality\_assessments (  
 id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),  
 dataset\_id UUID REFERENCES generated\_datasets(id) ON DELETE CASCADE,  
 assessment\_type VARCHAR(50) NOT NULL,  
 assessment\_results JSONB NOT NULL,  
 overall\_score DECIMAL(4,3) NOT NULL,  
 passed\_threshold BOOLEAN NOT NULL,  
 assessment\_date TIMESTAMP DEFAULT CURRENT\_TIMESTAMP  
);  
"""

## Pseudocode

### Synthetic Data Generation Workflow

``` ALGORITHM SyntheticDataGeneration INPUT: source\_data, generation\_requirements OUTPUT: synthetic\_dataset, quality\_report

BEGIN // Step 1: Validate and profile source data data\_profile = PROFILE\_SOURCE\_DATA(source\_data) validation\_result = VALIDATE\_DATA\_REQUIREMENTS(data\_profile, generation\_requirements)

IF NOT validation\_result.is\_valid THEN  
 RETURN ERROR("Data validation failed", validation\_result.errors)  
END IF  
  
// Step 2: Select optimal generative approach  
model\_selection = SELECT\_GENERATIVE\_MODEL(  
 data\_profile,   
 generation\_requirements.quality\_targets,  
 generation\_requirements.privacy\_requirements  
)  
  
// Step 3: Configure privacy constraints  
privacy\_config = CONFIGURE\_PRIVACY\_CONSTRAINTS(  
 generation\_requirements.privacy\_requirements,  
 data\_profile.sensitivity\_analysis  
)  
  
// Step 4: Train generative model  
IF generation\_requirements.use\_pretrained\_model THEN  
 generative\_model = LOAD\_PRETRAINED\_MODEL(model\_selection.model\_id)  
ELSE  
 generative\_model = TRAIN\_GENERATIVE\_MODEL(  
 source\_data,  
 model\_selection.model\_config,  
 privacy\_config  
 )  
END IF  
  
// Step 5: Generate synthetic data  
synthetic\_data = GENERATE\_SYNTHETIC\_SAMPLES(  
 generative\_model,  
 generation\_requirements.num\_samples,  
 generation\_requirements.generation\_constraints  
)  
  
// Step 6: Apply post-processing  
processed\_synthetic\_data = APPLY\_POST\_PROCESSING(  
 synthetic\_data,  
 data\_profile.constraints,  
 generation\_requirements.business\_rules  
)  
  
// Step 7: Comprehensive quality assessment  
quality\_report = ASSESS\_SYNTHETIC\_DATA\_QUALITY(  
 source\_data,  
 processed\_synthetic\_data,  
 generation\_requirements.quality\_criteria  
)  
  
// Step 8: Privacy risk analysis  
privacy\_report = ANALYZE\_PRIVACY\_RISKS(  
 source\_data,  
 processed\_synthetic\_data,  
 privacy\_config  
)  
  
// Step 9: Final validation and optimization  
IF quality\_report.overall\_score < generation\_requirements.min\_quality\_threshold THEN  
 IF generation\_requirements.auto\_optimize THEN  
 optimized\_data = OPTIMIZE\_SYNTHETIC\_DATA(  
 processed\_synthetic\_data,  
 quality\_report.improvement\_suggestions  
 )  
 processed\_synthetic\_data = optimized\_data  
 quality\_report = ASSESS\_SYNTHETIC\_DATA\_QUALITY(source\_data, optimized\_data, generation\_requirements.quality\_criteria)  
 ELSE  
 RETURN WARNING("Quality threshold not met", quality\_report)  
 END IF  
END IF  
  
RETURN SyntheticDataResult(  
 synthetic\_dataset = processed\_synthetic\_data,  
 quality\_report = quality\_report,  
 privacy\_report = privacy\_report,  
 generation\_metadata = EXTRACT\_GENERATION\_METADATA()  
)

END

FUNCTION TRAIN\_GENERATIVE\_MODEL(source\_data, model\_config, privacy\_config) BEGIN SWITCH model\_config.model\_type CASE “tabular\_gan”: model = TRAIN\_TABULAR\_GAN(source\_data, model\_config, privacy\_config) CASE “vae”: model = TRAIN\_VARIATIONAL\_AUTOENCODER(source\_data, model\_config, privacy\_config) CASE “diffusion”: model = TRAIN\_DIFFUSION\_MODEL(source\_data, model\_config, privacy\_config) CASE “transformer”: model = TRAIN\_TRANSFORMER\_MODEL(source\_data, model\_config, privacy\_config) DEFAULT: RAISE UnsupportedModelTypeError(model\_config.model\_type) END SWITCH

RETURN model

END

FUNCTION ASSESS\_SYNTHETIC\_DATA\_QUALITY(original\_data, synthetic\_data, quality\_criteria) BEGIN quality\_results = {}

// Statistical fidelity assessment  
statistical\_results = ASSESS\_STATISTICAL\_FIDELITY(original\_data, synthetic\_data)  
quality\_