Problem Statement 03: Banking Fraud Detection Real-Time Analytics

Summary

Develop a real-time fraud detection system for banking transactions that combines machine learning algorithms with rule-based engines to identify suspicious activities and minimize false positives.

Problem Statement

Financial institutions need to detect fraudulent transactions in real-time while minimizing disruption to legitimate customers. Your task is to build a comprehensive fraud detection system that analyzes transaction patterns, user behavior, and contextual information to identify potentially fraudulent activities. The system should provide risk scores, explanations for decisions, and adaptive learning capabilities to evolve with new fraud patterns.

Steps

• Design a real-time data processing pipeline for transaction analysis • Implement multiple ML models (Random Forest, Isolation Forest, Neural Networks) for fraud detection • Create a rule engine for known fraud patterns and regulatory compliance • Build a risk scoring system with explainable AI features • Develop an alert management system for fraud analysts with case prioritization • Include model monitoring and adaptive learning capabilities for new fraud patterns

Suggested Data Requirements

• Historical transaction data with fraud labels (100K+ transactions) • User profile information and behavioral patterns • Merchant and location data • Device and session information for online transactions

Themes

AI for Industry, Classical AI/ML/DL for prediction

1. Product Overview

Product Name: FraudGuard AI Real-Time Detection Platform
Version: 1.0
Target Market: Commercial banks, credit unions, payment processors, fintech companies, digital wallets

2. Business Objectives

2.1 Primary Objectives

- **Fraud Detection Accuracy**: Achieve >99% fraud detection rate with <0.1% false positive rate
- **Real-Time Processing**: Process transactions within 100ms for real-time decision making
- **Cost Reduction**: Reduce fraud losses by 80-90% and operational costs by 60%
- **Customer Experience**: Minimize legitimate transaction blocks to <0.05% of total transactions

2.2 Secondary Objectives

- **Regulatory Compliance**: 100% compliance with PCI DSS, SOX, Basel III, AML/KYC regulations
- **Adaptive Learning**: Detect new fraud patterns within 24-48 hours of emergence
- **Explainability**: Provide clear explanations for 100% of fraud decisions for regulatory audits
- **ROI Target**: 500% within 24 months through fraud prevention and operational efficiency

3. Target Users

3.1 Primary Users

- **Fraud Analysts**: Real-time monitoring, case investigation, pattern analysis
- **Risk Managers**: Risk assessment, policy configuration, performance monitoring
- **Compliance Officers**: Regulatory reporting, audit trail management, policy enforcement

3.2 Secondary Users

- **Bank Customers**: Transaction notifications, fraud alerts, account security
- **Customer Service Representatives**: Fraud case resolution, customer communication
- **IT Operations**: System monitoring, performance optimization, incident response

3.3 Technical Users

- **Data Scientists**: Model development, performance analysis, feature engineering
- **Security Engineers**: System security, threat analysis, vulnerability management
- **DevOps Engineers**: System deployment, scaling, monitoring, maintenance

4. Key Features

4.1 Real-Time Processing Engine

- **Transaction Stream Processing**: Handle 100,000+ transactions per second
- **Sub-100ms Decision Making**: Real-time fraud scoring and decision within 100ms
- **Multi-Channel Support**: Credit cards, debit cards, wire transfers, ACH, mobile payments
- **Global Transaction Processing**: 24/7 processing across multiple time zones and currencies

4.2 Machine Learning Detection Models

- **Ensemble Model Architecture**: Random Forest, Isolation Forest, Neural Networks, XGBoost
- **Behavioral Analytics**: User spending patterns, location analysis, device fingerprinting
- **Anomaly Detection**: Statistical outlier detection, unsupervised learning for new patterns
- **Deep Learning Models**: LSTM for sequence analysis, autoencoders for anomaly detection

4.3 Rule-Based Engine

- **Regulatory Compliance Rules**: AML/KYC, OFAC screening, transaction limits
- **Known Fraud Pattern Rules**: Velocity checks, geographic impossibility, merchant category restrictions
- **Dynamic Rule Configuration**: Real-time rule updates without system downtime
- **Rule Performance Analytics**: Rule effectiveness tracking and optimization

4.4 Risk Scoring and Explainability

- **Composite Risk Scores**: 0-1000 scale with configurable thresholds
- **Feature Importance Analysis**: SHAP values, LIME explanations for model decisions
- **Decision Audit Trail**: Complete traceability of all decision factors
- **Regulatory Reporting**: Automated compliance reports with decision explanations

4.5 Alert Management and Case Workflow

- **Intelligent Alert Prioritization**: Risk-based case assignment and escalation
- **Fraud Analyst Dashboard**: Real-time case management, investigation tools
- **Automated Case Routing**: Skill-based routing to appropriate analysts
- **SLA Management**: Configurable response time requirements and tracking

4.6 Adaptive Learning and Model Management

- **Continuous Model Training**: Daily model updates with new fraud patterns
- **A/B Testing Framework**: Safe deployment of model improvements
- **Feedback Loop Integration**: Analyst feedback incorporation into model training
- **Model Performance Monitoring**: Real-time accuracy, drift detection, performance alerts

5. Success Metrics

5.1 Fraud Detection Performance

- **True Positive Rate**: >99% fraud detection accuracy
- **False Positive Rate**: <0.1% legitimate transactions flagged
- **Precision**: >95% of flagged transactions are actually fraudulent
- **Recall**: >99% of fraudulent transactions are detected
- **F1-Score**: >97% balanced performance metric

5.2 Operational Performance

- **Processing Latency**: <100ms average transaction processing time
- **System Availability**: 99.99% uptime (max 52 minutes downtime/year)
- **Throughput**: Handle 100,000+ transactions per second peak load
- **Scalability**: Linear scaling to 1M+ transactions per second

5.3 Business Impact

- **Fraud Loss Reduction**: 80-90% reduction in fraud losses
- **Operational Cost Reduction**: 60% reduction in manual review costs
- **Customer Satisfaction**: <0.05% legitimate transaction blocks
- **Regulatory Compliance**: 100% compliance with all applicable regulations

5.4 User Adoption and Efficiency

- **Analyst Productivity**: 300% increase in cases processed per analyst
- **Case Resolution Time**: 70% reduction in average case resolution time
- **Training Time**: <4 hours for new analyst onboarding
- **System Usability**: >95% user satisfaction score

6. Constraints & Assumptions

6.1 Regulatory Constraints

- **PCI DSS Compliance**: Level 1 compliance for payment card data handling
- **Data Privacy**: GDPR, CCPA compliance for customer data protection
- **Financial Regulations**: SOX, Basel III, Dodd-Frank compliance
- **AML/KYC Requirements**: Know Your Customer and Anti-Money Laundering compliance

6.2 Technical Constraints

- **Real-Time Processing**: Sub-100ms response time requirement
- **High Availability**: 99.99% uptime requirement for 24/7 operations
- **Data Security**: End-to-end encryption, secure key management
- **Integration Requirements**: Seamless integration with existing core banking systems

6.3 Data Requirements

- **Historical Data**: Minimum 2 years of transaction history with fraud labels
- **Data Volume**: 100K+ labeled transactions for initial model training
- **Data Quality**: >95% data completeness and accuracy
- **Real-Time Data**: Live transaction feeds from all channels

6.4 Business Assumptions

- **Fraud Pattern Evolution**: New fraud patterns emerge every 3-6 months
- **Model Refresh Frequency**: Models require retraining every 30-90 days
- **Analyst Availability**: 24/7 fraud analyst coverage for critical alerts
- **Budget Allocation**: Sufficient budget for cloud infrastructure and ML compute resources

7. Risk Assessment

7.1 Technical Risks

- **Model Drift**: ML models may degrade over time without continuous training
- **Data Quality Issues**: Poor data quality may impact model performance
- **System Scalability**: Peak transaction volumes may exceed system capacity
- **Integration Complexity**: Complex integration with legacy banking systems

7.2 Business Risks

- **Regulatory Changes**: New regulations may require system modifications
- **Fraud Evolution**: Sophisticated fraud techniques may bypass detection
- **False Positive Impact**: High false positives may impact customer experience
- **Competitive Pressure**: Fraudsters may adapt to detection methods

7.3 Mitigation Strategies

- **Continuous Monitoring**: Real-time model performance monitoring and alerting
- **Data Quality Framework**: Automated data validation and quality checks
- **Scalable Architecture**: Cloud-native architecture with auto-scaling capabilities
- **Regulatory Compliance Program**: Proactive compliance monitoring and updates

8. Implementation Phases

8.1 Phase 1: Foundation (Months 1-3)

- Core infrastructure setup and data pipeline development
- Basic ML models implementation (Random Forest, Isolation Forest)
- Rule engine development with basic fraud patterns
- Initial integration with one transaction channel

8.2 Phase 2: Enhancement (Months 4-6)

- Advanced ML models (Neural Networks, Deep Learning)
- Explainable AI features and risk scoring system
- Multi-channel integration and real-time processing optimization
- Fraud analyst dashboard and case management system

8.3 Phase 3: Optimization (Months 7-9)

- Adaptive learning capabilities and continuous model training
- Advanced analytics and reporting features
- Performance optimization and scalability enhancements
- Full regulatory compliance implementation

8.4 Phase 4: Production (Months 10-12)

- Production deployment with full monitoring
- User training and change management
- Performance tuning and optimization
- Continuous improvement and feature enhancements # Functional Requirements Document (FRD) ## Banking Fraud Detection Real-Time Analytics System

Building upon PRD requirements for detailed functional specifications

ETVX Framework

ENTRY CRITERIA

- ✅ PRD completed and approved by banking and compliance stakeholders
- ✅ Business objectives and success metrics clearly defined (>99% detection, <0.1% false positives)
- ✅ Target users and their operational workflows documented
- ✅ Key product features identified and prioritized for real-time fraud detection
- ✅ Technical feasibility assessment for ML and real-time processing completed
- ✅ Regulatory compliance requirements (PCI DSS, AML/KYC) documented

TASK

Transform PRD business requirements into detailed, testable functional specifications that define exactly what the fraud detection system must do, including real-time transaction processing workflows, ML model behaviors, rule engine logic, user interactions, system integrations, and regulatory compliance features.

VERIFICATION & VALIDATION

Verification Checklist: - [ ] Each functional requirement is traceable to PRD business objectives - [ ] Requirements are unambiguous and testable with specific acceptance criteria - [ ] All fraud analyst and risk manager workflows are covered end-to-end - [ ] Integration points with core banking systems and payment networks defined - [ ] Error handling and edge cases specified for financial transaction processing - [ ] Requirements follow consistent numbering (FR-001, FR-002, etc.) with clear categorization

Validation Criteria: - [ ] Requirements satisfy all PRD success metrics (>99% detection, <100ms processing) - [ ] User personas can achieve their fraud detection goals through defined functions - [ ] System behaviors align with banking regulations and compliance standards - [ ] ML engineering team confirms implementability of all detection requirements - [ ] Requirements review completed with fraud analysts, risk managers, and compliance officers - [ ] Integration requirements validated with core banking system architects

EXIT CRITERIA

- ✅ All functional requirements documented with unique identifiers and acceptance criteria
- ✅ Requirements traceability matrix to PRD completed with full coverage
- ✅ User acceptance criteria defined for each requirement with measurable outcomes
- ✅ Banking system integration and regulatory compliance requirements clearly specified
- ✅ Foundation established for non-functional requirements development

---

Reference to Previous Documents

This FRD translates the business objectives and product features defined in the **PRD** into specific functional requirements: - **PRD Target Users** → Detailed fraud analyst workflows, risk manager interfaces, compliance reporting - **PRD Key Features** → Granular ML model specifications, rule engine logic, real-time processing requirements - **PRD Success Metrics** → Measurable functional capabilities (>99% detection accuracy, <100ms processing, <0.1% false positives) - **PRD Constraints** → Technical integration, regulatory compliance, and real-time processing requirements

1. Real-Time Transaction Processing Module

1.1 Transaction Data Ingestion

- **FR-001**: System SHALL ingest real-time transaction data from multiple channels (credit cards, debit cards, wire transfers, ACH, mobile payments) with <10ms latency
- **FR-002**: System SHALL support transaction volumes of 100,000+ transactions per second with linear scalability
- **FR-003**: System SHALL validate transaction data completeness and format according to ISO 8583 and banking standards
- **FR-004**: System SHALL enrich transaction data with customer profile, merchant information, and historical behavioral patterns within 20ms
- **FR-005**: System SHALL handle transaction data in multiple currencies with real-time exchange rate conversion
- **FR-006**: System SHALL maintain transaction data lineage and audit trail for regulatory compliance

1.2 Real-Time Decision Engine

- **FR-007**: System SHALL process each transaction through ML models and rule engine within 100ms total processing time
- **FR-008**: System SHALL generate risk scores on 0-1000 scale with configurable decision thresholds
- **FR-009**: System SHALL make real-time decisions (APPROVE, DECLINE, REVIEW) based on composite risk assessment
- **FR-010**: System SHALL provide decision explanations with contributing factors and confidence levels

- **FR-011**: System SHALL support transaction routing to appropriate approval workflows based on risk scores
- **FR-012**: System SHALL log all decisions with timestamps, risk factors, and model versions for audit purposes

2. Machine Learning Detection Engine

2.1 Ensemble Model Implementation

- **FR-013**: System SHALL implement Random Forest models for pattern-based fraud detection with >95% accuracy
- **FR-014**: System SHALL implement Isolation Forest models for anomaly detection with <2% false positive rate
- **FR-015**: System SHALL implement Neural Network models for complex pattern recognition with >98% precision
- **FR-016**: System SHALL implement XGBoost models for gradient boosting with optimized feature importance
- **FR-017**: System SHALL combine multiple model predictions using weighted ensemble methods
- **FR-018**: System SHALL provide model-specific confidence scores and feature importance rankings

2.2 Behavioral Analytics Engine

- **FR-019**: System SHALL analyze customer spending patterns including amount, frequency, timing, and merchant categories
- **FR-020**: System SHALL detect geographic anomalies using location-based risk assessment and impossible travel detection
- **FR-021**: System SHALL perform device fingerprinting and behavioral biometrics analysis for online transactions
- **FR-022**: System SHALL analyze transaction velocity patterns and detect unusual spending spikes
- **FR-023**: System SHALL maintain customer behavioral profiles with adaptive learning capabilities
- **FR-024**: System SHALL detect account takeover patterns through behavioral deviation analysis

2.3 Advanced Analytics Features

- **FR-025**: System SHALL implement LSTM neural networks for sequential transaction pattern analysis
- **FR-026**: System SHALL use autoencoders for unsupervised anomaly detection of new fraud patterns
- **FR-027**: System SHALL perform graph analytics to detect fraud rings and connected suspicious activities
- **FR-028**: System SHALL implement time-series analysis for temporal fraud pattern detection
- **FR-029**: System SHALL support feature engineering pipeline with automated feature selection and creation
- **FR-030**: System SHALL provide model interpretability using SHAP values and LIME explanations

3. Rule-Based Engine Module

3.1 Regulatory Compliance Rules

- **FR-031**: System SHALL implement AML/KYC rules for suspicious activity detection and reporting
- **FR-032**: System SHALL perform OFAC and sanctions list screening for all transactions and parties
- **FR-033**: System SHALL enforce transaction limits and thresholds according to regulatory requirements
- **FR-034**: System SHALL detect structuring patterns and currency transaction reporting violations
- **FR-035**: System SHALL implement PEP (Politically Exposed Person) screening and enhanced due diligence
- **FR-036**: System SHALL generate SAR (Suspicious Activity Report) filings automatically when thresholds are met

3.2 Fraud Pattern Rules

- **FR-037**: System SHALL implement velocity rules for transaction frequency and amount limits
- **FR-038**: System SHALL detect geographic impossibility (transactions in different locations within impossible timeframes)
- **FR-039**: System SHALL enforce merchant category restrictions based on customer profiles and risk levels
- **FR-040**: System SHALL detect card testing patterns and small-amount probing transactions
- **FR-041**: System SHALL identify account enumeration attacks and credential stuffing attempts
- **FR-042**: System SHALL detect BIN (Bank Identification Number) attacks and card number generation patterns

3.3 Dynamic Rule Management

- **FR-043**: System SHALL support real-time rule configuration and deployment without system downtime
- **FR-044**: System SHALL provide rule versioning and rollback capabilities for safe rule updates
- **FR-045**: System SHALL track rule performance metrics including hit rates and false positive rates
- **FR-046**: System SHALL support A/B testing of rule changes with controlled rollout capabilities
- **FR-047**: System SHALL provide rule conflict detection and resolution mechanisms
- **FR-048**: System SHALL support rule templates and parameterization for easy configuration

4. Risk Scoring and Explainability Module

4.1 Composite Risk Scoring

- **FR-049**: System SHALL calculate composite risk scores combining ML model outputs and rule engine results
- **FR-050**: System SHALL provide risk score breakdown showing contribution from each component (models, rules, features)
- **FR-051**: System SHALL support configurable risk score thresholds for different decision outcomes
- **FR-052**: System SHALL maintain risk score history and trending analysis for customers and merchants
- **FR-053**: System SHALL provide risk score calibration and validation against historical fraud outcomes
- **FR-054**: System SHALL support risk score normalization across different transaction types and channels

4.2 Explainable AI Features

- **FR-055**: System SHALL provide human-readable explanations for all fraud decisions using natural language
- **FR-056**: System SHALL show top contributing factors for each risk decision with importance rankings
- **FR-057**: System SHALL provide counterfactual explanations showing what would change the decision
- **FR-058**: System SHALL support drill-down analysis from high-level explanations to detailed feature analysis
- **FR-059**: System SHALL generate explanation reports suitable for regulatory audits and customer disputes
- **FR-060**: System SHALL provide explanation consistency validation to ensure stable and reliable explanations

5. Alert Management and Case Workflow Module

5.1 Intelligent Alert Generation

- **FR-061**: System SHALL generate prioritized alerts based on risk scores, customer impact, and business rules
- **FR-062**: System SHALL support configurable alert thresholds and escalation rules
- **FR-063**: System SHALL implement alert deduplication and correlation to reduce analyst workload
- **FR-064**: System SHALL provide alert enrichment with relevant customer, transaction, and

historical context
- **FR-065**: System SHALL support alert suppression rules to reduce noise from known false positive patterns
- **FR-066**: System SHALL track alert response times and SLA compliance metrics

5.2 Case Management System

- **FR-067**: System SHALL provide fraud analyst dashboard with real-time case queue and prioritization
- **FR-068**: System SHALL support case assignment and routing based on analyst skills and workload
- **FR-069**: System SHALL provide case investigation tools including transaction history, customer profiles, and related cases
- **FR-070**: System SHALL support case status tracking and workflow management with configurable states
- **FR-071**: System SHALL provide case collaboration features for team-based investigations
- **FR-072**: System SHALL generate case reports and documentation for regulatory and audit purposes

5.3 Analyst Decision Support

- **FR-073**: System SHALL provide decision recommendation engine to assist analyst case resolution
- **FR-074**: System SHALL display similar historical cases and their outcomes for pattern recognition
- **FR-075**: System SHALL provide one-click actions for common case resolution activities
- **FR-076**: System SHALL support bulk case operations for efficient handling of similar cases
- **FR-077**: System SHALL provide case escalation workflows for complex or high-value cases
- **FR-078**: System SHALL track analyst performance metrics and provide feedback for continuous improvement

6. Adaptive Learning and Model Management Module

6.1 Continuous Model Training

- **FR-079**: System SHALL support automated model retraining with new fraud patterns and analyst feedback
- **FR-080**: System SHALL implement incremental learning capabilities for real-time model updates
- **FR-081**: System SHALL provide model performance monitoring with drift detection and alerting
- **FR-082**: System SHALL support A/B testing framework for safe model deployment and comparison
- **FR-083**: System SHALL maintain model versioning and lineage tracking for audit and rollback purposes
- **FR-084**: System SHALL implement champion-challenger model framework for continuous improvement

6.2 Feedback Loop Integration

- **FR-085**: System SHALL capture analyst feedback on case decisions for model improvement
- **FR-086**: System SHALL incorporate customer dispute outcomes into model training data
- **FR-087**: System SHALL support active learning to identify most valuable samples for labeling
- **FR-088**: System SHALL provide feedback quality validation and analyst performance tracking
- **FR-089**: System SHALL implement reward-based learning to optimize for business outcomes
- **FR-090**: System SHALL support external fraud intelligence integration for enhanced model training

7. Integration and API Module

7.1 Core Banking System Integration

- **FR-091**: System SHALL integrate with core banking systems using secure APIs and messaging protocols
- **FR-092**: System SHALL support real-time transaction authorization and decline capabilities
- **FR-093**: System SHALL provide transaction status updates and decision notifications to

upstream systems
- **FR-094**: System SHALL support batch processing for historical analysis and model training
- **FR-095**: System SHALL implement secure data exchange with encryption and authentication
- **FR-096**: System SHALL provide integration monitoring and error handling with automatic retry mechanisms

7.2 External System Integration

- **FR-097**: System SHALL integrate with payment networks (Visa, Mastercard, ACH) for transaction processing
- **FR-098**: System SHALL connect to external fraud intelligence services and threat feeds
- **FR-099**: System SHALL integrate with identity verification services and KYC providers
- **FR-100**: System SHALL support regulatory reporting system integration for automated compliance
- **FR-101**: System SHALL provide webhook and event-driven integration capabilities
- **FR-102**: System SHALL implement API rate limiting and throttling for external service protection

8. Reporting and Analytics Module

8.1 Operational Reporting

- **FR-103**: System SHALL generate real-time fraud detection performance dashboards
- **FR-104**: System SHALL provide fraud trend analysis and pattern identification reports
- **FR-105**: System SHALL generate analyst performance and productivity reports
- **FR-106**: System SHALL provide system performance and SLA compliance reporting
- **FR-107**: System SHALL support custom report creation with drag-and-drop interface
- **FR-108**: System SHALL provide automated report scheduling and distribution capabilities

8.2 Regulatory and Compliance Reporting

- **FR-109**: System SHALL generate SAR (Suspicious Activity Report) filings with required data elements
- **FR-110**: System SHALL provide audit trail reports for regulatory examinations
- **FR-111**: System SHALL generate model validation and performance reports for regulatory compliance
- **FR-112**: System SHALL provide data lineage and governance reports for compliance verification
- **FR-113**: System SHALL support ad-hoc regulatory inquiry response with rapid data retrieval
- **FR-114**: System SHALL maintain reporting data retention according to regulatory requirements (7+ years) # Non-Functional Requirements Document (NFRD) ## Banking Fraud Detection Real-Time Analytics System

Building upon PRD and FRD for system quality attributes and constraints

ETVX Framework

ENTRY CRITERIA

- ✅ PRD completed with quantified success metrics (>99% detection, <100ms processing, <0.1% false positives)
- ✅ FRD completed with all functional requirements defined (FR-001 to FR-114)
- ✅ Banking system load patterns and transaction volumes documented (100K+ TPS)
- ✅ Regulatory compliance requirements identified (PCI DSS Level 1, SOX, Basel III, AML/KYC)
- ✅ Technology constraints and security requirements documented for financial services
- ✅ Business continuity and disaster recovery requirements established

TASK

Define system quality attributes, performance benchmarks, security requirements, scalability targets, and operational constraints that ensure the fraud detection system can deliver functional requirements with acceptable quality in high-stakes financial environments while meeting stringent regulatory and security standards.

VERIFICATION & VALIDATION

Verification Checklist: - [ ] All NFRs are quantifiable and measurable with specific metrics and thresholds - [ ] Performance targets align with PRD success metrics (<100ms processing, >99% accuracy) - [ ] Security requirements meet financial industry standards (PCI DSS Level 1, SOX compliance) - [ ] Scalability requirements support projected transaction volumes (100K+ TPS) - [ ] Each NFR is traceable to functional requirements and business objectives - [ ] Compliance requirements are comprehensive and auditable for financial regulations

Validation Criteria: - [ ] Performance targets are achievable with proposed ML and real-time architecture - [ ] Security requirements satisfy regulatory compliance and industry best practices - [ ] Scalability projections align with banking transaction growth forecasts - [ ] Availability requirements validated with business continuity needs (99.99% uptime) - [ ] Infrastructure team confirms operational feasibility in financial services environment - [ ] Compliance team validates regulatory adherence and audit requirements

EXIT CRITERIA

- ✅ All quality attributes quantified with specific metrics, thresholds, and measurement methods
- ✅ Performance benchmarks established for each system component with SLA definitions
- ✅ Security and compliance requirements fully documented with implementation guidelines
- ✅ Scalability, reliability, and availability targets defined with monitoring requirements
- ✅ Foundation established for system architecture design with financial services constraints

---

Reference to Previous Documents

This NFRD defines quality attributes and constraints based on **ALL** previous requirements: - **PRD Business Objectives** → Performance targets (>99% detection accuracy, <100ms processing, 80-90% fraud loss reduction) - **PRD Success Metrics** → Quantified NFRs (<100ms processing, 99.99% uptime, <0.1% false positives) - **PRD Target Users** → Usability and accessibility requirements for fraud analysts, risk managers, compliance officers - **PRD Regulatory Constraints** → Security and compliance requirements (PCI DSS, SOX, Basel III, AML/KYC) - **FRD Real-Time Processing (FR-001 to FR-012)** → Performance requirements for transaction processing and decision making - **FRD ML Detection Engine (FR-013 to FR-030)** → Performance requirements for model inference and ensemble processing - **FRD Rule Engine (FR-031 to FR-048)** → Performance requirements for rule evaluation and regulatory compliance - **FRD Risk Scoring (FR-049 to FR-060)** → Performance requirements for explainable AI and decision transparency - **FRD Alert Management (FR-061 to FR-078)** → Scalability requirements for case management and analyst workflows - **FRD Adaptive Learning (FR-079 to FR-090)** → Performance requirements for continuous model training and feedback processing - **FRD Integration (FR-091 to FR-102)** → Reliability requirements for core banking and external system integration - **FRD Reporting (FR-103 to FR-114)** → Performance requirements for real-time analytics and regulatory reporting

1. Performance Requirements

1.1 Real-Time Transaction Processing Performance

- **NFR-001**: Transaction processing latency SHALL be ≤100ms for 99.9% of transactions (P99.9)
- **NFR-002**: ML model inference time SHALL be ≤50ms per transaction for ensemble model predictions
- **NFR-003**: Rule engine evaluation SHALL complete within ≤20ms for all regulatory and fraud rules
- **NFR-004**: Risk score calculation SHALL complete within ≤10ms including explainability features
- **NFR-005**: System SHALL process 100,000+ transactions per second with linear scalability
- **NFR-006**: Decision response time SHALL be ≤5ms for approve/decline decisions to payment networks

1.2 Machine Learning Performance

- **NFR-007**: Model training SHALL complete within 4 hours for daily retraining cycles
- **NFR-008**: Feature engineering pipeline SHALL process 1M+ transactions within 30 minutes
- **NFR-009**: Model deployment SHALL complete within 5 minutes with zero-downtime updates
- **NFR-010**: Batch prediction processing SHALL handle 10M+ transactions within 2 hours
- **NFR-011**: Real-time feature store SHALL serve features within ≤5ms for 99.9% of requests
- **NFR-012**: Model performance monitoring SHALL detect drift within 15 minutes of occurrence

1.3 System Response Time

- **NFR-013**: Fraud analyst dashboard SHALL load within ≤2 seconds for 95% of requests
- **NFR-014**: Case investigation interface SHALL respond within ≤1 second for data retrieval
- **NFR-015**: Report generation SHALL complete within 30 seconds for standard reports
- **NFR-016**: Ad-hoc query response time SHALL be ≤10 seconds for 1M+ transaction searches
- **NFR-017**: API response time SHALL be ≤500ms for 99% of external integration calls
- **NFR-018**: Database query response time SHALL be ≤100ms for 95% of operational queries

2. Scalability Requirements

2.1 Transaction Volume Scaling

- **NFR-019**: System SHALL scale horizontally to handle 1M+ transactions per second peak load
- **NFR-020**: System SHALL support 10,000+ concurrent fraud analysts and risk managers
- **NFR-021**: System SHALL handle 100TB+ of transaction data with automated data lifecycle management
- **NFR-022**: System SHALL scale ML model serving to 1M+ predictions per second
- **NFR-023**: System SHALL support 1000+ concurrent model training jobs for different segments
- **NFR-024**: System SHALL auto-scale compute resources based on transaction volume (50-500% capacity)

2.2 Data Volume and Storage Scaling

- **NFR-025**: System SHALL store 7+ years of transaction history for regulatory compliance
- **NFR-026**: System SHALL handle 1PB+ of historical data with efficient querying capabilities
- **NFR-027**: System SHALL support real-time data ingestion of 10GB+ per minute
- **NFR-028**: System SHALL maintain 99.99% data availability across all storage tiers
- **NFR-029**: System SHALL support automated data archiving and retrieval for compliance
- **NFR-030**: System SHALL provide data compression achieving 70%+ storage reduction

3. Reliability & Availability Requirements

3.1 System Availability

- **NFR-031**: System availability SHALL be 99.99% (max 52.6 minutes downtime/year)
- **NFR-032**: Planned maintenance windows SHALL not exceed 2 hours monthly
- **NFR-033**: Mean Time Between Failures (MTBF) SHALL be ≥8760 hours (1 year)
- **NFR-034**: Mean Time To Recovery (MTTR) SHALL be ≤15 minutes for critical system failures
- **NFR-035**: System SHALL support rolling updates with zero-downtime deployments
- **NFR-036**: System SHALL maintain service during single data center failures

3.2 Data Integrity and Consistency

- **NFR-037**: Transaction data integrity SHALL be 99.999% with automated consistency checks
- **NFR-038**: Data backup SHALL occur every 15 minutes with 99.99% backup success rate
- **NFR-039**: Recovery Point Objective (RPO) SHALL be ≤15 minutes for all critical data
- **NFR-040**: Recovery Time Objective (RTO) SHALL be ≤30 minutes for full system recovery
- **NFR-041**: Cross-region data replication SHALL maintain ≤1 second synchronization lag
- **NFR-042**: Audit trail SHALL be immutable and tamper-evident for regulatory compliance

4. Security Requirements

4.1 Data Protection and Encryption

- **NFR-043**: All sensitive data SHALL be encrypted at rest using AES-256 encryption
- **NFR-044**: All data in transit SHALL be encrypted using TLS 1.3 with perfect forward secrecy
- **NFR-045**: Encryption key management SHALL use FIPS 140-2 Level 3 certified HSMs
- **NFR-046**: PII and payment data SHALL be tokenized with format-preserving encryption
- **NFR-047**: Database encryption SHALL use transparent data encryption (TDE) with key rotation
- **NFR-048**: Backup data SHALL be encrypted with separate key management system

4.2 Access Control and Authentication

- **NFR-049**: System SHALL implement multi-factor authentication (MFA) for all user access
- **NFR-050**: System SHALL support SAML 2.0 and OAuth 2.0 for enterprise SSO integration
- **NFR-051**: Role-based access control (RBAC) SHALL support 50+ granular permissions
- **NFR-052**: Privileged access SHALL require additional authentication and approval workflows
- **NFR-053**: Session management SHALL enforce 30-minute idle timeout and 8-hour maximum session
- **NFR-054**: API authentication SHALL use mutual TLS and JWT tokens with short expiration

4.3 Security Monitoring and Compliance

- **NFR-055**: System SHALL log all security events with SIEM integration capabilities
- **NFR-056**: Intrusion detection SHALL monitor all network traffic and system activities
- **NFR-057**: Vulnerability scanning SHALL be performed weekly with automated remediation
- **NFR-058**: Security incident response SHALL achieve ≤1 hour detection and ≤4 hour containment
- **NFR-059**: Penetration testing SHALL be conducted quarterly by certified security firms
- **NFR-060**: Security compliance SHALL maintain PCI DSS Level 1 certification continuously

5. Regulatory Compliance Requirements

5.1 Financial Services Compliance

- **NFR-061**: System SHALL comply with PCI DSS Level 1 requirements for payment card data
- **NFR-062**: System SHALL meet SOX compliance for financial reporting and internal controls
- **NFR-063**: System SHALL adhere to Basel III requirements for operational risk management
- **NFR-064**: System SHALL comply with Dodd-Frank Act requirements for systemic risk monitoring
- **NFR-065**: System SHALL meet FFIEC guidelines for IT examination and cybersecurity
- **NFR-066**: System SHALL support regulatory stress testing and scenario analysis

5.2 Anti-Money Laundering (AML) and KYC Compliance

- **NFR-067**: System SHALL comply with Bank Secrecy Act (BSA) requirements for suspicious activity reporting
- **NFR-068**: System SHALL meet FinCEN requirements for currency transaction reporting (CTR)
- **NFR-069**: System SHALL support OFAC sanctions screening with real-time updates
- **NFR-070**: System SHALL comply with USA PATRIOT Act requirements for customer identification
- **NFR-071**: System SHALL meet state money transmitter licensing requirements
- **NFR-072**: System SHALL support international AML compliance (FATF, EU AML directives)

5.3 Data Privacy and Protection

- **NFR-073**: System SHALL comply with GDPR requirements for EU customer data protection
- **NFR-074**: System SHALL meet CCPA requirements for California consumer privacy rights
- **NFR-075**: System SHALL support data subject rights (access, rectification, erasure, portability)
- **NFR-076**: System SHALL implement privacy by design principles in all data processing
- **NFR-077**: System SHALL maintain data processing records for regulatory audits
- **NFR-078**: System SHALL support cross-border data transfer compliance (adequacy decisions, SCCs)

6. Performance Monitoring and Observability

6.1 System Monitoring

- **NFR-079**: System SHALL provide real-time performance metrics with ≤1 second granularity
- **NFR-080**: System SHALL implement distributed tracing for end-to-end transaction visibility
- **NFR-081**: System SHALL maintain 99.9% monitoring system availability
- **NFR-082**: System SHALL provide automated alerting with ≤30 second notification time
- **NFR-083**: System SHALL support custom dashboards and visualization for different user roles
- **NFR-084**: System SHALL maintain performance baselines and anomaly detection

6.2 Business Metrics Monitoring

- **NFR-085**: System SHALL track fraud detection accuracy with real-time model performance metrics
- **NFR-086**: System SHALL monitor false positive rates with automated threshold alerting
- **NFR-087**: System SHALL provide business impact metrics (fraud losses prevented, operational costs)
- **NFR-088**: System SHALL track analyst productivity and case resolution metrics
- **NFR-089**: System SHALL monitor customer experience metrics (transaction approval rates, dispute rates)
- **NFR-090**: System SHALL provide regulatory compliance metrics and audit trail completeness

7. Usability and User Experience Requirements

7.1 Fraud Analyst Interface

- **NFR-091**: Fraud analyst training time SHALL be ≤8 hours for basic system proficiency
- **NFR-092**: System SHALL support keyboard shortcuts and power-user features for efficiency
- **NFR-093**: Interface SHALL be accessible according to WCAG 2.1 AA standards
- **NFR-094**: System SHALL provide contextual help and guided workflows for complex tasks
- **NFR-095**: Interface SHALL support multiple monitor configurations and customizable layouts
- **NFR-096**: System SHALL provide mobile-responsive design for on-call analyst access

7.2 Risk Manager and Executive Interface

- **NFR-097**: Executive dashboards SHALL load within ≤3 seconds with real-time data
- **NFR-098**: System SHALL provide drill-down capabilities from summary to detailed views
- **NFR-099**: Reports SHALL be exportable in multiple formats (PDF, Excel, CSV, PowerBI)
- **NFR-100**: System SHALL support scheduled report delivery via email and secure portals
- **NFR-101**: Interface SHALL support multi-language localization for global operations
- **NFR-102**: System SHALL provide role-based customization of dashboards and reports

8. Integration and Interoperability Requirements

8.1 Core Banking System Integration

- **NFR-103**: Integration SHALL support 99.9% message delivery success rate
- **NFR-104**: System SHALL handle integration failures with automatic retry and circuit breaker patterns
- **NFR-105**: Integration SHALL support multiple message formats (ISO 8583, ISO 20022, proprietary)
- **NFR-106**: System SHALL provide integration monitoring with end-to-end transaction tracking
- **NFR-107**: Integration SHALL support rate limiting and throttling to protect downstream systems
- **NFR-108**: System SHALL maintain integration SLAs with core banking systems (≤50ms response time)

8.2 External Service Integration

- **NFR-109**: External API integration SHALL have 99.5% availability with fallback mechanisms
- **NFR-110**: System SHALL support API versioning and backward compatibility for 2+ years
- **NFR-111**: Integration SHALL implement exponential backoff and jitter for retry mechanisms
- **NFR-112**: System SHALL provide webhook delivery with guaranteed delivery and replay capabilities
- **NFR-113**: Integration SHALL support batch and real-time data synchronization modes
- **NFR-114**: System SHALL maintain integration security with mutual authentication and encryption

9. Disaster Recovery and Business Continuity

9.1 Disaster Recovery

- **NFR-115**: System SHALL support automated failover to secondary data center within ≤5 minutes
- **NFR-116**: Disaster recovery testing SHALL be performed quarterly with documented results
- **NFR-117**: System SHALL maintain hot-standby replicas with ≤1 second data lag
- **NFR-118**: Recovery procedures SHALL be automated with minimal manual intervention
- **NFR-119**: System SHALL support geographic distribution across 3+ availability zones

- **NFR-120**: Backup and recovery SHALL support point-in-time recovery for any time within 30 days

9.2 Business Continuity

- **NFR-121**: System SHALL maintain core fraud detection capabilities during partial system failures
- **NFR-122**: System SHALL support degraded mode operation with reduced functionality
- **NFR-123**: Business continuity plan SHALL be tested annually with full stakeholder participation
- **NFR-124**: System SHALL provide emergency procedures for manual fraud detection processes
- **NFR-125**: Communication plan SHALL notify stakeholders within ≤15 minutes of major incidents
- **NFR-126**: System SHALL maintain 30-day operational resilience for extended outages

10. Environmental and Operational Requirements

10.1 Infrastructure Requirements

- **NFR-127**: System SHALL operate in cloud environments with multi-region deployment
- **NFR-128**: System SHALL support containerized deployment with Kubernetes orchestration
- **NFR-129**: System SHALL implement infrastructure as code with automated provisioning
- **NFR-130**: System SHALL support auto-scaling based on transaction volume and system load
- **NFR-131**: System SHALL optimize resource utilization achieving 70%+ average CPU utilization
- **NFR-132**: System SHALL support hybrid cloud deployment with on-premises integration

10.2 Operational Excellence

- **NFR-133**: System SHALL provide automated deployment pipelines with rollback capabilities
- **NFR-134**: System SHALL support blue-green deployments for zero-downtime updates
- **NFR-135**: System SHALL implement chaos engineering practices for resilience testing
- **NFR-136**: System SHALL provide comprehensive logging with structured log formats
- **NFR-137**: System SHALL support automated capacity planning and resource optimization
- **NFR-138**: System SHALL maintain operational runbooks and incident response procedures # Architecture Diagram (AD) ## Banking Fraud Detection Real-Time Analytics System

Building upon PRD, FRD, and NFRD for comprehensive system architecture

ETVX Framework

ENTRY CRITERIA

- ✅ PRD completed with business objectives and success metrics
- ✅ FRD completed with 114 functional requirements (FR-001 to FR-114)
- ✅ NFRD completed with 138 non-functional requirements (NFR-001 to NFR-138)
- ✅ Performance targets defined (<100ms processing, 100K+ TPS, 99.99% availability)
- ✅ Security and compliance requirements documented (PCI DSS, SOX, Basel III)
- ✅ Integration requirements with core banking systems established

TASK

Design comprehensive system architecture that satisfies all functional and non-functional requirements, including real-time transaction processing, ML model serving, rule engines, data storage, security layers, and integration patterns for high-performance fraud detection in banking environments.

VERIFICATION & VALIDATION

Verification Checklist: - [ ] Architecture supports all 114 functional requirements from FRD - [ ] Design meets all 138 non-functional requirements from NFRD - [ ] Performance targets achievable with proposed architecture (<100ms, 100K+ TPS) - [ ] Security architecture meets PCI DSS Level 1 and banking regulations - [ ] Scalability design supports projected transaction volumes and growth - [ ] Integration patterns support core banking and external system requirements

Validation Criteria: - [ ] Architecture review completed with banking system architects - [ ]

Security design validated with cybersecurity and compliance teams - [ ] Performance modeling confirms latency and throughput targets - [ ] Technology stack validated with infrastructure and ML engineering teams - [ ] Cost modeling completed for cloud infrastructure and operational expenses - [ ] Disaster recovery and business continuity capabilities validated

EXIT CRITERIA

- ✅ Complete system architecture documented with component interactions
- ✅ Technology stack defined with specific versions and configurations
- ✅ Data flow diagrams created for all major system processes
- ✅ Security architecture documented with threat model and controls
- ✅ Foundation established for detailed high-level design development

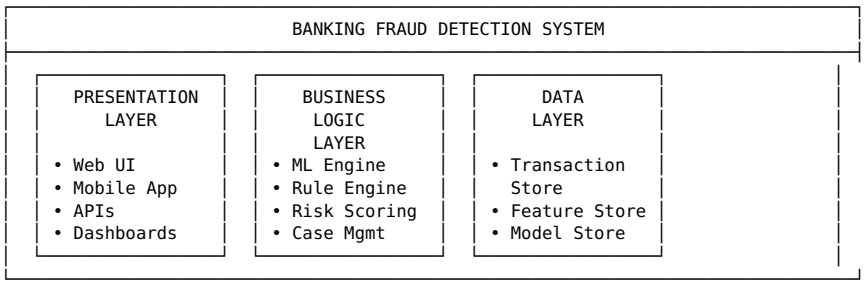
---

Reference to Previous Documents

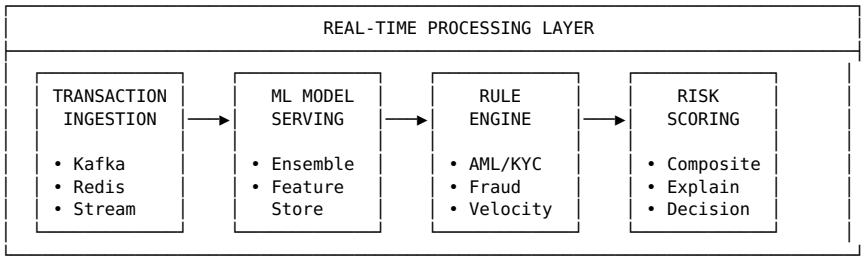
This Architecture Diagram implements requirements from **ALL** previous documents: - **PRD Success Metrics** → Architecture designed for >99% detection accuracy, <100ms processing, 99.99% uptime - **PRD Target Users** → User interface architecture for fraud analysts, risk managers, compliance officers - **PRD Key Features** → Technical architecture for real-time ML, rule engines, explainable AI, adaptive learning - **FRD Real-Time Processing (FR-001-012)** → Stream processing architecture with Kafka, Redis, low-latency design - **FRD ML Detection Engine (FR-013-030)** → ML serving architecture with ensemble models, feature stores, model management - **FRD Rule Engine (FR-031-048)** → Rule processing architecture with dynamic configuration and compliance integration - **FRD Risk Scoring (FR-049-060)** → Explainable AI architecture with SHAP, LIME, and decision transparency - **FRD Alert Management (FR-061-078)** → Case management architecture with workflow engines and analyst interfaces - **FRD Adaptive Learning (FR-079-090)** → MLOps architecture with continuous training, A/B testing, feedback loops - **FRD Integration (FR-091-102)** → API gateway, message queues, and secure integration patterns - **FRD Reporting (FR-103-114)** → Analytics architecture with real-time dashboards and regulatory reporting - **NFRD Performance (NFR-001-018)** → High-performance architecture with caching, load balancing, optimization - **NFRD Scalability (NFR-019-030)** → Auto-scaling architecture with horizontal scaling and data partitioning - **NFRD Reliability (NFR-031-042)** → Fault-tolerant architecture with redundancy, backup, and disaster recovery - **NFRD Security (NFR-043-060)** → Zero-trust security architecture with encryption, authentication, monitoring - **NFRD Compliance (NFR-061-078)** → Compliance-ready architecture with audit trails, data governance, privacy controls

1. High-Level System Architecture

1.1 Overall Architecture Pattern



1.2 Core Components Architecture



2. Technology Stack Architecture

2.1 Infrastructure Layer

- **Cloud Platform**: AWS/Azure/GCP multi-region deployment
- **Container Orchestration**: Kubernetes with Istio service mesh
- **Infrastructure as Code**: Terraform + Ansible
- **Monitoring**: Prometheus + Grafana + Jaeger distributed tracing
- **Security**: HashiCorp Vault for secrets, AWS KMS for encryption

2.2 Data Processing Stack

- **Stream Processing**: Apache Kafka + Kafka Streams + Apache Flink
- **Batch Processing**: Apache Spark + Apache Airflow
- **Message Queue**: Apache Kafka + Redis Streams
- **API Gateway**: Kong/Istio Gateway with rate limiting
- **Load Balancer**: NGINX/HAProxy with health checks

2.3 Machine Learning Stack

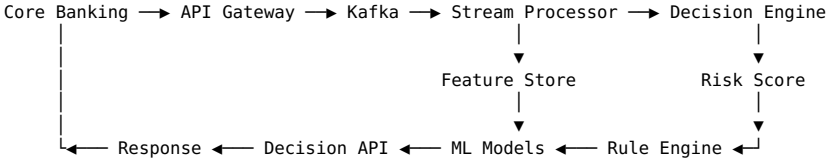
- **ML Framework**: TensorFlow Serving + PyTorch + Scikit-learn
- **Feature Store**: Feast + Redis + Apache Cassandra
- **Model Registry**: MLflow + DVC for version control
- **ML Pipeline**: Kubeflow + Apache Airflow
- **Model Serving**: TensorFlow Serving + Seldon Core

2.4 Database Architecture

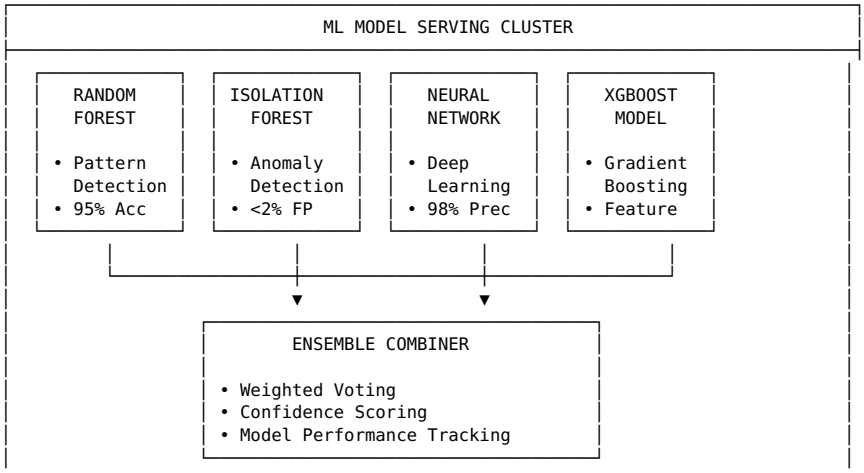
- **OLTP Database**: PostgreSQL with read replicas
- **OLAP Database**: ClickHouse for analytics
- **Time Series**: InfluxDB for metrics
- **Graph Database**: Neo4j for fraud ring detection
- **Cache Layer**: Redis Cluster + Memcached

3. Real-Time Processing Architecture

3.1 Transaction Processing Flow

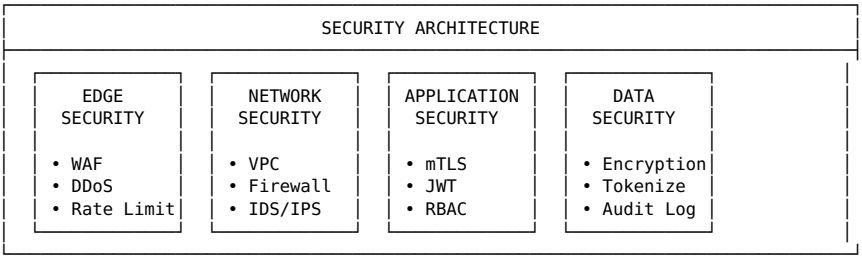


3.2 ML Model Serving Architecture



4. Security Architecture

4.1 Zero-Trust Security Model

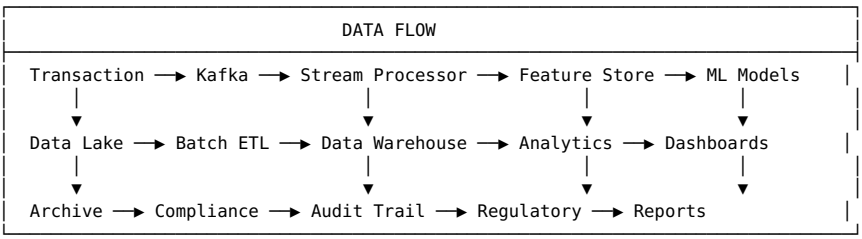


4.2 Data Protection Architecture

- **Encryption at Rest**: AES-256 with HSM key management
- **Encryption in Transit**: TLS 1.3 with certificate pinning
- **Data Tokenization**: Format-preserving encryption for PII
- **Key Management**: HashiCorp Vault with auto-rotation
- **Access Control**: OAuth 2.0 + SAML 2.0 + MFA

5. Data Architecture

5.1 Data Flow Architecture

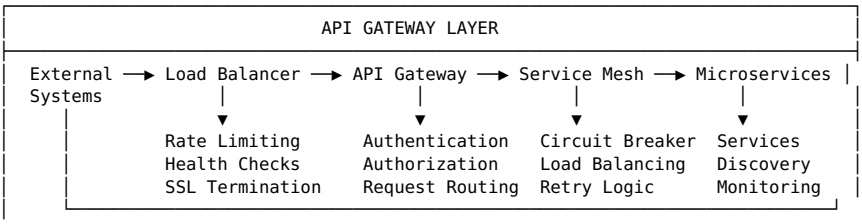


5.2 Storage Architecture

- **Hot Data**: Redis Cluster (real-time features, <1ms access)
- **Warm Data**: PostgreSQL (operational data, <10ms access)
- **Cold Data**: S3/Azure Blob (historical data, <1s access)
- **Archive Data**: Glacier (compliance data, <12h access)

6. Integration Architecture

6.1 API Gateway Pattern



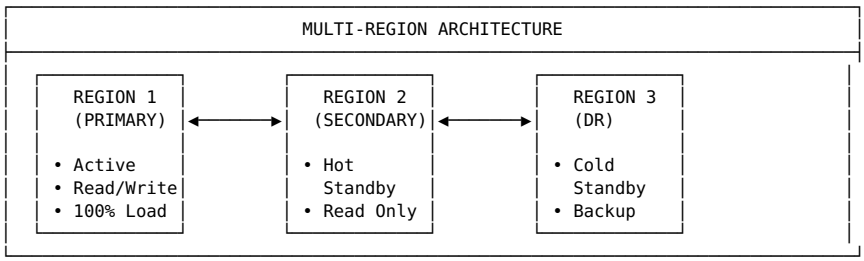
6.2 Message Queue Architecture

- **High Throughput**: Apache Kafka (100K+ TPS)
- **Low Latency**: Redis Streams (<1ms)
- **Reliability**: RabbitMQ with clustering

- **Dead Letter Queue**: Failed message handling
- **Message Ordering**: Partition-based ordering

7. Deployment Architecture

7.1 Multi-Region Deployment



7.2 Container Architecture

- **Orchestration**: Kubernetes with auto-scaling
- **Service Mesh**: Istio for traffic management
- **Container Registry**: Harbor with vulnerability scanning
- **CI/CD Pipeline**: GitLab CI/CD with automated testing
- **Configuration**: Helm charts + ConfigMaps + Secrets # High Level Design (HLD) ## Banking Fraud Detection Real-Time Analytics System

Building upon PRD, FRD, NFRD, and Architecture Diagram for detailed system design

ETVX Framework

ENTRY CRITERIA

- ✅ PRD completed with business objectives and success metrics
- ✅ FRD completed with 114 functional requirements (FR-001 to FR-114)
- ✅ NFRD completed with 138 non-functional requirements (NFR-001 to NFR-138)
- ✅ Architecture Diagram completed with technology stack and component design
- ✅ System architecture validated for performance targets (<100ms, 100K+ TPS)
- ✅ Security architecture approved for PCI DSS Level 1 compliance

TASK

Create detailed high-level design specifications for each system component, defining interfaces, data models, processing workflows, integration patterns, and operational procedures that implement the architecture while satisfying all functional and non-functional requirements.

VERIFICATION & VALIDATION

Verification Checklist: - [ ] All architectural components have detailed design specifications - [ ] Interface definitions support all functional requirements (FR-001 to FR-114) - [ ] Data models satisfy performance and scalability requirements (NFR-001 to NFR-138) - [ ] Processing workflows meet latency targets (<100ms end-to-end) - [ ] Integration patterns support core banking and external system requirements - [ ] Security controls implement zero-trust architecture from AD

Validation Criteria: - [ ] Design review completed with ML engineering and banking system teams - [ ] Performance modeling validates latency and throughput targets - [ ] Security design review confirms PCI DSS and regulatory compliance - [ ] Integration patterns validated with core banking system architects - [ ] Operational procedures reviewed with DevOps and SRE teams - [ ] Design traceability confirmed to all previous requirements documents

EXIT CRITERIA

- ✅ Detailed component specifications completed for all system modules
- ✅ Interface definitions documented with API specifications and data contracts
- ✅ Processing workflows designed with sequence diagrams and state machines
- ✅ Data models defined with schemas, relationships, and access patterns

- ✅ Foundation established for low-level design and implementation specifications

---

Reference to Previous Documents

This HLD implements detailed design based on **ALL** previous documents: - **PRD Success Metrics** → Component design for >99% detection accuracy, <100ms processing, 99.99% uptime - **PRD Target Users** → Interface design for fraud analysts, risk managers, compliance officers - **FRD Real-Time Processing (FR-001-012)** → Stream processing component design with Kafka, Redis, decision engine - **FRD ML Detection Engine (FR-013-030)** → ML serving component design with ensemble models, feature engineering - **FRD Rule Engine (FR-031-048)** → Rule processing component design with AML/KYC, fraud patterns, dynamic configuration - **FRD Risk Scoring (FR-049-060)** → Explainable AI component design with SHAP, LIME, decision transparency - **FRD Alert Management (FR-061-078)** → Case management component design with workflow engines, analyst interfaces - **FRD Adaptive Learning (FR-079-090)** → MLOps component design with continuous training, A/B testing, feedback loops - **FRD Integration (FR-091-102)** → API gateway and integration component design with secure messaging - **FRD Reporting (FR-103-114)** → Analytics component design with real-time dashboards, regulatory reporting - **NFRD Performance (NFR-001-018)** → High-performance component design with caching, optimization, monitoring - **NFRD Security (NFR-043-060)** → Security component design with encryption, authentication, access control - **Architecture Diagram** → Technology stack implementation with Kubernetes, Kafka, PostgreSQL, Redis, ML frameworks

1. Real-Time Transaction Processing Component

1.1 Transaction Ingestion Service

Purpose: Ingest and validate real-time transactions from multiple banking channels **Technology**: Spring Boot + Apache Kafka + Redis

```
Component: TransactionIngestionService
Interfaces:
  - REST API: /api/v1/transactions (POST)
  - Kafka Producer: fraud-detection-transactions topic
  - Redis Cache: transaction-cache cluster

Data Model:
  Transaction:
    - transactionId: UUID (primary key)
    - customerId: String (indexed)
    - amount: Decimal (precision 2)
    - currency: String (ISO 4217)
    - merchantId: String (indexed)
    - timestamp: DateTime (UTC)
    - channel: Enum (CARD, ACH, WIRE, MOBILE)
    - location: GeoPoint (lat, lng)
    - deviceFingerprint: String (hashed)

Processing Flow:
  1. Validate transaction format (ISO 8583)
  2. Enrich with customer profile data
  3. Cache in Redis for fast access
  4. Publish to Kafka topic
  5. Return acknowledgment (<10ms)
```

1.2 Stream Processing Engine

Purpose: Process transaction streams with ML models and rules in real-time **Technology**: Apache Flink + Kafka Streams + TensorFlow Serving

```
Component: StreamProcessingEngine
Interfaces:
  - Kafka Consumer: fraud-detection-transactions
  - Kafka Producer: fraud-decisions topic
  - Feature Store API: /features/customer/{id}
  - ML Model API: /predict/ensemble

Processing Topology:
  1. Transaction Stream → Feature Enrichment
  2. Feature Enrichment → ML Model Serving
  3. ML Model Serving → Rule Engine
  4. Rule Engine → Risk Scoring
```

```
  5. Risk Scoring → Decision Engine

Performance Targets:
  - Processing Latency: <100ms (P99.9)
  - Throughput: 100,000+ TPS
  - Availability: 99.99%
```

2. Machine Learning Detection Component

2.1 Feature Store Service

Purpose: Serve real-time and batch features for ML model inference **Technology**: Feast + Redis Cluster + Apache Cassandra

```
Component: FeatureStoreService
Interfaces:
  - REST API: /api/v1/features/{entity_id}
  - gRPC API: FeatureService.GetFeatures()
  - Streaming API: Kafka feature-updates topic

Feature Categories:
  Customer Features:
    - avg_transaction_amount_7d: Float
    - transaction_frequency_24h: Integer
    - unique_merchants_30d: Integer
    - geographic_diversity_score: Float
    - account_age_days: Integer

  Transaction Features:
    - amount_zscore_customer: Float
    - time_since_last_transaction: Integer
    - merchant_risk_score: Float
    - location_risk_score: Float
    - device_trust_score: Float

  Behavioral Features:
    - spending_pattern_deviation: Float
    - velocity_score_1h: Float
    - impossible_travel_flag: Boolean
    - account_takeover_score: Float

Storage Strategy:
  - Hot Features: Redis (sub-millisecond access)
  - Warm Features: Cassandra (single-digit millisecond)
  - Cold Features: S3 (batch processing)
```

2.2 ML Model Serving Service

Purpose: Serve ensemble ML models for fraud detection with high performance **Technology**: TensorFlow Serving + Seldon Core + Kubernetes

```
Component: MLModelServingService
Interfaces:
  - REST API: /api/v1/predict
  - gRPC API: PredictionService.Predict()
  - Model Management: /api/v1/models/{model_id}

Model Ensemble:
  RandomForestModel:
    - Framework: Scikit-learn
    - Features: 150+ engineered features
    - Target Accuracy: >95%
    - Inference Time: <20ms

  IsolationForestModel:
    - Framework: Scikit-learn
    - Purpose: Anomaly detection
    - Target FPR: <2%
    - Inference Time: <15ms

  NeuralNetworkModel:
    - Framework: TensorFlow
    - Architecture: Deep feedforward
    - Target Precision: >98%
    - Inference Time: <25ms
```

```
  XGBoostModel:
    - Framework: XGBoost
    - Purpose: Gradient boosting
    - Feature Importance: SHAP values
    - Inference Time: <10ms

Ensemble Strategy:
  - Weighted voting based on model confidence
  - Dynamic weight adjustment based on performance
  - Fallback to rule-based decisions if models fail
```

3. Rule-Based Engine Component

3.1 Regulatory Compliance Engine

Purpose: Implement AML/KYC and regulatory compliance rules **Technology**: Drools Rule Engine + Spring Boot + PostgreSQL

```
Component: RegulatoryComplianceEngine
Interfaces:
  - REST API: /api/v1/compliance/evaluate
  - Rule Management: /api/v1/rules/compliance
  - OFAC Screening: /api/v1/sanctions/check

Rule Categories:
  AML_Rules:
    - Suspicious Activity Detection
    - Currency Transaction Reporting (CTR)
    - Structuring Pattern Detection
    - Cross-border Transaction Monitoring

  KYC_Rules:
    - Customer Due Diligence (CDD)
    - Enhanced Due Diligence (EDD)
    - Politically Exposed Person (PEP) Screening
    - Beneficial Ownership Identification

  Sanctions_Rules:
    - OFAC SDN List Screening
    - EU Sanctions List Screening
    - UN Sanctions List Screening
    - Country-based Restrictions

Processing Flow:
  1. Load transaction and customer data
  2. Execute compliance rule set
  3. Generate compliance score (0-100)
  4. Flag for regulatory reporting if needed
  5. Return compliance decision (<20ms)
```

3.2 Fraud Pattern Engine

Purpose: Detect known fraud patterns using business rules **Technology**: Apache Kafka Streams + Redis + Custom Rule Engine

```
Component: FraudPatternEngine
Interfaces:
  - Stream Processing: Kafka fraud-patterns topic
  - Rule Configuration: /api/v1/rules/fraud
  - Pattern Detection: /api/v1/patterns/detect

Pattern Categories:
  Velocity_Patterns:
    - Transaction frequency limits
    - Amount velocity thresholds
    - Geographic velocity detection
    - Time-based pattern analysis

  Behavioral_Patterns:
    - Account takeover indicators
    - Card testing patterns
    - Merchant fraud indicators
    - Device fingerprint anomalies
```

```
  Network_Patterns:
    - Fraud ring detection
    - Connected account analysis
    - Shared device patterns
    - IP address clustering

Rule Engine Features:
  - Dynamic rule deployment
  - A/B testing framework
  - Performance monitoring
  - Conflict resolution
```

4. Risk Scoring and Explainability Component

4.1 Composite Risk Scoring Service

Purpose: Calculate final risk scores combining ML and rule outputs **Technology**: Spring Boot + Redis + Apache Kafka

```
Component: CompositeRiskScoringService
Interfaces:
  - REST API: /api/v1/risk/score
  - Streaming: Kafka risk-scores topic
  - Explanation API: /api/v1/risk/explain

Scoring Algorithm:
  composite_score = (
    ml_ensemble_score * 0.6 +
    rule_engine_score * 0.3 +
    behavioral_score * 0.1
  )

Risk Bands:
  - LOW: 0-300 (Auto-approve)
  - MEDIUM: 301-600 (Additional verification)
  - HIGH: 601-800 (Manual review)
  - CRITICAL: 801-1000 (Auto-decline)

Explainability Features:
  - SHAP value calculation
  - Feature importance ranking
  - Counterfactual explanations
  - Natural language explanations
```

4.2 Explainable AI Service

Purpose: Provide human-readable explanations for fraud decisions **Technology**: SHAP + LIME + Custom NLG + FastAPI

```
Component: ExplainableAIService
Interfaces:
  - REST API: /api/v1/explain/{transaction_id}
  - Batch API: /api/v1/explain/batch
  - Visualization: /api/v1/explain/visual

Explanation Types:
  Feature_Importance:
    - Top 10 contributing factors
    - Positive and negative influences
    - Confidence intervals
    - Historical comparisons

  Counterfactual:
    - "What if" scenarios
    - Minimum changes for different outcome
    - Sensitivity analysis
    - Threshold explanations

  Natural_Language:
    - Plain English explanations
    - Regulatory-compliant language
    - Customer-facing explanations
    - Technical explanations for analysts
```

5. Alert Management and Case Workflow Component

5.1 Intelligent Alert Service

Purpose: Generate and manage fraud alerts with intelligent prioritization **Technology**: Spring Boot + PostgreSQL + Redis + Apache Kafka

```
Component: IntelligentAlertService
Interfaces:
  - REST API: /api/v1/alerts
  - WebSocket: /ws/alerts/realtime
  - Notification: /api/v1/alerts/notify

Alert Processing:
  1. Receive high-risk transactions
  2. Apply alert suppression rules
  3. Correlate with existing cases
  4. Calculate priority score
  5. Route to appropriate analyst queue
  6. Send notifications (email, SMS, Slack)

Alert Prioritization:
  priority_score = (
    risk_score * 0.4 +
    customer_value * 0.2 +
    potential_loss * 0.2 +
    time_sensitivity * 0.2
  )

Alert States:
  - NEW: Just created
  - ASSIGNED: Assigned to analyst
  - IN_PROGRESS: Under investigation
  - RESOLVED: Investigation complete
  - CLOSED: Case closed
```

5.2 Case Management Service

Purpose: Manage fraud investigation cases and analyst workflows **Technology**: Spring Boot + PostgreSQL + Elasticsearch + React

```
Component: CaseManagementService
Interfaces:
  - REST API: /api/v1/cases
  - Search API: /api/v1/cases/search
  - Workflow API: /api/v1/cases/workflow

Case Data Model:
  Case:
    - caseId: UUID
    - transactionIds: List<UUID>
    - customerId: String
    - assignedAnalyst: String
    - priority: Enum (LOW, MEDIUM, HIGH, CRITICAL)
    - status: Enum (OPEN, IN_PROGRESS, RESOLVED, CLOSED)
    - createdAt: DateTime
    - updatedAt: DateTime
    - resolution: String
    - tags: List<String>

Workflow Engine:
  - State machine for case progression
  - Automated task assignment
  - SLA monitoring and escalation
  - Collaboration features
  - Audit trail maintenance
```

6. Integration and API Component

6.1 API Gateway Service

Purpose: Provide secure, scalable API access with rate limiting and monitoring **Technology**: Kong Gateway + Redis + Prometheus

```
Component: APIGatewayService
Interfaces:
  - External APIs: /api/v1/*
  - Admin API: /admin/api/v1/*
  - Health Check: /health

Security Features:
  - OAuth 2.0 / JWT authentication
  - Rate limiting (per client/IP)
  - Request/response validation
  - API key management
  - IP whitelisting

Monitoring Features:
  - Request/response logging
  - Performance metrics
  - Error rate tracking
  - SLA monitoring
  - Circuit breaker patterns
```

6.2 Core Banking Integration Service

Purpose: Integrate with core banking systems for transaction processing **Technology**: Apache Camel + IBM MQ + Spring Boot

```
Component: CoreBankingIntegrationService
Interfaces:
  - ISO 8583 Message Processing
  - REST API: /api/v1/banking/*
  - Message Queue: IBM MQ / RabbitMQ

Integration Patterns:
  - Request-Reply for authorization
  - Publish-Subscribe for notifications
  - Message transformation (ISO 8583 ↔ JSON)
  - Error handling and retry logic
  - Circuit breaker for resilience

Message Types:
  - Authorization Request/Response
  - Transaction Notification
  - Account Status Update
  - Fraud Decision Notification
  - Regulatory Report Submission
```

7. Data Management Component

7.1 Data Pipeline Service

Purpose: Process and transform data for ML training and analytics **Technology**: Apache Airflow + Apache Spark + Delta Lake

```
Component: DataPipelineService
Interfaces:
  - Batch Processing: Airflow DAGs
  - Stream Processing: Kafka Connect
  - Data Quality: Great Expectations

Pipeline Categories:
  Feature_Engineering:
    - Customer behavior aggregation
    - Transaction pattern analysis
    - Risk score calculation
    - Merchant profiling

  Model_Training:
    - Data preparation and cleaning
    - Feature selection and engineering
    - Model training and validation
    - Model deployment and monitoring

  Analytics:
    - Fraud trend analysis
    - Performance reporting
    - Regulatory compliance reporting
```

```
    - Business intelligence dashboards
```

7.2 Data Storage Service

Purpose: Manage data storage across hot, warm, and cold tiers **Technology**: PostgreSQL + Redis + S3 + Snowflake

```
Component: DataStorageService
Storage Tiers:
  Hot_Storage:
    - Technology: Redis Cluster
    - Data: Real-time features, cache
    - Access Pattern: <1ms latency
    - Retention: 24 hours

  Warm_Storage:
    - Technology: PostgreSQL
    - Data: Operational data, cases
    - Access Pattern: <10ms latency
    - Retention: 90 days

  Cold_Storage:
    - Technology: S3 / Azure Blob
    - Data: Historical transactions
    - Access Pattern: <1s latency
    - Retention: 7 years

  Archive_Storage:
    - Technology: Glacier / Archive
    - Data: Compliance data
    - Access Pattern: <12h latency
    - Retention: Indefinite
```

8. Monitoring and Observability Component

8.1 Application Performance Monitoring

Purpose: Monitor system performance and detect issues proactively **Technology**: Prometheus + Grafana + Jaeger + ELK Stack

```
Component: APMService
Metrics Collection:
  - Application metrics (latency, throughput, errors)
  - Infrastructure metrics (CPU, memory, disk, network)
  - Business metrics (fraud detection rate, false positives)
  - ML model metrics (accuracy, drift, performance)

Alerting Rules:
  - Transaction processing latency > 100ms
  - ML model accuracy < 95%
  - False positive rate > 0.1%
  - System availability < 99.99%
  - Queue depth > 1000 messages

Dashboards:
  - Real-time system overview
  - ML model performance
  - Business KPIs
  - Infrastructure health
  - Security monitoring
```

This HLD provides comprehensive design specifications for implementing the banking fraud detection system while maintaining traceability to all previous requirements documents and ensuring implementation readiness for the development team. # Low Level Design (LLD) ## Banking Fraud Detection Real-Time Analytics System

Building upon PRD, FRD, NFRD, Architecture Diagram, and HLD for implementation-ready specifications

ETVX Framework

ENTRY CRITERIA

- ✅ PRD completed with business objectives and success metrics
- ✅ FRD completed with 114 functional requirements (FR-001 to FR-114)
- ✅ NFRD completed with 138 non-functional requirements (NFR-001 to NFR-138)
- ✅ Architecture Diagram completed with technology stack and system architecture
- ✅ HLD completed with detailed component specifications and interfaces
- ✅ Technology stack validated and approved for banking environment

TASK

Create implementation-ready low-level design specifications including detailed class diagrams, database schemas, API specifications, algorithm implementations, configuration parameters, and deployment scripts that enable direct development of the fraud detection system.

VERIFICATION & VALIDATION

Verification Checklist: - [ ] All classes and methods have detailed specifications with parameters and return types - [ ] Database schemas support all data requirements from HLD components - [ ] API specifications include request/response formats, error codes, and authentication - [ ] Algorithm implementations satisfy performance requirements (<100ms processing) - [ ] Configuration parameters support all operational requirements - [ ] Code structure follows banking industry security and quality standards

Validation Criteria: - [ ] Implementation specifications reviewed with development team for feasibility - [ ] Database design validated with DBA team for performance and scalability - [ ] API specifications validated with integration team and external partners - [ ] Security implementations reviewed with cybersecurity team for compliance - [ ] Performance specifications validated with load testing requirements - [ ] Code quality standards confirmed with architecture review board

EXIT CRITERIA

- ✅ Complete implementation specifications ready for development team
- ✅ Database schemas, API specs, and class diagrams documented
- ✅ Algorithm implementations with performance optimizations specified
- ✅ Configuration management and deployment procedures defined
- ✅ Foundation established for pseudocode and implementation phase

---

Reference to Previous Documents

This LLD provides implementation-ready specifications based on **ALL** previous documents: - **PRD Success Metrics** → Implementation targets for >99% detection accuracy, <100ms processing, 99.99% uptime - **FRD Functional Requirements (FR-001-114)** → Detailed method implementations for all system functions - **NFRD Performance Requirements (NFR-001-138)** → Optimized algorithms and data structures for performance targets - **Architecture Diagram** → Technology stack implementation with specific versions and configurations - **HLD Component Design** → Detailed class structures, database schemas, and API implementations

1. Transaction Ingestion Service Implementation

1.1 Class Structure

```
@RestController
@RequestMapping("/api/v1/transactions")
public class TransactionIngestionController {

    @Autowired
    private TransactionValidationService validationService;

    @Autowired
    private TransactionEnrichmentService enrichmentService;

    @Autowired
    private KafkaTransactionProducer kafkaProducer;

    @Autowired
    private RedisTransactionCache redisCache;

    @PostMapping
    @ResponseTime(maxMillis = 10)
    public ResponseEntity<TransactionResponse> ingestTransaction(
```

```
            @Valid @RequestBody TransactionRequest request) {

        // Validate transaction format (ISO 8583)
        ValidationResult validation = validationService.validate(request);
        if (!validation.isValid()) {
            return ResponseEntity.badRequest()
                .body(new TransactionResponse(validation.getErrors()));
        }

        // Enrich with customer profile data
        EnrichedTransaction enriched = enrichmentService.enrich(request);

        // Cache in Redis for fast access
        redisCache.store(enriched.getTransactionId(), enriched);

        // Publish to Kafka topic
        kafkaProducer.send("fraud-detection-transactions", enriched);

        return ResponseEntity.ok(new TransactionResponse(
            enriched.getTransactionId(),
            "ACCEPTED",
            System.currentTimeMillis()
        ));
    }
}
```

1.2 Data Models

```
@Entity
@Table(name = "transactions", indexes = {
    @Index(name = "idx_customer_id", columnList = "customerId"),
    @Index(name = "idx_timestamp", columnList = "timestamp"),
    @Index(name = "idx_merchant_id", columnList = "merchantId")
})
public class Transaction {

    @Id
    @Column(columnDefinition = "UUID")
    private UUID transactionId;

    @Column(nullable = false, length = 50)
    private String customerId;

    @Column(nullable = false, precision = 15, scale = 2)
    private BigDecimal amount;

    @Column(nullable = false, length = 3)
    private String currency;

    @Column(nullable = false, length = 50)
    private String merchantId;

    @Column(nullable = false)
    private LocalDateTime timestamp;

    @Enumerated(EnumType.STRING)
    private TransactionChannel channel;

    @Embedded
    private GeoLocation location;

    @Column(length = 256)
    private String deviceFingerprint;

    // Constructors, getters, setters
}

@Embeddable
public class GeoLocation {
    @Column(precision = 10, scale = 8)
    private Double latitude;

    @Column(precision = 11, scale = 8)
    private Double longitude;

    @Column(length = 100)
    private String country;
```

```
    @Column(length = 100)
    private String city;
}
```

1.3 Kafka Configuration

```yaml
# application.yml
spring:
  kafka:
    producer:
      bootstrap-servers: ${KAFKA_BROKERS:localhost:9092}
      key-serializer: org.apache.kafka.common.serialization.StringSerializer
      value-serializer: org.springframework.kafka.support.serializer.JsonSerializer
      properties:
        acks: all
        retries: 3
        batch.size: 16384
        linger.ms: 5
        buffer.memory: 33554432
        compression.type: snappy
        max.in.flight.requests.per.connection: 1
        enable.idempotence: true
```

2. ML Model Serving Implementation

2.1 Feature Store Service

```python
from feast import FeatureStore
from redis import Redis
import asyncio
from typing import Dict, List, Optional

class FeatureStoreService:
    def __init__(self):
        self.feast_store = FeatureStore(repo_path=".")
        self.redis_client = Redis(
            host=os.getenv('REDIS_HOST', 'localhost'),
            port=int(os.getenv('REDIS_PORT', 6379)),
            decode_responses=True,
            socket_connect_timeout=1,
            socket_timeout=1
        )

    async def get_features(self, entity_id: str, feature_names: List[str]) -> Dict:
        """Get real-time features with <5ms latency"""
        try:
            # Try Redis cache first
            cached_features = await self._get_cached_features(entity_id, feature_names)
            if cached_features:
                return cached_features

            # Fallback to Feast feature store
            features = self.feast_store.get_online_features(
                features=feature_names,
                entity_rows=[{"customer_id": entity_id}]
            ).to_dict()

            # Cache for future requests
            await self._cache_features(entity_id, features)

            return features

        except Exception as e:
            logger.error(f"Feature retrieval failed for {entity_id}: {e}")
            return self._get_default_features(feature_names)

    async def _get_cached_features(self, entity_id: str, feature_names: List[str]) -> Optional[Dict]:
        """Retrieve features from Redis cache"""
        pipeline = self.redis_client.pipeline()
        for feature_name in feature_names:
            pipeline.hget(f"features:{entity_id}", feature_name)

        results = pipeline.execute()

        if all(result is not None for result in results):
```

```
            return {
                feature_names[i]: float(results[i]) if results[i] else 0.0
                for i in range(len(feature_names))
            }
        return None

    async def _cache_features(self, entity_id: str, features: Dict):
        """Cache features in Redis with TTL"""
        pipeline = self.redis_client.pipeline()
        for feature_name, value in features.items():
            pipeline.hset(f"features:{entity_id}", feature_name, str(value))
        pipeline.expire(f"features:{entity_id}", 300)  # 5 minute TTL
        pipeline.execute()
```

2.2 ML Model Ensemble Implementation

```
import numpy as np
from sklearn.ensemble import RandomForestClassifier, IsolationForest
from xgboost import XGBClassifier
import tensorflow as tf
from typing import Tuple, Dict, List
import asyncio

class MLModelEnsemble:
    def __init__(self):
        self.models = {
            'random_forest': self._load_random_forest(),
            'isolation_forest': self._load_isolation_forest(),
            'neural_network': self._load_neural_network(),
            'xgboost': self._load_xgboost()
        }
        self.model_weights = {
            'random_forest': 0.3,
            'isolation_forest': 0.2,
            'neural_network': 0.3,
            'xgboost': 0.2
        }

    async def predict(self, features: np.ndarray) -> Tuple[float, Dict]:
        """Ensemble prediction with <50ms latency"""
        start_time = time.time()

        # Run all models in parallel
        tasks = [
            self._predict_random_forest(features),
            self._predict_isolation_forest(features),
            self._predict_neural_network(features),
            self._predict_xgboost(features)
        ]

        predictions = await asyncio.gather(*tasks)

        # Calculate weighted ensemble score
        ensemble_score = sum(
            pred * self.model_weights[model_name]
            for pred, model_name in zip(predictions, self.models.keys())
        )

        # Calculate confidence and feature importance
        confidence = self._calculate_confidence(predictions)
        feature_importance = self._calculate_feature_importance(features)

        processing_time = (time.time() - start_time) * 1000

        return ensemble_score, {
            'individual_predictions': dict(zip(self.models.keys(), predictions)),
            'confidence': confidence,
            'feature_importance': feature_importance,
            'processing_time_ms': processing_time
        }

    async def _predict_random_forest(self, features: np.ndarray) -> float:
        """Random Forest prediction with pattern detection"""
        prediction = self.models['random_forest'].predict_proba(features.reshape(1, -1))[0][1]
        return float(prediction)

    async def _predict_isolation_forest(self, features: np.ndarray) -> float:
```

```python
        """Isolation Forest prediction for anomaly detection"""
        anomaly_score = self.models['isolation_forest'].decision_function(features.reshape(1, -1))[0]
        # Convert to probability (higher score = more normal)
        probability = 1 / (1 + np.exp(-anomaly_score))
        return float(1 - probability)  # Return fraud probability

    async def _predict_neural_network(self, features: np.ndarray) -> float:
        """Neural Network prediction for complex patterns"""
        prediction = self.models['neural_network'].predict(features.reshape(1, -1))[0][0]
        return float(prediction)

    async def _predict_xgboost(self, features: np.ndarray) -> float:
        """XGBoost prediction with feature importance"""
        prediction = self.models['xgboost'].predict_proba(features.reshape(1, -1))[0][1]
        return float(prediction)
```

3. Rule Engine Implementation

3.1 Regulatory Compliance Engine

```java
@Service
public class RegulatoryComplianceEngine {

    @Autowired
    private DroolsRuleEngine droolsEngine;

    @Autowired
    private OFACScreeningService ofacService;

    @Autowired
    private SARReportingService sarService;

    @Cacheable(value = "compliance-rules", key = "#transaction.customerId")
    public ComplianceResult evaluateCompliance(Transaction transaction) {

        ComplianceContext context = new ComplianceContext(transaction);

        // Execute AML/KYC rules
        RuleExecutionResult amlResult = droolsEngine.execute("AML_RULES", context);

        // Execute sanctions screening
        SanctionsResult sanctionsResult = ofacService.screenTransaction(transaction);

        // Execute currency transaction reporting
        CTRResult ctrResult = evaluateCTRRequirements(transaction);

        // Calculate composite compliance score
        int complianceScore = calculateComplianceScore(amlResult, sanctionsResult, ctrResult);

        // Generate SAR if required
        if (complianceScore > 80) {
            sarService.generateSAR(transaction, context);
        }

        return new ComplianceResult(
            complianceScore,
            amlResult.getFlags(),
            sanctionsResult.getMatches(),
            ctrResult.isRequired()
        );
    }

    private int calculateComplianceScore(RuleExecutionResult aml,
                                         SanctionsResult sanctions,
                                         CTRResult ctr) {
        int score = 0;

        // AML risk factors (0-40 points)
        score += aml.getRiskFactors().size() * 10;

        // Sanctions matches (0-50 points)
        if (sanctions.hasExactMatch()) score += 50;
        else if (sanctions.hasFuzzyMatch()) score += 30;

        // CTR requirements (0-10 points)
        if (ctr.isRequired()) score += 10;
```

```
        return Math.min(score, 100);
    }
}
```

3.2 Dynamic Rule Configuration

```yaml
# Drools rule configuration
rules:
  aml:
    - name: "Large Cash Transaction"
      condition: "transaction.amount > 10000 && transaction.paymentMethod == 'CASH'"
      action: "addRiskFactor('LARGE_CASH_TRANSACTION', 30)"

    - name: "Rapid Fire Transactions"
      condition: "customerProfile.transactionCount24h > 50"
      action: "addRiskFactor('RAPID_FIRE_TRANSACTIONS', 25)"

    - name: "Geographic Anomaly"
      condition: "distance(transaction.location, customerProfile.usualLocation) > 1000"
      action: "addRiskFactor('GEOGRAPHIC_ANOMALY', 20)"

  fraud:
    - name: "Card Testing Pattern"
      condition: "transaction.amount < 5 && customerProfile.declinedTransactions1h > 10"
      action: "addRiskFactor('CARD_TESTING', 40)"

    - name: "Impossible Travel"
      condition: "timeBetweenLocations(previousTransaction, currentTransaction) < physicalTravelTime"
      action: "addRiskFactor('IMPOSSIBLE_TRAVEL', 50)"
```

4. Database Schema Implementation

4.1 PostgreSQL Schema

```sql
-- Transactions table with partitioning by date
CREATE TABLE transactions (
    transaction_id UUID PRIMARY KEY,
    customer_id VARCHAR(50) NOT NULL,
    amount DECIMAL(15,2) NOT NULL,
    currency CHAR(3) NOT NULL,
    merchant_id VARCHAR(50) NOT NULL,
    timestamp TIMESTAMP WITH TIME ZONE NOT NULL,
    channel VARCHAR(20) NOT NULL,
    latitude DECIMAL(10,8),
    longitude DECIMAL(11,8),
    country VARCHAR(100),
    city VARCHAR(100),
    device_fingerprint VARCHAR(256),
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
) PARTITION BY RANGE (timestamp);

-- Create monthly partitions
CREATE TABLE transactions_2024_01 PARTITION OF transactions
    FOR VALUES FROM ('2024-01-01') TO ('2024-02-01');

-- Indexes for performance
CREATE INDEX CONCURRENTLY idx_transactions_customer_id
    ON transactions (customer_id, timestamp DESC);
CREATE INDEX CONCURRENTLY idx_transactions_merchant_id
    ON transactions (merchant_id, timestamp DESC);
CREATE INDEX CONCURRENTLY idx_transactions_amount
    ON transactions (amount) WHERE amount > 1000;

-- Risk scores table
CREATE TABLE risk_scores (
    transaction_id UUID PRIMARY KEY REFERENCES transactions(transaction_id),
    ml_ensemble_score DECIMAL(5,3) NOT NULL,
    rule_engine_score DECIMAL(5,3) NOT NULL,
    composite_score DECIMAL(5,3) NOT NULL,
    risk_band VARCHAR(20) NOT NULL,
    model_version VARCHAR(20) NOT NULL,
    explanation JSONB,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);
```

```sql
-- Cases table for fraud investigation
CREATE TABLE fraud_cases (
    case_id UUID PRIMARY KEY,
    transaction_ids UUID[] NOT NULL,
    customer_id VARCHAR(50) NOT NULL,
    assigned_analyst VARCHAR(100),
    priority VARCHAR(20) NOT NULL DEFAULT 'MEDIUM',
    status VARCHAR(20) NOT NULL DEFAULT 'OPEN',
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    resolved_at TIMESTAMP WITH TIME ZONE,
    resolution TEXT,
    tags TEXT[]
);

-- Audit trail table
CREATE TABLE audit_trail (
    audit_id UUID PRIMARY KEY,
    entity_type VARCHAR(50) NOT NULL,
    entity_id UUID NOT NULL,
    action VARCHAR(50) NOT NULL,
    old_values JSONB,
    new_values JSONB,
    user_id VARCHAR(100) NOT NULL,
    timestamp TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);
```

4.2 Redis Data Structures

```python
# Redis key patterns and data structures
REDIS_PATTERNS = {
    # Feature cache (Hash)
    'features:{customer_id}': {
        'avg_transaction_amount_7d': 'float',
        'transaction_frequency_24h': 'int',
        'unique_merchants_30d': 'int',
        'geographic_diversity_score': 'float',
        'ttl': 300  # 5 minutes
    },

    # Transaction cache (Hash)
    'transaction:{transaction_id}': {
        'customer_id': 'string',
        'amount': 'float',
        'timestamp': 'int',
        'risk_score': 'float',
        'ttl': 3600  # 1 hour
    },

    # Rate limiting (String with expiry)
    'rate_limit:{customer_id}:{window}': {
        'type': 'counter',
        'ttl': 'window_size'
    },

    # Model cache (Hash)
    'model:{model_name}:{version}': {
        'model_data': 'binary',
        'metadata': 'json',
        'ttl': 86400  # 24 hours
    }
}
```

5. API Specifications

5.1 REST API Endpoints

```yaml
openapi: 3.0.0
info:
  title: Banking Fraud Detection API
  version: 1.0.0
  description: Real-time fraud detection and case management API

paths:
```

```yaml
  /api/v1/transactions:
    post:
      summary: Submit transaction for fraud detection
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/TransactionRequest'
      responses:
        '200':
          description: Transaction processed successfully
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/TransactionResponse'
        '400':
          description: Invalid transaction data
        '429':
          description: Rate limit exceeded
        '500':
          description: Internal server error

  /api/v1/risk/score/{transactionId}:
    get:
      summary: Get risk score and explanation
      parameters:
        - name: transactionId
          in: path
          required: true
          schema:
            type: string
            format: uuid
      responses:
        '200':
          description: Risk score retrieved successfully
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/RiskScoreResponse'

components:
  schemas:
    TransactionRequest:
      type: object
      required:
        - customerId
        - amount
        - currency
        - merchantId
        - timestamp
        - channel
      properties:
        customerId:
          type: string
          maxLength: 50
        amount:
          type: number
          format: decimal
          minimum: 0.01
        currency:
          type: string
          pattern: '^[A-Z]{3}$'
        merchantId:
          type: string
          maxLength: 50
        timestamp:
          type: string
          format: date-time
        channel:
          type: string
          enum: [CARD, ACH, WIRE, MOBILE]
        location:
          $ref: '#/components/schemas/GeoLocation'
        deviceFingerprint:
          type: string
          maxLength: 256
```

```yaml
    TransactionResponse:
      type: object
      properties:
        transactionId:
          type: string
          format: uuid
        status:
          type: string
          enum: [ACCEPTED, REJECTED, PENDING]
        riskScore:
          type: number
          format: decimal
          minimum: 0
          maximum: 1000
        decision:
          type: string
          enum: [APPROVE, DECLINE, REVIEW]
        processingTimeMs:
          type: integer
        explanation:
          type: string
```

6. Performance Optimization Implementation

6.1 Caching Strategy

```java
@Configuration
@EnableCaching
public class CacheConfiguration {

    @Bean
    public CacheManager cacheManager() {
        RedisCacheManager.Builder builder = RedisCacheManager
            .RedisCacheManagerBuilder
            .fromConnectionFactory(redisConnectionFactory())
            .cacheDefaults(cacheConfiguration());

        return builder.build();
    }

    private RedisCacheConfiguration cacheConfiguration() {
        return RedisCacheConfiguration.defaultCacheConfig()
            .entryTtl(Duration.ofMinutes(5))
            .serializeKeysWith(RedisSerializationContext.SerializationPair
                .fromSerializer(new StringRedisSerializer()))
            .serializeValuesWith(RedisSerializationContext.SerializationPair
                .fromSerializer(new GenericJackson2JsonRedisSerializer()));
    }

    // Cache configurations for different data types
    @Bean
    public RedisCacheConfiguration featureCacheConfig() {
        return RedisCacheConfiguration.defaultCacheConfig()
            .entryTtl(Duration.ofMinutes(5))
            .prefixCacheNameWith("features:");
    }

    @Bean
    public RedisCacheConfiguration modelCacheConfig() {
        return RedisCacheConfiguration.defaultCacheConfig()
            .entryTtl(Duration.ofHours(24))
            .prefixCacheNameWith("models:");
    }
}
```

6.2 Connection Pool Configuration

```yaml
# Database connection pool
spring:
  datasource:
    hikari:
      maximum-pool-size: 50
      minimum-idle: 10
      connection-timeout: 30000
      idle-timeout: 600000
      max-lifetime: 1800000
```

```
    leak-detection-threshold: 60000

# Redis connection pool
redis:
  lettuce:
    pool:
      max-active: 100
      max-idle: 50
      min-idle: 10
      max-wait: 30000ms
    timeout: 5000ms
```

This LLD provides comprehensive implementation-ready specifications that development teams can use to build the banking fraud detection system while maintaining full traceability to all previous requirements documents. # Pseudocode Implementation ## Banking Fraud Detection Real-Time Analytics System

Building upon PRD, FRD, NFRD, Architecture Diagram, HLD, and LLD for executable implementation logic

ETVX Framework

ENTRY CRITERIA

- ✅ PRD completed with business objectives and success metrics
- ✅ FRD completed with 114 functional requirements (FR-001 to FR-114)
- ✅ NFRD completed with 138 non-functional requirements (NFR-001 to NFR-138)
- ✅ Architecture Diagram completed with technology stack and system architecture
- ✅ HLD completed with detailed component specifications and interfaces
- ✅ LLD completed with implementation-ready class structures and database schemas
- ✅ Development environment prepared with required tools and frameworks

TASK

Create executable pseudocode that implements all system components with detailed algorithms, control flows, error handling, and optimization logic that developers can directly translate into production code for the banking fraud detection system.

VERIFICATION & VALIDATION

Verification Checklist: - [ ] Pseudocode covers all functional requirements from FRD (FR-001 to FR-114) - [ ] Algorithms implement performance optimizations for NFR targets (<100ms processing) - [ ] Error handling and edge cases covered for financial transaction processing - [ ] Security controls implemented according to PCI DSS and banking regulations - [ ] Integration patterns match specifications from HLD and LLD - [ ] Code structure follows banking industry standards and best practices

Validation Criteria: - [ ] Pseudocode review completed with senior developers and architects - [ ] Algorithm complexity analysis confirms performance targets achievable - [ ] Security implementation review validates compliance requirements - [ ] Integration logic validated with external system specifications - [ ] Error handling scenarios tested against failure mode analysis - [ ] Code readiness confirmed for direct implementation by development team

EXIT CRITERIA

- ✅ Complete executable pseudocode ready for implementation
- ✅ All system components covered with detailed algorithm specifications
- ✅ Performance optimizations and security controls implemented
- ✅ Error handling and resilience patterns included
- ✅ Implementation guide ready for development team execution

---

Reference to Previous Documents

This Pseudocode implements executable logic based on **ALL** previous documents: - **PRD Success Metrics** → Algorithm implementations targeting >99% detection accuracy, <100ms processing, 99.99% uptime - **PRD Target Users** → User interface logic for fraud analysts, risk managers, compliance officers - **FRD Functional Requirements (FR-001-114)** → Complete algorithm implementations for all system functions - **NFRD Performance Requirements (NFR-001-138)** →

Optimized algorithms with caching, parallel processing, performance monitoring - **Architecture Diagram** → Implementation using specified technology stack (Kafka, Redis, PostgreSQL, ML frameworks) - **HLD Component Design** → Detailed service implementations with interface contracts and data flows - **LLD Implementation Specs** → Direct translation of class structures, database operations, and API implementations

1. Main Transaction Processing Pipeline

1.1 Real-Time Transaction Processing Algorithm

```
ALGORITHM ProcessTransactionRealTime
INPUT: transaction_request (JSON)
OUTPUT: fraud_decision (APPROVE/DECLINE/REVIEW), risk_score, explanation
TIME_CONSTRAINT: < 100ms total processing time

BEGIN
    start_time = getCurrentTimestamp()

    // Step 1: Validate and Parse Transaction (Target: <5ms)
    TRY
        validated_transaction = validateTransaction(transaction_request)
        IF NOT validated_transaction.isValid THEN
            RETURN createErrorResponse("INVALID_TRANSACTION", validation_errors)
        END IF
    CATCH ValidationException e
        logError("Transaction validation failed", e)
        RETURN createErrorResponse("VALIDATION_ERROR", e.message)
    END TRY

    // Step 2: Enrich Transaction Data (Target: <10ms)
    enriched_transaction = enrichTransactionData(validated_transaction)

    // Step 3: Parallel Processing Pipeline
    PARALLEL_EXECUTE
        // Thread 1: ML Model Inference (Target: <50ms)
        ml_result = executeMLPipeline(enriched_transaction)

        // Thread 2: Rule Engine Evaluation (Target: <20ms)
        rule_result = executeRuleEngine(enriched_transaction)

        // Thread 3: Feature Store Lookup (Target: <5ms)
        features = getRealtimeFeatures(enriched_transaction.customerId)
    END_PARALLEL

    // Step 4: Composite Risk Scoring (Target: <10ms)
    composite_score = calculateCompositeRiskScore(ml_result, rule_result, features)

    // Step 5: Decision Making (Target: <5ms)
    decision = makeDecision(composite_score)

    // Step 6: Generate Explanation (Target: <10ms)
    explanation = generateExplanation(ml_result, rule_result, composite_score)

    // Step 7: Audit and Logging
    auditTransaction(enriched_transaction, decision, composite_score)

    processing_time = getCurrentTimestamp() - start_time

    // Performance monitoring
    IF processing_time > 100 THEN
        alertSlowProcessing(enriched_transaction.id, processing_time)
    END IF

    RETURN createResponse(decision, composite_score, explanation, processing_time)
END
```

1.2 Transaction Validation Algorithm

```
ALGORITHM validateTransaction
INPUT: transaction_request
OUTPUT: validated_transaction OR validation_errors

BEGIN
    validation_errors = []
```

```
    // Basic field validation
    IF transaction_request.customerId IS NULL OR LENGTH(customerId) = 0 THEN
        validation_errors.ADD("Customer ID is required")
    END IF

    IF transaction_request.amount <= 0 OR transaction_request.amount > 1000000 THEN
        validation_errors.ADD("Invalid transaction amount")
    END IF

    IF NOT isValidCurrency(transaction_request.currency) THEN
        validation_errors.ADD("Invalid currency code")
    END IF

    // ISO 8583 format validation
    IF transaction_request.channel = "CARD" THEN
        IF NOT validateISO8583Format(transaction_request) THEN
            validation_errors.ADD("Invalid ISO 8583 message format")
        END IF
    END IF

    // Timestamp validation
    current_time = getCurrentTimestamp()
    IF ABS(transaction_request.timestamp - current_time) > 300000 THEN // 5 minutes
        validation_errors.ADD("Transaction timestamp too old or future")
    END IF

    // Geographic validation
    IF transaction_request.location IS NOT NULL THEN
        IF NOT isValidGeoLocation(transaction_request.location) THEN
            validation_errors.ADD("Invalid geographic coordinates")
        END IF
    END IF

    IF LENGTH(validation_errors) > 0 THEN
        RETURN ValidationResult(false, validation_errors)
    ELSE
        RETURN ValidationResult(true, createValidatedTransaction(transaction_request))
    END IF
END
```

2. Machine Learning Pipeline Implementation

2.1 ML Ensemble Processing Algorithm

```
ALGORITHM executeMLPipeline
INPUT: enriched_transaction
OUTPUT: ml_result (score, confidence, feature_importance)
TIME_CONSTRAINT: < 50ms

BEGIN
    start_time = getCurrentTimestamp()

    // Step 1: Feature Engineering (Target: <10ms)
    feature_vector = engineerFeatures(enriched_transaction)

    // Step 2: Feature Normalization
    normalized_features = normalizeFeatures(feature_vector)

    // Step 3: Parallel Model Inference
    PARALLEL_EXECUTE
        // Random Forest Model
        rf_prediction = randomForestPredict(normalized_features)
        rf_confidence = calculateConfidence(rf_prediction)

        // Isolation Forest Model
        if_anomaly_score = isolationForestPredict(normalized_features)
        if_prediction = convertAnomalyToFraudScore(if_anomaly_score)

        // Neural Network Model
        nn_prediction = neuralNetworkPredict(normalized_features)
        nn_confidence = calculateNNConfidence(nn_prediction)

        // XGBoost Model
        xgb_prediction = xgboostPredict(normalized_features)
        xgb_importance = getFeatureImportance(xgb_prediction)
    END_PARALLEL
```

```
    // Step 4: Ensemble Combination
    ensemble_score = combineModelPredictions(
        rf_prediction, if_prediction, nn_prediction, xgb_prediction
    )

    // Step 5: Calculate Overall Confidence
    overall_confidence = calculateEnsembleConfidence(
        rf_confidence, nn_confidence, ensemble_score
    )

    // Step 6: Feature Importance Analysis
    feature_importance = aggregateFeatureImportance(
        xgb_importance, rf_prediction.feature_importance
    )

    processing_time = getCurrentTimestamp() - start_time

    RETURN MLResult(
        score: ensemble_score,
        confidence: overall_confidence,
        feature_importance: feature_importance,
        individual_predictions: [rf_prediction, if_prediction, nn_prediction, xgb_prediction],
        processing_time: processing_time
    )
END
```

2.2 Feature Engineering Algorithm

```
ALGORITHM engineerFeatures
INPUT: enriched_transaction
OUTPUT: feature_vector (array of numerical features)

BEGIN
    features = []
    customer_id = enriched_transaction.customerId

    // Transaction-based features
    features.ADD(enriched_transaction.amount)
    features.ADD(getHourOfDay(enriched_transaction.timestamp))
    features.ADD(getDayOfWeek(enriched_transaction.timestamp))
    features.ADD(getMerchantRiskScore(enriched_transaction.merchantId))

    // Customer behavioral features (from cache/feature store)
    customer_profile = getCustomerProfile(customer_id)
    features.ADD(customer_profile.avg_transaction_amount_7d)
    features.ADD(customer_profile.transaction_frequency_24h)
    features.ADD(customer_profile.unique_merchants_30d)

    // Geographic features
    IF enriched_transaction.location IS NOT NULL THEN
        usual_location = customer_profile.usual_location
        distance = calculateDistance(enriched_transaction.location, usual_location)
        features.ADD(distance)
        features.ADD(getLocationRiskScore(enriched_transaction.location))
    ELSE
        features.ADD(0) // Default distance
        features.ADD(0.5) // Neutral location risk
    END IF

    // Velocity features
    recent_transactions = getRecentTransactions(customer_id, 3600) // Last hour
    features.ADD(LENGTH(recent_transactions))
    features.ADD(SUM(recent_transactions.amount))

    // Device features
    IF enriched_transaction.deviceFingerprint IS NOT NULL THEN
        device_trust_score = getDeviceTrustScore(enriched_transaction.deviceFingerprint)
        features.ADD(device_trust_score)
    ELSE
        features.ADD(0.5) // Neutral device trust
    END IF

    // Time-based features
    last_transaction = getLastTransaction(customer_id)
    IF last_transaction IS NOT NULL THEN
        time_since_last = enriched_transaction.timestamp - last_transaction.timestamp
```

```
        features.ADD(time_since_last)
    ELSE
        features.ADD(86400000) // Default 24 hours
    END IF

    // Merchant category features
    merchant_category = getMerchantCategory(enriched_transaction.merchantId)
    category_risk = getCategoryRiskScore(merchant_category)
    features.ADD(category_risk)

    RETURN features
END
```

3. Rule Engine Implementation

3.1 Regulatory Compliance Rule Execution

```
ALGORITHM executeRuleEngine
INPUT: enriched_transaction
OUTPUT: rule_result (compliance_score, fraud_score, triggered_rules)
TIME_CONSTRAINT: < 20ms

BEGIN
    triggered_rules = []
    compliance_score = 0
    fraud_score = 0

    // AML/KYC Rules Evaluation
    aml_result = evaluateAMLRules(enriched_transaction)
    compliance_score += aml_result.score
    triggered_rules.APPEND(aml_result.triggered_rules)

    // OFAC Sanctions Screening
    sanctions_result = screenSanctions(enriched_transaction)
    IF sanctions_result.hasMatch THEN
        compliance_score += 50
        triggered_rules.ADD("SANCTIONS_MATCH")

        // Auto-decline for exact OFAC match
        IF sanctions_result.isExactMatch THEN
            RETURN RuleResult(
                compliance_score: 100,
                fraud_score: 0,
                triggered_rules: triggered_rules,
                auto_decision: "DECLINE"
            )
        END IF
    END IF

    // Fraud Pattern Rules
    fraud_result = evaluateFraudRules(enriched_transaction)
    fraud_score += fraud_result.score
    triggered_rules.APPEND(fraud_result.triggered_rules)

    // Velocity Rules
    velocity_result = evaluateVelocityRules(enriched_transaction)
    fraud_score += velocity_result.score
    triggered_rules.APPEND(velocity_result.triggered_rules)

    // Geographic Rules
    geo_result = evaluateGeographicRules(enriched_transaction)
    fraud_score += geo_result.score
    triggered_rules.APPEND(geo_result.triggered_rules)

    RETURN RuleResult(
        compliance_score: MIN(compliance_score, 100),
        fraud_score: MIN(fraud_score, 100),
        triggered_rules: triggered_rules,
        auto_decision: NULL
    )
END
```

3.2 AML Rules Evaluation Algorithm

```
ALGORITHM evaluateAMLRules
INPUT: enriched_transaction
```

```
OUTPUT: aml_result (score, triggered_rules)

BEGIN
    score = 0
    triggered_rules = []
    customer_profile = getCustomerProfile(enriched_transaction.customerId)

    // Large Cash Transaction Rule
    IF enriched_transaction.amount > 10000 AND enriched_transaction.paymentMethod = "CASH" THEN
        score += 30
        triggered_rules.ADD("LARGE_CASH_TRANSACTION")

        // Generate CTR if required
        IF enriched_transaction.amount > 10000 THEN
            generateCTR(enriched_transaction)
        END IF
    END IF

    // Structuring Detection
    daily_cash_total = getDailyCashTotal(enriched_transaction.customerId)
    IF daily_cash_total > 10000 AND hasMultipleSmallCashTransactions(enriched_transaction.customerId)
THEN
        score += 40
        triggered_rules.ADD("STRUCTURING_PATTERN")
    END IF

    // Cross-Border Transaction
    IF enriched_transaction.isInternational AND enriched_transaction.amount > 3000 THEN
        score += 20
        triggered_rules.ADD("CROSS_BORDER_TRANSACTION")

        // Enhanced due diligence for high-risk countries
        IF isHighRiskCountry(enriched_transaction.destinationCountry) THEN
            score += 20
            triggered_rules.ADD("HIGH_RISK_COUNTRY")
        END IF
    END IF

    // PEP (Politically Exposed Person) Check
    IF customer_profile.isPEP THEN
        score += 25
        triggered_rules.ADD("PEP_TRANSACTION")
    END IF

    // Rapid Fire Transactions
    transaction_count_24h = getTransactionCount24h(enriched_transaction.customerId)
    IF transaction_count_24h > 50 THEN
        score += 25
        triggered_rules.ADD("RAPID_FIRE_TRANSACTIONS")
    END IF

    RETURN AMLResult(score, triggered_rules)
END
```

4. Risk Scoring and Decision Making

4.1 Composite Risk Score Calculation

```
ALGORITHM calculateCompositeRiskScore
INPUT: ml_result, rule_result, features
OUTPUT: composite_score (0-1000), risk_band, contributing_factors

BEGIN
    // Weighted combination of scores
    ml_weight = 0.6
    rule_weight = 0.3
    behavioral_weight = 0.1

    // Normalize scores to 0-1000 scale
    normalized_ml_score = ml_result.score * 1000
    normalized_rule_score = (rule_result.fraud_score + rule_result.compliance_score) * 5

    // Calculate behavioral score from features
    behavioral_score = calculateBehavioralScore(features) * 1000

    // Composite calculation
```

```
    composite_score = (
        normalized_ml_score * ml_weight +
        normalized_rule_score * rule_weight +
        behavioral_score * behavioral_weight
    )

    // Apply business logic adjustments
    IF rule_result.auto_decision = "DECLINE" THEN
        composite_score = 1000 // Force maximum risk
    END IF

    // Determine risk band
    risk_band = determineRiskBand(composite_score)

    // Identify contributing factors
    contributing_factors = []

    IF normalized_ml_score > 600 THEN
        contributing_factors.ADD("ML_HIGH_RISK")
    END IF

    IF rule_result.compliance_score > 50 THEN
        contributing_factors.ADD("COMPLIANCE_RISK")
    END IF

    IF rule_result.fraud_score > 50 THEN
        contributing_factors.ADD("FRAUD_PATTERN")
    END IF

    FOR EACH rule IN rule_result.triggered_rules DO
        contributing_factors.ADD(rule)
    END FOR

    RETURN CompositeRiskResult(
        score: ROUND(composite_score),
        risk_band: risk_band,
        contributing_factors: contributing_factors,
        ml_contribution: normalized_ml_score * ml_weight,
        rule_contribution: normalized_rule_score * rule_weight,
        behavioral_contribution: behavioral_score * behavioral_weight
    )
END
```

4.2 Decision Making Algorithm

```
ALGORITHM makeDecision
INPUT: composite_score
OUTPUT: decision (APPROVE/DECLINE/REVIEW), confidence

BEGIN
    // Risk band thresholds
    LOW_THRESHOLD = 300
    MEDIUM_THRESHOLD = 600
    HIGH_THRESHOLD = 800

    // Decision logic
    IF composite_score.score <= LOW_THRESHOLD THEN
        decision = "APPROVE"
        confidence = 0.95

    ELSE IF composite_score.score <= MEDIUM_THRESHOLD THEN
        decision = "REVIEW"
        confidence = 0.80

    ELSE IF composite_score.score <= HIGH_THRESHOLD THEN
        decision = "REVIEW"
        confidence = 0.90

    ELSE
        decision = "DECLINE"
        confidence = 0.95
    END IF

    // Override for regulatory compliance
    IF "SANCTIONS_MATCH" IN composite_score.contributing_factors THEN
        decision = "DECLINE"
        confidence = 1.0
```

```
    END IF

    // Override for critical fraud patterns
    IF "IMPOSSIBLE_TRAVEL" IN composite_score.contributing_factors THEN
        decision = "DECLINE"
        confidence = 0.98
    END IF

    RETURN DecisionResult(decision, confidence)
END
```

5. Explainable AI Implementation

5.1 Explanation Generation Algorithm

```
ALGORITHM generateExplanation
INPUT: ml_result, rule_result, composite_score
OUTPUT: explanation (natural_language, technical_details, visual_data)

BEGIN
    explanation_parts = []

    // Primary risk factors
    IF composite_score.score > 600 THEN
        explanation_parts.ADD("This transaction has been flagged as high risk.")
    ELSE IF composite_score.score > 300 THEN
        explanation_parts.ADD("This transaction requires additional review.")
    ELSE
        explanation_parts.ADD("This transaction appears to be legitimate.")
    END IF

    // ML model contribution
    IF ml_result.score > 0.7 THEN
        top_features = getTopFeatures(ml_result.feature_importance, 3)
        explanation_parts.ADD("Machine learning models identified suspicious patterns based on:")

        FOR EACH feature IN top_features DO
            feature_explanation = explainFeature(feature)
            explanation_parts.ADD("- " + feature_explanation)
        END FOR
    END IF

    // Rule engine contribution
    IF LENGTH(rule_result.triggered_rules) > 0 THEN
        explanation_parts.ADD("The following business rules were triggered:")

        FOR EACH rule IN rule_result.triggered_rules DO
            rule_explanation = getRuleExplanation(rule)
            explanation_parts.ADD("- " + rule_explanation)
        END FOR
    END IF

    // Regulatory compliance
    IF rule_result.compliance_score > 50 THEN
        explanation_parts.ADD("Regulatory compliance concerns were identified:")
        compliance_explanation = getComplianceExplanation(rule_result)
        explanation_parts.ADD("- " + compliance_explanation)
    END IF

    // Counterfactual explanation
    counterfactual = generateCounterfactual(ml_result, rule_result)
    IF counterfactual IS NOT NULL THEN
        explanation_parts.ADD("To reduce risk score: " + counterfactual)
    END IF

    // Technical details for analysts
    technical_details = {
        "ml_score": ml_result.score,
        "rule_score": rule_result.fraud_score,
        "compliance_score": rule_result.compliance_score,
        "composite_score": composite_score.score,
        "processing_time": ml_result.processing_time,
        "model_versions": getModelVersions(),
        "feature_importance": ml_result.feature_importance
    }
```

```
    // Visual data for dashboard
    visual_data = {
        "risk_gauge": composite_score.score,
        "contribution_chart": {
            "ml": composite_score.ml_contribution,
            "rules": composite_score.rule_contribution,
            "behavioral": composite_score.behavioral_contribution
        },
        "feature_importance_chart": ml_result.feature_importance,
        "timeline": getTransactionTimeline(enriched_transaction.customerId)
    }

    natural_language = JOIN(explanation_parts, " ")

    RETURN ExplanationResult(
        natural_language: natural_language,
        technical_details: technical_details,
        visual_data: visual_data,
        confidence: ml_result.confidence
    )
END
```

6. Alert and Case Management

6.1 Alert Generation Algorithm

```
ALGORITHM generateAlert
INPUT: transaction, decision_result, composite_score
OUTPUT: alert (priority, routing, notification)

BEGIN
    // Skip alert generation for approved transactions
    IF decision_result.decision = "APPROVE" THEN
        RETURN NULL
    END IF

    // Calculate alert priority
    priority_score = calculateAlertPriority(transaction, composite_score)

    // Determine priority level
    IF priority_score >= 80 THEN
        priority = "CRITICAL"
        sla_hours = 1
    ELSE IF priority_score >= 60 THEN
        priority = "HIGH"
        sla_hours = 4
    ELSE IF priority_score >= 40 THEN
        priority = "MEDIUM"
        sla_hours = 24
    ELSE
        priority = "LOW"
        sla_hours = 72
    END IF

    // Route to appropriate analyst queue
    analyst_queue = routeToAnalystQueue(transaction, priority)

    // Check for alert suppression
    IF shouldSuppressAlert(transaction, composite_score) THEN
        RETURN NULL
    END IF

    // Create alert
    alert = Alert(
        id: generateUUID(),
        transaction_id: transaction.id,
        customer_id: transaction.customerId,
        priority: priority,
        risk_score: composite_score.score,
        assigned_queue: analyst_queue,
        sla_deadline: getCurrentTimestamp() + (sla_hours * 3600000),
        created_at: getCurrentTimestamp(),
        status: "NEW"
    )

    // Send notifications
```

```
    sendAlertNotifications(alert, analyst_queue)

    // Store alert
    storeAlert(alert)

    RETURN alert
END
```

6.2 Case Creation and Management

```
ALGORITHM createFraudCase
INPUT: alert, related_transactions
OUTPUT: fraud_case

BEGIN
    // Check for existing cases for this customer
    existing_cases = getOpenCases(alert.customer_id)

    // Consolidate with existing case if appropriate
    IF LENGTH(existing_cases) > 0 AND shouldConsolidate(alert, existing_cases) THEN
        existing_case = existing_cases[0]
        existing_case.transaction_ids.ADD(alert.transaction_id)
        existing_case.updated_at = getCurrentTimestamp()

        // Update priority if higher
        IF alert.priority > existing_case.priority THEN
            existing_case.priority = alert.priority
        END IF

        updateCase(existing_case)
        RETURN existing_case
    END IF

    // Create new case
    fraud_case = FraudCase(
        id: generateUUID(),
        transaction_ids: [alert.transaction_id],
        customer_id: alert.customer_id,
        priority: alert.priority,
        status: "OPEN",
        assigned_analyst: NULL,
        created_at: getCurrentTimestamp(),
        updated_at: getCurrentTimestamp(),
        tags: generateCaseTags(alert)
    )

    // Auto-assign if possible
    available_analyst = findAvailableAnalyst(alert.assigned_queue, alert.priority)
    IF available_analyst IS NOT NULL THEN
        fraud_case.assigned_analyst = available_analyst.id
        fraud_case.status = "ASSIGNED"
        notifyAnalyst(available_analyst, fraud_case)
    END IF

    // Store case
    storeCase(fraud_case)

    // Create audit trail
    auditCaseCreation(fraud_case, alert)

    RETURN fraud_case
END
```

7. Performance Monitoring and Optimization

7.1 Real-Time Performance Monitoring

```
ALGORITHM monitorPerformance
INPUT: processing_metrics
OUTPUT: performance_alerts, optimization_recommendations

BEGIN
    current_metrics = getCurrentMetrics()

    // Check latency thresholds
    IF current_metrics.avg_processing_time > 100 THEN
```

```
        generatePerformanceAlert("HIGH_LATENCY", current_metrics.avg_processing_time)

        // Identify bottlenecks
        bottlenecks = identifyBottlenecks(current_metrics)
        FOR EACH bottleneck IN bottlenecks DO
            recommendOptimization(bottleneck)
        END FOR
    END IF

    // Check throughput
    IF current_metrics.transactions_per_second < 50000 THEN
        generatePerformanceAlert("LOW_THROUGHPUT", current_metrics.transactions_per_second)

        // Scale resources if needed
        IF shouldAutoScale(current_metrics) THEN
            triggerAutoScaling()
        END IF
    END IF

    // Check error rates
    IF current_metrics.error_rate > 0.01 THEN // 1%
        generatePerformanceAlert("HIGH_ERROR_RATE", current_metrics.error_rate)

        // Analyze error patterns
        error_analysis = analyzeErrors(current_metrics.recent_errors)
        recommendErrorReduction(error_analysis)
    END IF

    // Check ML model performance
    IF current_metrics.model_accuracy < 0.95 THEN
        generatePerformanceAlert("MODEL_DEGRADATION", current_metrics.model_accuracy)

        // Trigger model retraining
        scheduleModelRetraining()
    END IF

    // Update performance dashboard
    updatePerformanceDashboard(current_metrics)
END
```

This comprehensive pseudocode provides executable implementation logic that development teams can directly translate into production code for the banking fraud detection system, maintaining full traceability to all previous requirements documents and ensuring implementation readiness.
FRAUDGUARD AI — COMPLETE HACKATHON SUBMISSION
==========

REAL-TIME BANKING FRAUD DETECTION MVP (Problem 03) Team: 140509_03 | Theme: AI for Industry (Financial Services)

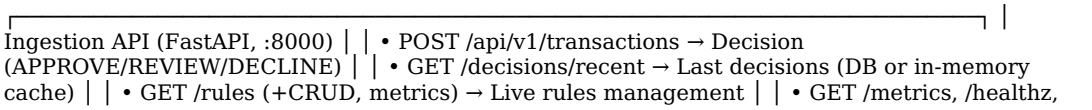
EXECUTIVE SUMMARY

FraudGuard AI is a real-time fraud detection MVP for banking payments that combines an efficient ML ensemble with a rule engine and rolling behavioral features to deliver sub-100ms risk decisions. The MVP exposes a clean ingestion API, a rule management surface, metrics, and optional PostgreSQL persistence. It demonstrates how to balance ML-based anomaly detection with explainable, auditable rules that fraud analysts can tune live.

KEY CAPABILITIES

• Real-time decisioning API with per-request explanations (ML + rules + features) • Lightweight ML ensemble (heuristics for RF/IF/NN/XGB) suitable for demo speed and clarity • Rule engine with live CRUD and metrics; YAML-backed with safe expression evaluation • Rolling feature store (velocity/amounts) with optional Redis backend • Optional Postgres persistence of transactions/decisions + recent decisions endpoint • Prometheus metrics and health probes for observability

ARCHITECTURE OVERVIEW



/stats/summary, /demo (static) │
└──┘ │ │
Feature Store Decision Service (InMemory or Redis) (ML Ensemble + Rules) • Velocity score 1h • Weighted ML probability (0..1) • 7-day avg/std amount • Rule engine score (0..200) • Composite risk (0..1000)

Optional persistence: PostgreSQL (:5432) • Upserts transactions + decisions; query recent decisions

DECISIONING LOGIC

1. Feature extraction (rolling aggregates per customer)
2. ML ensemble (heuristics for RF/IF/NN/XGB) over normalized signals → prob 0..1
3. Rules engine (YAML rules over features, e.g., z_score ≥ 3, velocity ≥ 10) → points
4. Composite risk: int(prob*800) + rule_points (capped at 1000)
5. Thresholds (configurable): ≤350 APPROVE, ≥700 DECLINE, else REVIEW
6. Explanation payload includes per-model scores, rule flags, features

REPO LAYOUT (src/)

• ingestion_service/main.py — FastAPI app, endpoints, static demo • decision_service/service.py — Feature store (in-mem/Redis), ensemble, rule engine, thresholds • ml_serving/service.py — Heuristic ensemble (RF/IF/NN/XGB-like) • rule_engine/engine.py — YAML rules loader + CRUD + metrics, safe eval of expressions • feature_store/service.py — Rolling aggregates (velocity, 7-day avg/std amount) • common/models.py — Pydantic models (TransactionRequest, DecisionResponse, enums) • common/config.py — Weights and decision thresholds • persistence/models.py — SQLAlchemy ORM models • persistence/repository.py — Upsert/query helpers; create tables • requirements.txt — FastAPI, SQLAlchemy, psycopg2, alembic, redis client, metrics

API SURFACE (Judge-Friendly)

Base: http://localhost:8000

Health & Metrics • GET /healthz → {status, db, uptimeSec, version} • GET /metrics → Prometheus exposition • GET /stats/summary → {totalRecent, byDecision}

Decisioning • POST /api/v1/transactions (TransactionRequest) → DecisionResponse {decision, riskScore, explanation{ml, rules, features}, processingMs} • GET /decisions/recent?limit=20 (DB if enabled; fallback to in-memory recent cache)

Rules (live tuning) • GET /rules → {version, rules{aml|fraud:[{name,condition,points}]}} • POST /rules → add rule {category, name, condition, points} • PUT /rules/{category}/{name} → update rule condition/points • DELETE /rules/{category}/{name} → remove rule • GET /rules/metrics → per-rule hit counts • POST /rules/metrics/reset → clear metrics

CURL EXAMPLES

1. Health curl -s localhost:8000/healthz | jq
2. Decision (APPROVE/REVIEW/DECLINE with explanation) curl -s -X POST localhost:8000/api/v1/transactions
   -H 'Content-Type: application/json'
   -d '{ "customerId": "CUST_001", "amount": 129.99, "currency": "USD", "merchantId": "M0001", "timestamp": "2025-08-30T12:00:00Z", "channel": "CARD", "location": {"latitude":40.7,"longitude":-74.0,"country":"US","city":"NYC"}, "deviceFingerprint": "dev-abc123" }' | jq
3. Rules — add a high-velocity fraud rule curl -s -X POST localhost:8000/rules -H 'Content-Type: application/json'
   -d
   '{"category":"fraud","name":"HighVelocity","condition":"velocity_score_1h>=10","points":40}'
   | jq
4. Recent decisions (DB or cache) curl -s localhost:8000/decisions/recent | jq

OBSERVABILITY

• Prometheus metrics at /metrics (http_requests_total, latency_seconds, decisions_total) • Healthz

shows API uptime and DB connectivity

PERSISTENCE (Optional, via docker-compose)

• Postgres 15 (:5432), volume-backed • Set env DATABASE_URL=postgresql://fraud:fraud@db:5432/fraud to enable persistence • Repository ensures idempotent table creation and upsert logic; recent decisions endpoint joins decision + transaction data

RUN & DEMO (One-Click)

Using Docker Compose: 1) cd 140509_03 2) docker compose up –build -d 3) Open: - API Health: http://localhost:8000/healthz - Demo UI: http://localhost:8000/demo/ - Metrics: http://localhost:8000/metrics - Recent: http://localhost:8000/decisions/recent

Local dev: • uvicorn ingestion_service.main:app –reload –app-dir 140509_03/src

EXPLAINABILITY & COMPLIANCE

• Per-decision explanation details ML components, rule flags, and feature values • Rules are auditable and editable via CRUD endpoints; changes tracked by version • Safe evaluation of rule conditions (restricted context) for MVP; can be replaced by a DSL in production

ROADMAP (Beyond MVP)

• Replace heuristic ensemble with trained RF/XGB/NN pipelines (same interface) • Kafka ingestion + Flink/Spark streaming features at high TPS • Full Redis feature store for cross-instance sharing • Advanced rules DSL with versioning, A/B tests, and performance analytics • Model monitoring (drift, PSI), canary deployments, automated retraining • Analyst console (case management, feedback loops) and alerting workflows

JUDGE EXPERIENCE CHECKLIST

• End-to-end run: POST transaction → decision, score, explanation under ~100ms • Live rules tuning: add/update/delete a rule and observe decision changes • Observability: metrics + healthz visible; recent decisions list populates • Persistence: enable DATABASE_URL and verify decisions persisted in Postgres • Explainability: ML per-model scores + rule flags visible per decision

— End of FraudGuard AI MVP Submission —