Fraud Detection Forensic Systems Backend

A comprehensive guide to building a production-grade fraud detection platform with FastAPI, PostgreSQL, and machine learning capabilities for real-time transaction monitoring and forensic investigation.

Executive Overview

This presentation documents the complete implementation of a sophisticated fraud detection and forensic investigation platform designed for financial crime investigation teams. The system combines real-time machine learning-based anomaly detection with rule-based compliance checks, comprehensive case management capabilities, and detailed audit trails.

The platform achieves sub-100ms inference targets for transaction scoring whilst maintaining the audit rigour and investigative workflows expected in Deloitte-style forensic engagements. Built on FastAPI for high-performance API operations, the backend leverages PyTorch autoencoders for anomaly detection, SQLAlchemy for robust database operations, and Alembic for managed schema migrations.

The system supports role-based access control with JWT authentication, immutable audit logging for regulatory compliance, entity relationship tracking for network analysis, and Prometheus metrics endpoints for operational monitoring. The architecture is containerised with Docker and ready for deployment on platforms such as Fly.io, with comprehensive documentation covering setup, operation, and forensic investigation workflows.

System Architecture Overview

Backend Core

FastAPI Python framework with SQLAlchemy ORM, Alembic migrations, and PostgreSQL database

- RESTful API design
- Async operation support
- Sub-100ms response targets

ML Pipeline

PyTorch autoencoder for anomaly detection with 18-feature engineering pipeline

- Real-time scoring engine
- Feature standardisation
- Model versioning support

Investigation Tools

Forensic agents system with case management and entity tracking capabilities

- Compliance checks
- Network graph analysis
- Audit trail logging

Security & Auth

JWT-based authentication with role-based access control and immutable audit logs

- Password hashing
- Token management
- RBAC enforcement

Technology Stack

Backend Technologies

Core Framework: FastAPI 0.104.1 provides high-performance async API operations with automatic OpenAPI documentation generation and Pydantic validation.

Database Layer: SQLAlchemy 2.0.23 for ORM operations with PostgreSQL, Alembic 1.12.1 for database migrations and schema versioning.

Machine Learning: PyTorch for autoencoder implementation, NumPy and Pandas for numerical operations and data manipulation, scikit-learn for feature preprocessing.

Security: python-jose for JWT token generation and validation, passlib with bcrypt for password hashing.

Observability: Prometheus client for metrics collection and monitoring.

Development & Deployment

Configuration: Pydantic Settings 2.1.0 for environment-based configuration management with validation.

Server: Uvicorn 0.24.0 with standard extras for ASGI server capabilities including WebSocket support.

Testing: pytest with async support configured for comprehensive unit and integration testing.

Containerisation: Docker multi-stage builds for optimised image sizes and faster deployment cycles.

Cloud Platform: Fly.io configuration for scalable deployment with managed PostgreSQL databases.

Project Initialisation

The project initialisation process encountered a PowerShell-specific issue with the `mkdir` command, which was subsequently resolved by using PowerShell-native directory creation methods. This highlights the importance of understanding platform-specific tooling when setting up development environments.

The initial setup created a comprehensive project structure with proper Python package organisation, including an `__init__.py` file to mark the application directory as a Python package. The requirements.txt file was configured with pinned versions to ensure reproducible builds across development, staging, and production environments.

The pyproject.toml file established pytest configuration with specific test discovery patterns, async mode support, and proper test path definitions. This configuration ensures consistent test execution across different development environments and CI/CD pipelines.

Dependencies Management

```
fastapi==0.104.1
uvicorn[standard]==0.24.0
sqlalchemy==2.0.23
alembic==1.12.1
pydantic==2.5.0
pydantic-settings==2.1.0
python-jose[cryptography]==3.3.0
passlib[bcrypt]==1.7.4
python-multipart==0.0.6
psycopg2-binary==2.9.9
torch==2.1.0
numpy==1.24.3
pandas==2.1.3
scikit-learn==1.3.2
prometheus-client==0.19.0
pytest==7.4.3
pytest-asyncio==0.21.1
```

All dependencies are pinned to specific versions to prevent unexpected breaking changes during deployment. The `uvicorn[standard]` package includes WebSocket and HTTP/2 support, whilst `python-jose[cryptography]` provides secure JWT implementations. The `psycopg2-binary` package offers precompiled PostgreSQL adapter binaries for faster installation, particularly valuable in containerised environments.

Application Configuration

The configuration system leverages Pydantic Settings to provide type-safe, validated configuration management with automatic environment variable loading. This approach ensures that misconfigured applications fail fast during startup rather than at runtime, significantly improving operational reliability.

The Settings class defines all configuration parameters with appropriate types, default values, and validation rules. Database connection strings, JWT secrets, token expiration times, and model paths are all managed through environment variables, allowing seamless configuration changes between development, staging, and production environments without code modifications.

Environment-specific settings are loaded from `.env` files during development whilst production deployments rely on platform-provided environment variables. The configuration supports optional parameters with sensible defaults, making local development straightforward whilst maintaining security in production through required secret values.

Key configuration areas include database connection parameters, authentication settings (JWT secret, algorithm, token expiration), model file paths for the trained autoencoder, and API-specific settings such as CORS origins and rate limiting parameters.

Database Connection Architecture

The database layer implements a robust connection management system using SQLAlchemy's create_engine function with connection pooling for optimal performance. The engine is configured with appropriate pool sizes, timeout settings, and echo parameters for development debugging.

A declarative base class provides the foundation for all ORM models, enabling Python classes to map directly to database tables with automatic schema generation capabilities. The sessionmaker factory creates database sessions with specific transaction isolation levels and autocommit behaviour configured for the application's needs.

The `get_db` dependency function implements proper session lifecycle management using FastAPI's dependency injection system. Each request receives a fresh database session that's automatically closed after the request completes, preventing connection leaks and ensuring proper transaction boundaries. The session is yielded rather than returned, allowing FastAPI to handle cleanup even when exceptions occur.

Data Models Overview

Core Transaction Models

Transaction, Score, and Entity models form the foundation of the fraud detection system, capturing transaction details, risk scores, and involved parties.

Case Management

Case, CaseEvent, CaseTransaction, and CaseEntity models enable comprehensive investigation workflows with full audit trails.

Security & Audit

User, AuditLog, and EntityLink models provide authentication, authorisation, and immutable audit logging capabilities.

Transaction Model

The Transaction model represents individual financial transactions with comprehensive tracking of all relevant attributes for fraud analysis. Each transaction captures temporal data (timestamp), parties involved (sender and recipient entities via foreign keys), financial details (amount, currency), and contextual information (merchant, location, device identifiers).

The model includes a `metadata_json` field for storing additional transaction attributes without schema modifications, providing flexibility for varying data sources. The `is_flagged` boolean indicates whether the transaction has been marked for investigation, whilst the `risk_level` enum field (LOW, MEDIUM, HIGH) provides quick filtering capabilities.

Relationships to the Score model (one-to-many) allow multiple scoring attempts with different model versions, and to Case models (many-to-many through CaseTransaction) enable linking transactions to ongoing investigations. The created_at and updated_at timestamps use `server_default=func.now()` for database-level timestamp management, ensuring consistency even with direct database operations.

Key Attributes

- Transaction ID (UUID)
- Timestamp (indexed)
- Amount and currency
- Sender/recipient entities
- Location data
- Device fingerprints
- Merchant details
- Risk level enum
- Metadata JSON field

Score Model

The Score model captures the output of fraud detection algorithms applied to transactions. Each score record links to a specific transaction and stores the calculated anomaly score (float between 0 and 1), risk level classification, and model version identifier for traceability.

The `features_json` field stores the engineered features used for scoring, enabling retrospective analysis and model debugging. This is particularly valuable when investigating why specific transactions received certain scores or when retraining models with improved feature engineering.

The `threshold` field records the decision boundary used during scoring, allowing for post-hoc threshold adjustments without recomputing anomaly scores. The `model_version` string enables tracking which trained model produced each score, essential for A/B testing new models and maintaining audit trails for regulated industries.

Multiple Score records per transaction support scenarios where transactions are rescored with updated models, threshold adjustments, or manual investigator overrides. The created_at timestamp tracks when scoring occurred, enabling temporal analysis of model performance.

Entity Model

The Entity model represents individuals or organisations involved in transactions, whether as senders, recipients, or merchants. Each entity has a unique identifier, type classification (INDIVIDUAL, BUSINESS, MERCHANT), and comprehensive profile information stored in the `profile_json` field.

The `risk_score` field maintains an aggregated risk assessment based on the entity's transaction history, updated asynchronously by the monitoring agent. The `is_blocked` boolean flag enables immediate transaction blocking for high-risk entities, whilst `tags` provide flexible categorisation for investigation workflows.

Relationships include sent and received transactions (one-to-many), entity links for network analysis (many-to-many self-referential relationship), and case associations through CaseEntity. This relationship structure enables sophisticated network graph analysis for identifying fraud rings and suspicious entity networks.

The model supports storing KYC verification data, historical addresses, associated devices, and behavioural patterns in the profile_json field, providing investigators with comprehensive entity histories during case investigations.

Case Management Models

01

Case Model

Core case tracking with title, description, status (OPEN, IN_PROGRESS, CLOSED, FALSE_POSITIVE), priority, and assigned investigator. Stores investigation notes and resolution details.

02

CaseEvent Model

Immutable audit trail of all case activities including status changes, investigator actions, and system events. Each event captures timestamp, actor, event type, and detailed descriptions.

03

CaseTransaction Model

Many-to-many association linking transactions to cases with additional metadata about why the transaction was included and investigator notes specific to that transaction-case relationship.

04

CaseEntity Model

Links entities to cases with role classification (SUSPECT, VICTIM, WITNESS, RELATED) and relationship descriptions, enabling comprehensive entity involvement tracking.

User and Authentication Models

User Model

The User model implements comprehensive authentication and authorisation with username, email, and hashed password fields. The `hashed_password` field stores bcrypt hashes, never plain text passwords, ensuring security even in the event of database compromise.

Role-based access control uses the UserRole enum (ADMIN, INVESTIGATOR, ANALYST, VIEWER) to define permission levels. The `is_active` boolean enables account suspension without data deletion, whilst `last_login` tracking supports security auditing.

The model includes relationships to assigned cases (one-to-many), created audit log entries (one-to-many), and department/team affiliations through the profile metadata.

Audit Log Model

The AuditLog model provides immutable logging of all system actions for regulatory compliance and forensic investigation. Each entry captures the user who performed the action, the affected entity and entity ID, the action type, and detailed before/after state in JSON format.

The `ip_address` field records the source of the action for security analysis, whilst `user_agent` provides device/browser context. The timestamp is server-generated and immutable, preventing tampering with audit trails.

Database Migration Strategy

Alembic provides powerful database migration capabilities with version control for schema changes. The alembic.ini configuration file defines migration script locations, database connection strings, and logging behaviour. The script_location parameter points to the alembic directory containing version-controlled migration scripts.

The env.py file configures the Alembic environment, importing the SQLAlchemy models and database configuration. It defines two migration modes: offline for generating SQL scripts without database connectivity, and online for direct database migrations. The `run_migrations_offline` and `run_migrations_online` functions handle these scenarios respectively.

The script.py.mako template generates new migration files with consistent formatting, including revision IDs, parent revision references, and creation timestamps. Each migration includes upgrade() and downgrade() functions for applying and reverting schema changes.

Migrations are applied using `alembic upgrade head` to move to the latest version, or `alembic upgrade <revision>` for specific versions. The `alembic downgrade` command reverts changes, essential for rollback scenarios during deployment issues.

Feature Engineering Pipeline

The FeatureEngineer class transforms raw transaction data into 18 engineered features optimised for anomaly detection. The pipeline implements sophisticated temporal, statistical, and behavioural feature extraction to capture fraud indicators that simple rule-based systems miss.

Temporal Features

- Hour of day (0-23)
- Day of week (0-6)
- Is weekend boolean
- Time since last transaction

Statistical Features

- Transaction amount (log-transformed)
- Amount Z-score vs. historical mean
- Velocity features (24h, 7d counts)
- Amount percentile in history

Behavioural Features

- New merchant indicator
- Geographic distance from previous
- Device fingerprint matching
- Recipient novelty score

Network Features

- Sender transaction count
- Recipient transaction count
- Merchant transaction frequency
- Entity clustering coefficient

Feature Engineering Implementation

The feature engineering implementation uses pandas DataFrames for efficient vectorised operations on historical transaction data. The StandardScaler from scikit-learn normalises features to zero mean and unit variance, crucial for neural network training and ensuring no single feature dominates the model.

The `calculate_velocity_features` method queries recent transaction history within specified time windows (24 hours, 7 days) to count transaction frequency and sum amounts. These velocity features are among the strongest fraud indicators, as fraudsters often perform multiple rapid transactions before accounts are frozen.

Geographic features leverage geohash encoding and Haversine distance calculations to measure how far a transaction's location deviates from the user's typical patterns. The `calculate_geo_distance` method handles missing coordinates gracefully whilst still providing useful signals when available.

The `fit` method pre-computes scaling parameters from a representative dataset, whilst `transform` applies these parameters to new transactions. This separation ensures that scoring doesn't leak information from test data and maintains consistency across model versions.

Key Methods

- `fit(transactions)`
- `transform(transaction)`
- `fit_transform(transactions)`
- `calculate_velocity_features()`
- `calculate_geo_distance()`
- `get_merchant_frequency()`
- `is_new_recipient()`
- calculate_amount_zscore()

Autoencoder Architecture

The PyTorch autoencoder implements an anomaly detection neural network with an hourglass architecture: encoder layers compress the 18 input features down to a 6-dimensional latent representation, then decoder layers reconstruct the original features. Fraud transactions, being anomalous, produce higher reconstruction errors than legitimate transactions.

The architecture uses three encoding layers (18→12→6) with ReLU activation functions, and three symmetric decoding layers (6→12→18). Dropout layers (p=0.2) prevent overfitting during training, and batch normalisation stabilises training dynamics. The network is deliberately shallow to avoid memorising training data, which would reduce anomaly detection performance.

The reconstruction loss uses Mean Squared Error (MSE) between input features and reconstructed outputs. During inference, transactions with reconstruction errors exceeding a learned threshold are flagged as anomalous. The threshold is typically set at the 95th or 99th percentile of training data reconstruction errors.

Autoencoder Training Process

Model training implements mini-batch gradient descent with the Adam optimiser, typically running for 50-100 epochs with early stopping based on validation loss. The training loop processes batches of normalised feature vectors, computing reconstruction loss and backpropagating gradients.

The `train_epoch` method iterates through training batches, performing forward passes to compute reconstructions, calculating MSE loss, and updating weights via backpropagation. The method returns average epoch loss for monitoring convergence.

Model checkpointing saves weights periodically using `torch.save(model.state_dict(), path)`, enabling training resumption and model versioning. The saved weights are loaded during API initialisation for inference.

Training data should exclude known fraud examples to ensure the model learns normal transaction patterns. The model is trained exclusively on legitimate transactions, making it particularly effective at detecting novel fraud patterns not seen during training.

Training Configuration

- Epochs: 50-100
- Batch size: 32-128
- Learning rate: 0.001
- Optimiser: Adam
- Loss: MSE
- Regularisation: Dropout 0.2
- Early stopping: Patience 5
- Validation split: 20%

Fraud Scoring Engine

The FraudScoringEngine class orchestrates the complete scoring pipeline: feature engineering, autoencoder inference, threshold comparison, and risk level classification. The engine is initialised once at application startup and processes each transaction through this pipeline with sub-100ms latency targets.

The `score_transaction` method accepts a Transaction object and historical context, extracts features using the FeatureEngineer, computes reconstruction error through the autoencoder, and classifies risk level based on configurable thresholds. The method returns a Score object containing the anomaly score, risk level, model version, and engineered features.

1

Feature Extraction

Transform transaction into 18 engineered features using historical context

2

Model Inference

Pass features through autoencoder to compute reconstruction error

3

Classification

Compare score to thresholds and assign risk level (LOW/MEDIUM/HIGH)

Risk Level Classification

The scoring engine implements a three-tier risk classification system based on reconstruction error thresholds. These thresholds are calibrated during model training to achieve desired false positive rates whilst maintaining high fraud detection rates.

LOW Risk

Reconstruction error below the 95th percentile of legitimate transactions. These transactions match normal patterns closely and require no manual review. Automatically approved for processing with minimal friction.

Typical Characteristics:

- Familiar merchant
- Expected amount range
- Known device/location
- Normal velocity

MEDIUM Risk

Reconstruction error between 95th-99th percentiles. These transactions show some anomalous characteristics but may be legitimate. Flagged for automated compliance checks and potential manual review.

Review Triggers:

- New merchant/recipient
- Unusual amount
- Geographic anomaly
- Elevated velocity

HIGH Risk

Reconstruction error above the 99th percentile. These transactions are highly anomalous and warrant immediate investigation. Automatically flagged for forensic review with case creation.

Typical Patterns:

- Multiple red flags
- Extreme deviation
- Suspicious patterns
- Known fraud indicators

Forensic Agent System

The forensic agent system implements four specialised agents that work together to detect fraud, ensure compliance, investigate cases, and monitor system health. Each agent operates semi-autonomously, triggered by events or scheduled tasks, and interacts with the database and external systems as needed.

Anomaly Detection Agent

Continuously monitors incoming transactions, scores them using the ML model, and flags high-risk transactions for investigation. Maintains scoring throughput and latency targets.

Compliance Agent

Executes rule-based checks for regulatory compliance including velocity limits, geographic restrictions, merchant blocklists, and amount thresholds. Operates independently of ML scores.

Investigation Agent

Assists human investigators by gathering related transactions, identifying entity networks, correlating events, and suggesting investigation priorities based on risk scores and patterns.

Monitoring Agent

Tracks system health metrics, model performance, scoring latency, and investigation outcomes. Exports metrics to Prometheus for alerting and dashboards.

Anomaly Detection Agent

The AnomalyDetectionAgent orchestrates the real-time fraud detection pipeline. When a new transaction arrives via the API, the agent immediately scores it using the FraudScoringEngine, persists the Score record to the database, and conditionally triggers downstream actions based on risk level.

For HIGH risk transactions, the agent automatically creates a Case record, assigns it to an available investigator based on workload, and generates a CaseEvent documenting the automatic case creation. The agent also updates the Transaction record's `is_flagged` field for efficient filtering in the investigation UI.

The agent implements circuit breaker patterns to handle model failures gracefully. If the ML model is unavailable, the agent falls back to rule-based scoring using the ComplianceAgent whilst logging errors for operational alerting. This ensures that the transaction pipeline continues operating even during model deployment or infrastructure issues.

Performance is critical: the agent must complete scoring within 50ms on average to maintain sub-100ms end-to-end API response times. The implementation uses async/await patterns, connection pooling, and result caching to achieve these targets.

Compliance Agent

The ComplianceAgent implements rule-based fraud detection and regulatory compliance checks that operate independently of ML scores. These rules encode business logic, regulatory requirements, and known fraud patterns that are deterministic rather than probabilistic.

Velocity checks prevent rapid-fire transactions by counting transaction frequency within time windows (24 hours, 7 days). Configurable thresholds trigger blocks when users exceed limits, preventing account draining attacks. The agent implements both count-based and amount-based velocity checks.

Geographic restriction rules block transactions from sanctioned countries or regions flagged for high fraud rates. The agent also enforces merchant blocklists, identifying businesses known for fraud or regulatory violations. Amount threshold checks flag unusually large transactions for mandatory manual review, satisfying anti-money laundering (AML) requirements.

The agent generates ComplianceViolation records for each failed check, capturing the rule violated, severity level, and remediation actions. These violations are linked to cases and inform investigation priorities.

Investigation Agent

The InvestigationAgent provides intelligent assistance to human investigators, automating routine investigative tasks and surfacing relevant context. When an investigator opens a case, the agent immediately gathers related transactions by querying for shared entities, similar amounts, temporal proximity, and matching merchants.

Network graph generation identifies entity relationships by traversing the EntityLink graph to specified depths. The agent constructs subgraphs showing how entities in a case are connected through transaction histories, shared addresses, devices, or explicit relationships. These graphs are particularly valuable for identifying fraud rings.

Pattern correlation applies clustering algorithms to identify groups of similar transactions that may represent coordinated fraud campaigns. The agent compares transaction features, timing patterns, and entity attributes to surface non-obvious relationships.

Investigation prioritisation ranks open cases based on risk scores, transaction amounts, entity risk profiles, and case age. The agent suggests which cases investigators should work on first, optimising investigation resource allocation.

Monitoring Agent

The MonitoringAgent implements comprehensive observability for the fraud detection system, tracking performance metrics, model health, investigation outcomes, and system resource utilisation. The agent exports metrics to Prometheus for alerting, dashboards, and long-term trend analysis.

Scoring metrics include latency percentiles (p50, p95, p99), throughput (transactions/second), model accuracy based on investigated outcomes, and false positive/negative rates. These metrics inform model retraining decisions and performance optimisation efforts.

Investigation metrics track case resolution times, investigator workload, case outcome distributions (true positive, false positive, inconclusive), and financial impact (prevented fraud amounts). These metrics demonstrate ROI and identify process bottlenecks.

Key Metrics

- Scoring latency (p50, p95, p99)
- Throughput (TPS)
- Model accuracy (%)
- False positive rate (%)
- Case resolution time (hours)
- Prevented fraud amount ($)
- System resource utilisation
- API error rates
- Database connection pool

Authentication System

The authentication system implements JWT (JSON Web Token) based authentication with bcrypt password hashing for secure credential storage. The system supports role-based access control, token expiration, and refresh token patterns for session management.

Password hashing uses bcrypt with a configurable work factor (default 12 rounds), making brute-force attacks computationally infeasible. The `verify_password` function compares provided passwords against stored hashes using constant-time comparison to prevent timing attacks.

JWT tokens encode user identity, role, and expiration timestamp, signed with a secret key using HS256 algorithm. The `create_access_token` function generates tokens with configurable expiration times (typically 30 minutes), whilst refresh tokens (not implemented in basic version) would have longer validity (7-30 days).

The `get_current_user` dependency extracts and validates JWT tokens from Authorization headers, decodes the payload, queries the database for the user record, and verifies that the user account is still active. This dependency is injected into all protected API endpoints.

API Schema Definitions

Pydantic schemas define the API contract between frontend and backend, providing automatic request validation, response serialisation, and OpenAPI documentation generation. Each schema class represents either a request payload, response body, or shared data structure.

01

Request Schemas

TransactionCreate, CaseCreate, CaseUpdateStatus define inbound payloads with validation rules, required fields, and default values. Pydantic validates these automatically, returning 422 errors for invalid requests.

02

Response Schemas

TransactionResponse, CaseResponse, EntityResponse serialise database models to JSON, including related objects and computed fields. The `from_orm` config enables automatic ORM model conversion.

03

Shared Schemas

ScoreResponse, EntityLinkResponse define structures used in multiple endpoints. These promote consistency and reduce duplication across the API surface.

Audit Logging System

The audit logging system provides immutable, tamper-evident records of all system actions for regulatory compliance, forensic investigation, and security monitoring. The system captures who performed each action, what was modified, when it occurred, and from where the action originated.

The `log_action` utility function creates AuditLog records automatically, capturing the authenticated user, target entity type and ID, action performed, IP address, user agent, and before/after state snapshots in JSON format. The function integrates with FastAPI's dependency injection to access request context.

Critical actions that always generate audit logs include: user authentication, case creation/updates, transaction flagging, entity blocking, compliance rule violations, and configuration changes. The logs are write-only; no API endpoint permits modification or deletion.

Audit queries support filtering by date range, user, entity type, and action type, enabling investigators to reconstruct timelines of events during forensic investigations or regulatory audits.

Logged Actions

- USER_LOGIN / USER_LOGOUT
- TRANSACTION_CREATED
- TRANSACTION_SCORED
- TRANSACTION_FLAGGED
- CASE_CREATED / CASE_UPDATED
- CASE_STATUS_CHANGED
- ENTITY_BLOCKED
- COMPLIANCE_VIOLATION
- CONFIG_CHANGED

Transaction API Endpoints

1

POST /api/transactions

Creates a new transaction, scores it via AnomalyDetectionAgent, and returns the Transaction with Score. Accepts TransactionCreate payload with validation.

2

GET /api/transactions

Lists transactions with pagination, filtering by risk level, date range, entity ID, and flagged status. Returns paginated TransactionResponse list.

3

GET /api/transactions/{id}

Retrieves a single transaction by ID with all associated scores, case links, and entity details. Returns comprehensive TransactionResponse.

4

POST /api/transactions/{id}/flag

Manually flags a transaction for investigation, creating a case if one doesn't exist. Requires INVESTIGATOR role or higher.

All endpoints require authentication via JWT token. The create endpoint triggers automatic scoring and potential case creation for HIGH risk transactions. The flag endpoint allows investigators to manually escalate MEDIUM risk transactions based on additional context not captured by the model.

Case Management API

The case management API provides comprehensive endpoints for creating, updating, and querying fraud investigation cases. These endpoints enable investigators to manage their workloads, track investigation progress, and document findings.

The POST /api/cases endpoint creates new cases with title, description, priority, and initial transaction associations. The system automatically assigns cases to investigators based on current workload and specialisation, though investigators can manually reassign cases as needed.

The PATCH /api/cases/{id}/status endpoint updates case status through the defined workflow: OPEN → IN_PROGRESS → CLOSED or FALSE_POSITIVE. Status transitions automatically generate CaseEvent records for audit trails, capturing the status change, investigator, timestamp, and optional notes.

The GET /api/cases endpoint supports sophisticated filtering by status, assigned investigator, priority, date range, and associated entity. Investigators use these filters to build personalised work queues, focusing on high-priority or time-sensitive cases.

The POST /api/cases/{id}/events endpoint allows investigators to add timestamped notes, attach evidence, record interviews, and document investigation steps. These events form a complete investigation history that satisfies regulatory requirements and supports legal proceedings.

Entity Network Analysis

The entity API provides endpoints for querying entity profiles, transaction histories, and relationship networks. These capabilities are essential for understanding the context around suspicious transactions and identifying fraud rings.

GET /api/entities/{id} retrieves comprehensive entity profiles including aggregated risk scores, transaction counts, recent activity summaries, and profile metadata (KYC data, addresses, devices). This 360-degree view helps investigators assess entity legitimacy quickly.

GET /api/entities/{id}/network constructs network graphs by traversing EntityLink relationships to specified depths (typically 1-3 hops). The response includes nodes (entities) and edges (relationships with types and metadata), formatted for graph visualisation libraries.

POST /api/entities/{id}/block immediately prevents an entity from sending or receiving transactions, used when investigators confirm fraud. The endpoint updates the entity's `is_blocked` flag and generates audit logs.

Network Analysis Features

- Multi-hop relationship traversal
- Entity clustering identification
- Shared attribute detection
- Transaction flow visualisation
- Temporal pattern analysis
- Device/location sharing
- Suspicious pattern detection

Metrics and Monitoring API

The /api/metrics endpoint exposes Prometheus-compatible metrics in text format, enabling integration with standard observability stacks (Prometheus, Grafana, AlertManager). The endpoint requires no authentication to support pull-based monitoring architectures.

Exported metrics include counters (transaction count, case count, fraud prevented count), gauges (active cases, average risk score, system load), histograms (scoring latency distribution, case resolution time distribution), and summaries (throughput percentiles, error rates).

The MonitoringAgent updates these metrics continuously as transactions are processed, cases are created/resolved, and system events occur. Prometheus scrapes the endpoint every 15-60 seconds, storing time-series data for alerting and long-term analysis.

Common alerts configured from these metrics include: scoring latency exceeding SLA thresholds, false positive rate increasing above baselines, case backlog growing, model accuracy degrading, and system error rates spiking. These alerts enable proactive operational response before issues impact users.

Main Application Setup

The main.py file orchestrates the FastAPI application, configuring middleware, registering routers, initialising the ML model, and exposing the ASGI application for Uvicorn to serve.

The FastAPI instance is configured with title, version, and OpenAPI documentation settings. CORS middleware is added to allow frontend applications from configured origins to make cross-origin requests, essential for the Next.js frontend.

API routers are included with path prefixes: /api/transactions, /api/cases, /api/entities, /api/auth, /api/metrics. This organisation keeps the codebase modular whilst providing a clean, predictable API surface.

On startup, the application creates database tables (Base.metadata.create_all), loads the trained autoencoder model, and initialises the FeatureEngineer with scaling parameters. The FraudScoringEngine is instantiated once and reused across requests for efficiency.

The root path (/) provides API health information including version, status, and uptime. The /docs path serves interactive Swagger UI documentation generated automatically from schemas.

Startup Tasks

- Create database tables
- Load ML model weights
- Initialise feature scaler
- Start monitoring agent
- Configure logging
- Verify configuration
- Warm up connection pools

Frontend Architecture

The Next.js frontend provides a responsive, type-safe React application for fraud investigators and analysts. The architecture leverages Next.js App Router for file-based routing, Server Components for performance, and Material-UI for sophisticated component design.

TypeScript ensures type safety across the application, catching errors at compile time and providing excellent IDE support. The app/layout.tsx defines the root layout with theme provider, authentication context, and global styling.

The frontend implements a single-page application (SPA) pattern with client-side routing for fast navigation between dashboard, transactions, cases, and entity views. API calls use axios with interceptors for JWT token injection and error handling.

The theme system supports light and dark modes, automatically persisting user preferences to localStorage. Material-UI components are customised with the application colour palette for consistent branding.

Frontend Routing Structure

/login

Authentication page with username/password form, JWT token acquisition, and role-based routing to appropriate dashboards.

/dashboard

Overview page with KPI cards (transaction counts, risk distributions, case statistics), recent transactions table, and alert summaries.

/transactions

Searchable, filterable transaction list with risk indicators, date range pickers, and drill-down to detail views.

/cases

Case management workspace with status filters, priority sorting, investigator assignment, and comprehensive case detail pages.

/entities

Entity search and 360-degree profile views with transaction histories, network graphs, and risk scoring summaries.

Dashboard Implementation

The dashboard page (app/dashboard/page.tsx) provides at-a-glance system status with key performance indicators, recent activity, and alert summaries. The page uses Material-UI Grid for responsive layout and Card components for visual organisation.

KPI cards display total transactions processed, flagged transaction counts, open case counts, and prevented fraud amounts. These metrics update in real-time via WebSocket connections (or periodic polling in the basic implementation).

A line chart shows transaction volume over the past 30 days, broken down by risk level (low/medium/high). A donut chart displays the distribution of risk levels, helping analysts understand current risk posture at a glance.

The recent transactions table shows the 10 most recent flagged transactions with columns for timestamp, amount, risk level, and status. Clicking a row navigates to the transaction detail page for investigation.

Dashboard Widgets

- Transaction volume KPI
- Flagged count KPI
- Open cases KPI
- Prevented fraud KPI
- Volume trend chart
- Risk distribution donut
- Recent transactions table
- Alert summary list

Transaction Explorer

The transaction explorer (app/transactions/page.tsx) provides sophisticated searching and filtering of the transaction database. Investigators use this tool to find suspicious patterns, track specific entities, and build case evidence.

Filter controls include date range pickers (start/end date), risk level multi-select, entity ID search, amount range sliders, merchant search, and flagged status toggles. These filters combine with AND logic and are applied client-side after fetching filtered server results.

The transactions table displays key attributes: timestamp, amount, sender/recipient, risk level indicator (coloured badge), and action buttons (view details, flag for investigation). The table supports sorting by any column and pagination for large result sets.

Clicking "View Details" navigates to the transaction detail page, which shows the complete transaction record including metadata, scoring history, associated entities, related cases, and audit log entries. Investigators can add notes, link to cases, or manually adjust risk levels with appropriate justification.

Case Management Interface

The case management interface (app/cases/page.tsx) enables investigators to track and manage fraud investigations through their complete lifecycle. The interface prioritises usability under high workload conditions, with fast filters and batch operations.

The case list displays all cases with columns for case ID, title, status (with coloured badges), priority, assigned investigator, creation date, and action buttons. Filters enable querying by status (OPEN, IN_PROGRESS, CLOSED, FALSE_POSITIVE), assigned investigator, priority level, and date range.

Clicking a case opens the detailed case view (app/cases/[id]/page.tsx), which shows case metadata, investigation timeline (CaseEvent list), associated transactions with risk scores, related entities with network graphs, investigator notes, and status change controls.

The case timeline presents events chronologically with timestamps, actor names, event types, and detailed descriptions. Investigators add new events via a form at the bottom, capturing interview summaries, evidence collection, external reports, and investigation findings.

Entity 360 View

The entity 360 view (app/entities/[id]/page.tsx) consolidates all information about an entity into a comprehensive profile for investigator review. The page combines static profile data, transaction history, network relationships, and risk assessment.

The entity profile section displays identifying information (name, type, registration details), contact information, KYC verification status, and current risk score with trend indicators. Tags and notes added by investigators provide qualitative context.

The transaction history table shows all transactions involving the entity (as sender or recipient), with filters for date range, amount, and risk level. Investigators analyse this history to identify suspicious patterns.

The network graph visualisation (placeholder in current implementation) would show entity relationships using a force-directed graph layout. Nodes represent entities, edges represent relationships (transaction, shared device, shared address), and node size/colour encode risk scores. Investigators use this visualisation to identify fraud rings and money laundering networks.

API Client Implementation

The frontend API client (lib/api.ts) encapsulates all backend communication using axios with interceptors for authentication and error handling. The client provides a clean, typed interface for React components to fetch data without managing HTTP details.

The axios instance is configured with the backend base URL (from environment variables), timeout settings (10 seconds), and content-type headers. Request interceptors inject JWT tokens from localStorage into Authorization headers for authenticated endpoints.

Response interceptors handle errors globally, parsing 401 (Unauthorized) responses to redirect to login, 403 (Forbidden) responses to show permission errors, and 422 (Validation Error) responses to display field-specific validation messages.

The API client exports functions for each backend endpoint: `api.transactions.list(filters)`, `api.transactions.get(id)`, `api.cases.create(payload)`, etc. These functions return typed promises that resolve to schema-validated response objects.

API Client Methods

- `auth.login(credentials)`
- `transactions.list(filters)`
- `transactions.get(id)`
- `transactions.flag(id)`
- `cases.list(filters)`
- `cases.get(id)`
- `cases.create(payload)`
- `cases.updateStatus(id, status)`
- `entities.get(id)`
- entities.network(id, depth)

Docker Containerisation

The backend Dockerfile implements a multi-stage build that produces a minimal, security-hardened production image. The build stage installs dependencies and compiles Python packages, whilst the final stage copies only necessary artifacts, significantly reducing image size and attack surface.

The build stage uses python:3.11-slim as the base, installs system dependencies (build-essential, libpq-dev for PostgreSQL), copies requirements.txt, and runs `pip install`. The final stage uses the same base image, copies installed packages from the builder, copies application code, and sets up a non-root user for running the application.

The EXPOSE directive documents that the container listens on port 8000, and the CMD directive specifies the Uvicorn command for running FastAPI with appropriate workers and host binding. Environment variables are injected at runtime via Docker Compose or Fly.io secrets.

Frontend Docker Configuration

The frontend Dockerfile also uses multi-stage builds, with stages for dependency installation, application building, and production serving. This approach optimises image size and build caching, crucial for fast CI/CD pipelines.

1

Dependencies Stage

Copies package files, runs npm install to create node_modules, and caches dependencies for subsequent builds.

2

Builder Stage

Copies source code and dependencies, runs next build to create optimised production bundles, and generates standalone server.

3

Runner Stage

Uses minimal alpine base, copies built artifacts, sets up non-root user, and configures Node.js server to serve the application.

Fly.io Deployment Configuration

Fly.io provides a modern platform-as-a-service with global edge deployment, managed PostgreSQL, and automatic SSL certificates. The fly.toml configuration files define application settings, build processes, and runtime behaviour for both backend and frontend applications.

Backend fly.toml

Specifies the application name (fraud-detection-backend), primary region (iad - US East), build configuration (Dockerfile path), and HTTP service definition (internal port 8000, public ports 80/443). Environment variables reference Fly.io secrets for sensitive configuration.

The health check configuration defines HTTP endpoints for liveness and readiness probes, enabling Fly.io to detect and restart unhealthy instances. The PostgreSQL database is provisioned separately and attached via DATABASE_URL secret.

Frontend fly.toml

Similar structure to backend but with port 3000 and Next.js-specific environment variables. The frontend references the backend URL via NEXT_PUBLIC_API_URL environment variable, configured to use the deployed backend application URL.

Static asset serving is optimised with CDN caching headers, and the standalone build output ensures minimal image size and fast cold starts.

Model Training Script

The train_model.py script generates synthetic training data, trains the autoencoder model, evaluates performance, and saves model weights for production use. This script is run periodically (monthly or quarterly) to retrain models with updated transaction patterns.

Synthetic data generation creates realistic transaction features with controlled distributions: normal transactions cluster around typical patterns, whilst synthetic fraud examples (excluded from training) show extreme deviations. The script generates 10,000-50,000 training samples.

The training loop implements early stopping based on validation loss, preventing overfitting. Training typically converges in 50-100 epochs. The script computes reconstruction error distribution on validation data to calibrate risk level thresholds (95th percentile for MEDIUM, 99th for HIGH).

Model evaluation measures false positive rate, true positive rate, and AUC-ROC on held-out test data containing both legitimate and synthetic fraud transactions. These metrics inform whether a newly trained model should replace the current production model.

Database Seeding

The seed_data.py script populates a fresh database with initial data required for application operation: user accounts, sample entities, synthetic transactions, and reference data. This script is essential for development environments and demo deployments.

User accounts are created for each role (admin, investigator, analyst) with known passwords for development. Production deployments should modify or skip this seeding to prevent known credentials.

Sample entities represent realistic individuals and businesses with varied profiles, enabling testing of entity search, network analysis, and risk scoring. The script creates 100-500 entities with randomised attributes.

Synthetic transactions are generated between entities with realistic distributions of amounts, merchants, locations, and timestamps. The script includes both normal and suspicious transaction patterns to test fraud detection.

Seeded Data

- 3 user accounts (admin/investigator/analyst)
- 100+ entities (individuals, businesses)
- 1000+ synthetic transactions
- 10+ reference merchants
- Sample entity relationships
- Initial compliance rules
- Test cases with events

Documentation Structure

README.md

Quick start guide covering installation, configuration, running locally, and basic usage examples. Includes prerequisites, dependency installation, and first-run instructions.

architecture.md

Comprehensive system architecture documentation covering component interactions, data flows, ML pipeline, API design, database schema, and deployment topology.

forensics-playbook.md

Investigator guide describing how to use the platform for fraud investigations, including case workflows, evidence collection, network analysis, and reporting.

deployment-checklist.md

Step-by-step deployment guide for production environments, covering infrastructure setup, security hardening, database migrations, secrets management, and monitoring configuration.

Testing Strategy

The testing strategy implements unit tests for core business logic (feature engineering, scoring, agents) and integration tests for API endpoints and database operations. The pytest framework provides async support and fixtures for test isolation.

Unit Tests

test_features.py validates that feature engineering produces expected outputs for known transaction patterns, handles missing data gracefully, and maintains consistent feature ranges.

test_scoring.py verifies that the scoring engine correctly loads models, computes reconstruction errors, assigns risk levels based on thresholds, and handles model failures gracefully.

test_agents.py confirms that agents execute expected actions (case creation, compliance checks, network analysis) and properly handle edge cases.

Integration Tests

test_api.py validates API endpoints with real database operations, testing authentication, authorisation, request validation, response formats, and error handling.

test_database.py verifies database models, relationships, constraints, and queries function correctly with PostgreSQL-specific features.

Tests use in-memory SQLite for speed during development but run against PostgreSQL in CI to catch database-specific issues.

Git Repository Organisation

The Git repository follows a standard structure with .gitignore files for Python and Node.js artifacts, separate backend/ and frontend/ directories, and docs/ for documentation. The root directory contains README.md, LICENSE, and CONTRIBUTING.md.

The .gitignore files exclude compiled Python bytecode (__pycache__, *.pyc), virtual environments (venv/, .env), Node modules (node_modules/), build artifacts (.next/, dist/), and IDE-specific files (.vscode/, .idea/).

The .gitkeep files ensure that empty directories (like app/models/) are tracked by Git, which otherwise ignores empty directories. This is important for maintaining project structure even before files are added.

The LICENSE file specifies MIT License, permitting commercial use, modification, and distribution with attribution. The CONTRIBUTING.md provides guidelines for extending the codebase, including code style, testing requirements, and pull request processes.

Environment Configuration

Both backend and frontend require environment-specific configuration via .env files. These files are excluded from Git (.gitignore) and must be created manually during deployment, preventing accidental exposure of secrets.

Backend .env

`DATABASE_URL`: PostgreSQL connection string
`JWT_SECRET`: Random secret for token signing
`JWT_ALGORITHM`: HS256
`ACCESS_TOKEN_EXPIRE_MINUTES`: 30
`MODEL_PATH`: models/autoencoder.pth

Frontend .env.local

`NEXT_PUBLIC_API_URL`: Backend API URL (http://localhost:8000 for local, production URL for deployed)

Deployment Checklist Overview

The deployment checklist (docs/deployment-checklist.md) provides a comprehensive step-by-step guide for deploying the fraud detection system to production. The checklist covers pre-deployment preparation, infrastructure provisioning, application deployment, security hardening, monitoring setup, and post-deployment validation.

Pre-Deployment Tasks

- Generate JWT secret with high entropy
- Provision PostgreSQL database (Fly.io or RDS)
- Configure DNS records for custom domains
- Set up SSL certificates (automatic with Fly.io)
- Create deployment user accounts
- Prepare synthetic training data or historical data
- Train initial model and validate performance

Deployment Tasks

- Deploy backend application (fly deploy)
- Run database migrations (alembic upgrade head)
- Seed initial data if needed
- Deploy frontend application
- Verify health check endpoints
- Configure environment variables/secrets
- Test end-to-end flows

Security Hardening

Production deployments require security hardening beyond the baseline configuration. The deployment checklist includes specific hardening tasks to protect against common attack vectors and satisfy regulatory requirements.

Password policies should enforce minimum length (12 characters), complexity requirements (uppercase, lowercase, numbers, symbols), and expiration intervals (90 days for privileged accounts). The bcrypt work factor should be increased to 12-14 rounds to stay ahead of improving computational capabilities.

API rate limiting prevents abuse and denial-of-service attacks. Implement per-IP and per-user rate limits using middleware or reverse proxy (Nginx, Cloudflare). Typical limits: 100 requests/minute per IP, 1000 requests/hour per authenticated user.

Database access should use least-privilege principles: the application user should only have INSERT, SELECT, UPDATE permissions on application tables, not DROP or ALTER. Database credentials should rotate quarterly and be stored in secret management systems (Fly.io secrets, AWS Secrets Manager, HashiCorp Vault).

HTTPS must be enforced for all traffic with HSTS headers and modern TLS versions (1.2+). Disable insecure ciphers and protocols. Fly.io handles this automatically with automatic Let's Encrypt certificates.

Monitoring and Alerting

Production monitoring requires comprehensive observability across application metrics, infrastructure health, and business KPIs. The system exports Prometheus metrics which feed into Grafana dashboards and AlertManager for alerting.

Application Metrics

- Request rate and latency (p50, p95, p99)
- Error rate by endpoint and status code
- Database query performance
- Model inference latency
- Scoring throughput (TPS)

Infrastructure Metrics

- CPU utilisation
- Memory usage
- Disk I/O
- Network bandwidth
- Container health status

Business Metrics

- Transaction volume
- Fraud detection rate
- False positive rate
- Case resolution time
- Prevented fraud amount

Security Metrics

- Failed login attempts
- Authentication token invalidations
- Blocked entities count
- Compliance violations
- Suspicious activity patterns

Performance Optimisation

The system is designed for high performance with sub-100ms transaction scoring targets. Several optimisation techniques ensure these targets are met under production load.

Database query optimisation includes proper indexing on frequently queried columns (transaction.timestamp, transaction.sender_id, transaction.recipient_id, transaction.risk_level), query plan analysis using EXPLAIN, and denormalisation where appropriate. Connection pooling (SQLAlchemy pool) reuses database connections across requests, avoiding connection establishment overhead.

Model inference optimisation loads model weights once at startup rather than per request, uses GPU acceleration if available (CUDA), and implements batch prediction for bulk scoring operations. Feature engineering is optimised with vectorised NumPy operations instead of Python loops.

API response caching uses HTTP ETags and Last-Modified headers for conditional requests, reducing bandwidth for unchanged resources. Redis could be added for application-level caching of frequently accessed entities and aggregate statistics.

Asynchronous processing offloads slow operations (network graph generation, bulk exports, email notifications) to background workers (Celery, AWS SQS), preventing API timeouts. The main request path remains fast whilst heavy computation occurs asynchronously.

Scalability Considerations

As transaction volume grows, the system must scale horizontally to maintain performance and reliability. The architecture supports several scaling approaches depending on bottlenecks.

Application Scaling

The FastAPI backend is stateless, enabling horizontal scaling by running multiple application instances behind a load balancer. Fly.io supports autoscaling based on CPU or request rate, automatically spinning up additional instances during peak load.

Database read replicas distribute read queries across multiple PostgreSQL instances, reducing load on the primary. The application can route read-only queries (transaction lists, entity lookups) to replicas whilst write operations go to primary.

Model inference can be offloaded to dedicated services (GPU instances, inference servers) if scoring latency becomes a bottleneck. The application calls inference APIs rather than running models in-process.

Data Scaling

Transaction data grows continuously, eventually requiring partitioning strategies. Partition tables by date (monthly partitions) to maintain query performance, and archive old data to cold storage (S3, Glacier) after retention periods expire.

Database connection pooling becomes critical at scale to prevent connection exhaustion. Configure pool sizes based on expected concurrent requests and database connection limits.

Implement database sharding if a single PostgreSQL instance cannot handle write throughput, partitioning data by entity ID or geographic region.

Operational Runbook

The operational runbook documents common operational tasks and troubleshooting procedures for production support teams. This knowledge base reduces mean time to resolution (MTTR) for incidents.

Deployment Procedure

1. Run tests locally
2. Merge to main branch
3. CI/CD pipeline builds Docker images
4. Deploy to staging, run smoke tests
5. Deploy to production with blue-green deployment
6. Monitor error rates for 30 minutes
7. Rollback if errors exceed threshold

Database Migration

1. Test migration in staging environment
2. Schedule maintenance window (low-traffic period)
3. Create database backup
4. Run `alembic upgrade head`
5. Verify migration success
6. Restart application instances
7. Monitor for errors

Model Retraining

1. Export recent transaction data (exclude fraud)
2. Run train_model.py script
3. Evaluate model performance on test set
4. Compare metrics to current production model
5. Deploy new model to staging for A/B test
6. Promote to production if performance improves
7. Archive old model version

Common Troubleshooting Scenarios

High API Latency

Symptoms: p95 latency exceeds 200ms
Diagnosis: Check database query performance, model inference time, external API calls
Resolution: Optimise slow queries, add indexes, scale horizontally, cache results

Scoring Errors

Symptoms: Transactions fail to receive scores
Diagnosis: Check model file existence, feature engineering errors, missing historical data
Resolution: Verify model path, check logs for feature calculation failures, ensure sufficient historical data

Database Connection Exhaustion

Symptoms: "Too many connections" errors
Diagnosis: Check pool size configuration, connection leaks
Resolution: Increase pool size, fix leaked connections, implement connection timeouts

Authentication Failures

Symptoms: Users report login issues
Diagnosis: Check JWT secret consistency, token expiration, user account status
Resolution: Verify JWT_SECRET matches across instances, adjust token expiration, check user.is_active

Future Enhancement Roadmap

The current implementation provides a solid foundation for fraud detection and forensic investigation. Several enhancements would further improve capabilities and usability.

Near-Term Enhancements

- Real-time WebSocket notifications for high-risk transactions
- Advanced network graph visualisation with D3.js or vis.js
- Excel/PDF report generation for case documentation
- Email alerts for case assignments and status changes
- Bulk transaction upload via CSV
- Mobile-responsive UI improvements
- Multi-factor authentication (TOTP, SMS)
- Audit log viewer with advanced filtering

Long-Term Enhancements

- Graph neural networks for entity network analysis
- Federated learning across multiple institutions
- Integration with external fraud databases (shared blacklists)
- Natural language search for transactions and entities
- Automated investigation workflow orchestration
- Real-time model retraining pipeline
- Blockchain integration for immutable audit trails
- AI-powered investigation suggestions

Compliance and Regulatory Considerations

Financial fraud detection systems must comply with various regulations depending on jurisdiction and industry. The architecture supports compliance requirements but requires configuration for specific regulatory frameworks.

GDPR (EU): Right to access, right to erasure, data minimisation, and explicit consent requirements. The system supports data export (via API), anonymisation (entity masking), and audit logging for compliance demonstration. Implement retention policies to delete old data after regulatory periods expire.

PCI DSS (Payment Cards): Requires encryption of cardholder data at rest and in transit, access logging, and regular security audits. The system uses SSL/TLS for transit encryption and should encrypt sensitive database columns (card numbers, CVV) at rest using field-level encryption.

AML/KYC (Anti-Money Laundering): Requires transaction monitoring, suspicious activity reporting (SARs), and customer due diligence. The system provides transaction monitoring and case management but requires integration with KYC providers for identity verification and risk scoring.

SOC 2: Requires security controls, audit trails, and regular penetration testing. The immutable audit log satisfies logging requirements, and the architecture document supports control documentation.

Conclusion and Next Steps

This comprehensive fraud detection forensic system provides a production-ready foundation for financial crime investigation teams. The architecture combines real-time machine learning, rule-based compliance checks, sophisticated case management, and detailed audit logging into a cohesive platform.

1. **Review the Documentation**

   Familiarise yourself with the architecture, API documentation, and forensic playbook. Understanding the system design will help you customise it for your specific requirements.

2. **Set Up Local Environment**

   Follow the quick start guide in README.md to run the system locally. Experiment with the frontend, API endpoints, and database to understand component interactions.

3. **Train Models with Your Data**

   The synthetic data is useful for testing, but production deployment requires training on your historical transaction data. Prepare a dataset of legitimate transactions and run the training script.

4. **Deploy to Production**

   Follow the deployment checklist to deploy to Fly.io or your preferred platform. Configure environment variables, run migrations, and verify system health.

5. **Customise and Extend**

   Adapt the system to your organisation's workflows, compliance requirements, and integration needs. The modular architecture supports customisation without major refactoring.

The system is ready for production deployment and provides a solid foundation for sophisticated fraud detection and forensic investigation capabilities. Reach out with questions or contributions as you extend the platform for your needs.