Trading Analytics Platform: Real-time Crypto Trading with ML-Driven Predictions

Technical Architecture Article
University Technical Project (UTM)
Team: Nick, Dan, Damian, Valentina

December 2025

Abstract

This article presents a comprehensive technical analysis of a production-grade microservices-based cryptocurrency trading platform combining real-time market data ingestion, ACID-compliant data lakehouse storage, and machine learning predictions. The system processes live Binance market data through Apache Kafka, stores time-series data in Apache Iceberg format with PostgreSQL metadata catalog, and exposes a full-stack application with WebSocket support for real-time client-server communication. The architecture implements industry-standard patterns including circuit breakers for resilience, JWT-based authentication, CSRF protection, and event sourcing for order management. This article examines core design decisions, implementation patterns, and production deployment considerations.

Contents

1 Executive Summary

The Trading Analytics Platform represents a modern, scalable approach to cryptocurrency trading platform architecture. Unlike traditional monolithic trading systems, this platform decouples business logic into eight independently deployable microservices communicating asynchronously via Apache Kafka and synchronously via REST APIs. This architectural approach enables teams to work independently on different domains while maintaining consistency through well-defined API contracts and event schemas.

Key Distinguishing Features:

- **Real-time Data Lakehouse**: Kafka-to-Iceberg pipeline ensures ACID compliance and schema evolution capabilities for time-series market data, enabling both streaming analytics and complex historical queries over the same dataset.
- **ML-driven Order Signals**: TensorFlow LSTM model with 91.8% F1 score provides predictive sell/hold signals based on technical indicators, augmenting human trading decisions with data-driven recommendations.
- **Zero Data Loss Architecture**: Transactional guarantees across Kafka, PostgreSQL, and Iceberg prevent duplicate or lost trades through idempotent operations and message retention policies.
- **Horizontal Scalability**: Kafka with 12 partitions, PostgreSQL service isolation, and stateless microservices enable linear scaling across multiple machines.
- **Event-Driven Order Management**: Event sourcing pattern provides complete audit trail and time-travel debugging capabilities, critical for regulatory compliance in financial applications.

2 System Overview

2.1 Project Context and Business Requirements

The platform originated as a university technical project but implements production-grade engineering practices typically found in fintech organizations at venture-backed startups. The project serves as an educational platform for understanding distributed systems patterns, real-time data processing, and machine learning integration in financial applications while maintaining sufficient complexity to represent real-world challenges.

The platform must accomplish several interconnected business objectives. First, it ingests real-time cryptocurrency price data from Binance, the world's largest cryptocurrency exchange, capturing millions of trades per day across multiple trading pairs. Second, it executes trading orders—including market orders that execute immediately, limit orders that wait for favorable prices, and sophisticated stop-loss and stop-limit orders that protect against adverse market movements. Third, the system provides portfolio management capabilities allowing users to track positions, view balances, and analyze historical performance. Fourth, it generates machine learning-based trading signals using historical price patterns to identify potential sell opportunities before prices decline significantly. Finally, the system maintains complete audit trails for regulatory compliance and operational debugging, essential in any financial system handling real value.

2.2 Quality Attributes and Performance Targets

The system design prioritizes specific quality attributes based on trading platform requirements. Availability targets 99.5% uptime through health checks, circuit breakers, and service isolation

preventing cascading failures. Consistency enforces strong ACID guarantees at multiple layers: PostgreSQL provides transactional consistency for accounts, Iceberg provides snapshot isolation for data lake operations, and Kafka provides message ordering within partitions. Latency requirements demand order placement completes within 500 milliseconds, achieved through Redis caching of portfolio data and optimized database query indexes. The system must sustain 10,000 trades per minute sustained throughput through Kafka partitioning and asynchronous processing. Auditability requires maintaining full trade history through event sourcing, enabling investigators to understand exactly what happened and when.

3 Microservices Architecture

3.1 Decomposition Principles

The platform decomposes functionality along business domain boundaries, a principle known as domain-driven design. This approach creates eight independent services, each owning a specific business capability and the data required to fulfill it. The API Gateway sits at the front, receiving all client requests, validating authentication, enforcing rate limits, and routing requests to appropriate backend services. The Portfolio Service manages wallet balances and asset holdings, maintaining the single source of truth for user account state. The Order Service implements sophisticated order lifecycle management, handling the state transitions from creation through execution or cancellation. The Transaction Service tracks settled transactions and provides historical records for reconciliation and reporting. The Analytics Service aggregates metrics across users and trades, providing business intelligence and operational insights. The ML Service computes technical indicators from market data and generates predictive signals. The User Service manages authentication, profiles, and identity. The Market Data Service, built with NestJS, continuously streams live prices from Binance WebSocket feeds.

3.2 Communication Patterns: Synchronous and Asynchronous

Microservices communicate through two distinct patterns depending on requirements. Synchronous REST communication occurs when immediate responses are necessary, such as when the API Gateway needs to retrieve a user's portfolio before authorizing an order placement. The API Gateway implements a circuit breaker pattern using the Opossum library, a technique borrowed from electrical systems that prevents cascading failures. When a backend service becomes unhealthy or slow, the circuit breaker stops forwarding requests, instead returning cached responses or error messages. This allows struggling services to recover without becoming overwhelmed by traffic from retry attempts. After a service exhibits stable health for a timeout period, the circuit breaker gradually reopens, resuming normal traffic flow.

Asynchronous communication through Apache Kafka enables loose coupling between services. When an order executes, the Order Service publishes an order event to Kafka rather than directly calling the Portfolio Service to update balances. The Portfolio Service subscribes to order events and processes them at its own pace, updating account state. This decoupling has profound implications: services can be restarted or temporarily unavailable without affecting others, multiple services can independently process the same business event, and the system can replay events to debug issues or migrate data. Event sourcing—recording a complete history of what happened—enables forensic analysis and regulatory compliance documentation.

4 Data Architecture: The Lakehouse Pattern

4.1 The Lakehouse Concept

Modern data systems increasingly adopt the lakehouse architecture, combining the flexibility of data lakes with the structure and guarantees of data warehouses. Data lakes store raw data in economical object storage like Amazon S3, maintaining schema flexibility and supporting diverse data formats. However, they traditionally lack structure and consistency guarantees, making complex analytics problematic. Data warehouses enforce strict schemas and provide SQL query capabilities, but require expensive compute resources and rigid schema evolution. The lakehouse pattern layers data warehouse capabilities—ACID transactions, schema enforcement, SQL queries—over lake-style storage infrastructure.

The Trading Analytics Platform implements this pattern through Apache Iceberg, an open-source table format that brings database-like semantics to object storage. Iceberg maintains a metadata layer describing table schema, partitions, and snapshots. Stored in PostgreSQL, this metadata enables time travel—querying historical table versions—and schema evolution, where new columns can be added without rewriting existing data. The actual data lives in Parquet format on MinIO S3, providing columnar storage enabling efficient analytical queries and dramatic compression.

4.2 Data Pipeline: Binance to Analytics

The data pipeline follows a classic extract-transform-load pattern optimized for streaming data. Real-time ingestion begins at Binance WebSocket API, streaming live trades for ten cryptocurrency pairs. As trades occur globally on Binance's matching engine, the system receives tick-by-tick updates including timestamp, symbol, trade side, price, and volume. These raw events flow into Apache Kafka, a distributed message broker that decouples producers from consumers.

Kafka distributes the incoming trade stream across twelve partitions based on currency symbol as the partition key. This design ensures all trades for a given cryptocurrency stay ordered—Bitcoin trades maintain their sequence, Ethereum trades maintain their sequence—while allowing different trading pairs to be processed in parallel by separate consumer instances. Kafka retains messages for twenty-four hours, enabling replay from recent history if processing fails or becomes corrupted, a critical reliability feature for financial systems.

The Kafka Consumer pulls trade events from the broker and applies transformations. Data arriving from different sources may have different formats—legacy systems send trades with different field names, different timestamp formats, or different measurement units. The transformer normalizes these variations into a canonical internal representation. It parses timestamps, converts between timezone formats, validates that prices and volumes are numeric and positive, and enriches records with processing metadata like when the system processed the event versus when the trade actually occurred.

Once transformed, records are batched—the consumer accumulates 1,000 records or waits up to ten seconds—before writing to Iceberg. Batching improves efficiency, as each write operation carries overhead, and batches compress better than single records. The write operation itself uses Iceberg's atomic append semantics, ensuring that if multiple consumer instances write concurrently to the same partition, conflicts are resolved through exponential backoff retry logic rather than data loss or corruption.

The Iceberg table schema prescribes nine columns: symbol, timestamp, price, volume, trade side, trade identifier, data source, processing time, and date partition. Day-level partitioning means all trades for a single day live in one partition, enabling efficient queries like "show me all trades from yesterday" without scanning the entire dataset. Parquet columnar storage with ZSTD compression achieves 33x reduction—raw CSV of 1GB per day becomes 30MB in the lakehouse, enabling cost-effective storage of years of historical data.

5 Application Tier: Core Business Logic

5.1 Technology Stack Rationale

The platform selects technologies based on specific requirements. Node.js with Express dominates the microservices tier due to rapid development velocity, extensive package ecosystem, and excellent real-time capabilities through Socket.IO. NestJS provides the Market Data Service with an enterprise framework layering TypeScript, dependency injection, and structured module organization over Express. PostgreSQL 15 provides ACID transactional guarantees essential for financial consistency—users cannot accidentally double-spend balances or lose trades. Redis caches frequently accessed data like current portfolio balances and real-time prices, reducing database load and ensuring portfolio queries complete within 50 milliseconds even under high load. Docker containerizes each service for reproducible deployments across development and production environments.

5.2 API Gateway: Entry Point and Orchestration

The API Gateway implements the Backend-for-Frontend pattern, providing a unified API tailored to the web application's needs. All client requests flow through the gateway, which validates authentication through JWT tokens, enforces rate limiting to prevent abuse, checks CSRF tokens on state-changing operations, and routes requests to appropriate backend services. The gateway maintains an in-memory cache of authentication credentials and service health status, enabling rapid decisions without querying databases on every request.

The authentication system uses JWT (JSON Web Tokens) with RS256 cryptographic signatures. Users authenticate by providing email and password; the User Service validates credentials against bcrypt password hashes stored in the database. Upon successful authentication, the system generates a JWT token containing the user identifier, email, and token expiry time, signed with a private key. The JWT is returned to the client, which includes it in subsequent requests via the Authorization header. The API Gateway verifies the JWT signature using the public key, confirming the token originated from the system and hasn't been tampered with. Tokens expire after 24 hours, forcing users to re-authenticate, limiting the window of time a compromised token remains valid.

CSRF protection prevents attackers from forging cross-site requests. When a user loads the trading dashboard, the server generates a unique CSRF token tied to that session and returns it in an HttpOnly cookie. When the user places an order, the browser sends both the cookie (automatically by the browser) and the token in the request body. The API Gateway compares these two values; if they don't match, the request is rejected. This prevents attackers on other websites from forging valid requests, since they can read neither the HttpOnly cookie nor generate valid CSRF tokens for legitimate sessions.

Rate limiting protects against brute force attacks and denial of service. Each IP address is limited to 100 requests per 15-minute window. After exceeding this limit, the API Gateway returns an error and requests are rejected until the window resets. This makes systematic attacks prohibitively expensive—an attacker would need to distribute requests across many IP addresses or wait for rate limit windows to expire.

5.3 Portfolio Service: State Management

The Portfolio Service maintains the single source of truth for user account state. Each user has a portfolio, which is a collection of holdings in different cryptocurrency assets. Holdings track the quantity of each asset owned and the cost basis at which it was acquired. Separately, the service maintains balance records for each asset, tracking total quantity, available quantity (not reserved for pending orders), and held quantity (reserved by pending orders). A critical invariant

enforces that available plus held equals total—this database constraint prevents programming errors from creating or destroying assets.

The Portfolio Service stores no business logic—it simply exposes CRUD operations for creating portfolios, adding holdings, and updating balances. When an order executes, the Order Service publishes an order event to Kafka; the Portfolio Service consumes this event and updates holdings and balances accordingly. This event-driven design enables the Portfolio Service to handle order execution asynchronously, and multiple independent services can consume order events for their own purposes—perhaps a Notifications Service sends confirmation emails, a Reporting Service updates analytics dashboards, a Fraud Detection Service analyzes for suspicious patterns.

5.4 Order Service: Complex State Machines

The Order Service implements sophisticated trading logic. An order begins in pending state after creation. The system validates that the user's portfolio has sufficient available balance to cover the order—if not, it is rejected. For market orders, execution happens immediately at current market price. For limit orders, execution waits until market price reaches the specified price or better. For stop-loss orders, a trigger price is monitored, and when market price falls below the trigger, a market order executes. Stop-limit orders combine these behaviors: when the stop price is breached, a limit order is placed rather than a market order.

As orders execute, they transition through states: open (waiting for execution), partially filled (some quantity executed, remainder waiting), filled (completely executed), or canceled. For each state transition, an event is published to Kafka. The event contains complete order details, execution prices, filled quantities, and timestamps, creating a durable log of what happened. This event log enables later investigation—a user disputes an order, the system can replay events to reproduce the sequence of actions and timing.

A critical responsibility of the Order Service is preventing double-spending. If a user places two orders, each attempting to sell 1 Bitcoin, but the portfolio contains only 1 Bitcoin total, at most one order should succeed. The service achieves this through balance reservation: when an order is created, available balance is decremented and held balance is incremented. When orders execute, held balance is released and holdings are transferred. When orders are canceled, held balance is released back to available. Database constraints ensure balance arithmetic remains consistent.

6 Machine Learning: Predictive Trading Signals

6.1 Model Architecture and Design

The machine learning component trains a TensorFlow LSTM model to recognize patterns predictive of favorable selling opportunities. LSTM (Long Short-Term Memory) networks excel at recognizing patterns in sequential data like time series. The model processes a 15-day sliding window of historical daily price data, analyzing how prices, trends, and volatility evolved over two weeks, to predict whether the next day will be a good selling opportunity.

Model performance metrics demonstrate practical utility: 91.8% F1 score, 0.969 AUC-ROC, 89.2% precision, 94.5% recall. The F1 score balances precision (of predicted sells, how many actually occur) and recall (of actual sells, how many does the model catch). The AUC-ROC score measures how well the model distinguishes between positive and negative cases across all classification thresholds. These metrics suggest the model successfully identifies meaningful patterns in price data—performance far better than random coin flips but acknowledging imperfect prediction.

6.2 Feature Engineering: Technical Indicators

The ML Service computes twelve technical indicators from open, high, low, close, and volume (OHLCV) data. Simple Moving Averages over 20, 50, and 200 days smooth price data and identify trends—when short-term averages trade above long-term averages, momentum is upward; when below, momentum is downward. Bollinger Bands create a dynamically-adjusted trading range around a moving average; when price approaches the upper band, the asset may be overbought; when approaching the lower band, oversold. MACD (Moving Average Convergence Divergence) measures momentum by comparing fast and slow exponential moving averages; when MACD crosses above its signal line, momentum is strengthening; crossings below suggest weakening momentum. Volatility measures price uncertainty—is price drifting slowly or swinging wildly? High volatility may precede trend reversals.

A critical feature captures whether price has fallen below all three key moving averages—a rare but significant event suggesting breakdown of support levels and potential continuation of downtrends. These features capture different aspects of price dynamics: trend following, mean reversion, momentum, and volatility. Historical backtesting identified these indicators as having predictive power for sell opportunities.

6.3 Production Inference Pipeline

During operation, the ML Service maintains a 15-day rolling window of daily features for each tracked cryptocurrency. When new daily OHLCV data arrives, the service computes all 12 technical indicators and appends them to the window. Once the window contains 15 complete days, it becomes ready for prediction. The model processes the window and outputs a probability between 0 and 1 indicating the likelihood that tomorrow will present a favorable selling opportunity.

The system converts this probability to actionable decisions using a decision threshold optimized during training. The threshold 0.853 was selected to maximize F1 score on validation data. Predictions with probability above this threshold are classified as SELL signals; below, HOLD signals. The distance from this threshold determines confidence—predictions far from 0.5 are marked STRONG, moderate distances marked MODERATE, and predictions near 0.5 marked WEAK. This confidence scoring helps users prioritize signals: a STRONG SELL signal deserves more attention than a WEAK SELL.

Frontend applications display these signals to traders, who use them as decision inputs rather than automatic trading rules. A trader might see "Strong Sell signal for Bitcoin" and combine this with their own analysis, recent news, and broader market sentiment before deciding whether to sell. This human-in-the-loop design balances algorithmic insights with human judgment.

7 Frontend Application

7.1 User Interface Architecture

The frontend is a Next.js 16 application built with React 19, providing a responsive web interface for trading. The application renders a dashboard displaying current portfolio holdings with real-time market values, market data showing live prices for tracked cryptocurrency pairs, and an order placement interface supporting all order types. TailwindCSS provides responsive styling enabling the application to adapt from mobile phones to desktop monitors.

Real-time updates flow through WebSocket connections established between the browser and API Gateway. When market prices update, the server pushes updates through WebSocket to all connected clients, enabling charts to animate in real-time without constant HTTP polling. Orders placed by users are transmitted to the API Gateway via WebSocket, receiving instant

confirmation or error feedback. This real-time bidirectional communication enhances user experience and enables responsive user interfaces.

8 Infrastructure and Deployment

8.1 Containerization and Orchestration

Each microservice is packaged as a Docker container following layered image principles: base image (Node.js or Python), dependencies installation, source code copying, and container startup command. This approach provides reproducible deployments—the exact same image runs identically in development, testing, and production. Container registries store images, enabling rapid deployment across multiple machines or cloud providers.

Docker Compose defines the entire application as a single declarative configuration. Fifteen services are defined: Zookeeper and Kafka provide message streaming, Redis caches data, PostgreSQL stores relational data, MinIO provides S3-compatible object storage, Kafka Producer and Consumer implement the data pipeline, eight microservices provide application logic, and frontend and Kafka UI provide interfaces. Dependencies are declared—services wait for their prerequisites to start in proper order—and health checks verify services are actually running correctly before dependent services begin.

8.2 Resource Management

Each service declares memory limits and reservations. Kafka is allocated 8GB reflecting its role as an event broker handling 12 parallel partitions. The Kafka Consumer is allocated 2GB for batch processing and in-memory model inference. PostgreSQL receives 4GB for index caching and connection pooling. Redis receives 4GB enabling in-memory caching of substantial datasets with LRU eviction. Microservices are allocated 512MB, sufficient for minimal operation while encouraging stateless design and horizontal scaling. These limits prevent one service from starving others or exhausting host machine resources.

8.3 Monitoring and Observability

Kafka UI provides visibility into topics, partitions, consumer group lag, and message throughput. Lag indicates how far behind consumers are relative to producers—high lag suggests processing cannot keep up with ingestion. MinIO Console displays data lake statistics, showing how much data has been ingested and stored, enabling capacity planning. PostgreSQL logs provide query analysis and slow query identification, exposing database performance bottlenecks.

Application health endpoints return service status in JSON format. The API Gateway health endpoint includes circuit breaker status for downstream services, indicating which backends are healthy or struggling. These health checks inform orchestration systems about which services can receive traffic.

9 Security Architecture

9.1 Authentication and Authorization

JWT-based authentication verifies user identity without maintaining server-side sessions. Password storage uses bcrypt with sufficient computational cost to make brute force attacks impractical. Authorization checks enforce that users can only access their own data—an order request includes user authentication context, verified before returning sensitive balance information.

9.2 Defense in Depth

Multiple security layers provide defense against common attacks. SQL injection is prevented through parameterized queries, ensuring user input is never directly interpolated into SQL commands. Cross-site scripting is prevented through output encoding, converting dangerous characters to safe representations. Rate limiting makes brute force and denial of service attacks economically infeasible. CORS configuration restricts requests to known frontend domains.

10 Production Lessons and Insights

10.1 Architectural Successes

The microservices decomposition enabled a four-person team to work simultaneously on different domains without constant coordination. Services could be built, tested, and deployed independently. Kafka buffering prevented the API Gateway from becoming a bottleneck—if order processing temporarily slowed, Kafka accumulated orders, and processing resumed when capacity returned. Iceberg's ACID guarantees prevented data corruption when multiple processes accessed the lakehouse concurrently. Circuit breaker patterns protected against cascading failures—when one service degraded, others continued functioning and automatically stopped sending requests to the failing service.

10.2 Technical Challenges Encountered

Distributed transactions presented challenges. An order execution requires atomically updating both the order record and portfolio balances. Coordinating this across services without distributed transaction support requires careful design—the Order Service places an order, publishes an event, and the Portfolio Service later consumes it. If the Portfolio Service crashes before consuming the event, eventually it recovers and processes the event. This eventual consistency model differs from traditional ACID transactions, requiring careful design to prevent inconsistent intermediate states visible to clients.

Model deployment across Python and Node.js boundaries required careful serialization. TensorFlow models are trained and saved in Python but must be invoked from Node.js services. The solution uses a dedicated Python ML Service that other services query via HTTP, encapsulating Python dependencies in a single service rather than requiring Python knowledge throughout the codebase.

10.3 Recommended Future Improvements

Kafka Streams would enable complex real-time computations over the event stream without separate consumer applications. Service meshes like Istio would provide advanced traffic management, observability, and security policies without modifying individual services. PostgreSQL read replicas would scale analytical queries without burdening the primary database. Database sharding by user ID would enable unlimited horizontal scaling of storage capacity. Distributed tracing with OpenTelemetry would provide visibility into request flows through complex microservice chains.

11 Conclusion

The Trading Analytics Platform demonstrates applied distributed systems engineering for cryptocurrency trading. By combining microservices architecture, real-time data streaming, machine learning predictions, and ACID-compliant storage, the platform handles concurrent trading operations while maintaining consistency guarantees crucial in financial systems.

Core architectural decisions—Kafka for event distribution, Iceberg for data lakehouse capabilities, JWT for authentication, and circuit breakers for resilience—provide a scalable foundation suitable for production deployment. The machine learning component adds predictive capability, transforming raw market data into actionable trading signals. Event sourcing enables complete audit trails essential for regulatory compliance and operational debugging.

The platform succeeds as both an educational resource and functional prototype. As a teaching tool, it demonstrates how modern distributed systems principles apply to real-world problems. As a prototype, it establishes proof of concept that complex financial systems can be built on open-source components. Production deployment would require enhancements in observability, sophisticated operations procedures, and disaster recovery mechanisms, but the architectural foundation provides a solid basis for such hardening.

References

[1] Richardson, C. (2019). Microservices Patterns: With examples in Java. Manning Publications.

[2] Kleppmann, M. (2017). Designing Data-Intensive Applications. O'Reilly Media.

[3] Apache Iceberg Project. (2023). PyIceberg: Python API for Apache Iceberg. Retrieved from iceberg.apache.org

[4] Confluent Inc. (2023). Apache Kafka: The Event Streaming Platform. Retrieved from kafka.apache.org

[5] Chollet, F. (2021). Deep Learning with Python (2nd ed.). Manning Publications.

[6] Newman, S. (2015). Building Microservices. O'Reilly Media.