Trading Engine Backend Service - Comprehensive Development Plan

Executive Summary

This document outlines the comprehensive development plan for creating a unified Trading Engine Backend Service for the Alphintra platform. The plan consolidates the existing trading-api and strategy-engine services into a single, high-performance, scalable trading engine that provides advanced trading capabilities, real-time execution, and comprehensive market data management.

1. Current State Analysis

Existing Services Assessment

Trading API Service

- **Current Status**: Functional but limited
- **Strengths**:
  - Basic trading operations (buy/sell orders)
  - Portfolio management
  - Integration with multiple exchanges
  - Basic risk management
- **Limitations**:
  - Limited order types (market, limit, stop)
  - Basic strategy execution
  - No advanced order management
  - Limited real-time capabilities

Strategy Engine Service

- **Current Status**: Advanced but standalone
- **Strengths**:
  - Multiple strategy types (DCA, Grid, Momentum, ML-based)
  - Advanced technical indicators
  - Machine learning integration
  - Real-time signal generation
- **Limitations**:
  - Separate from trading execution
  - Limited integration with order management
  - No advanced risk management
  - Complex deployment and maintenance

Integration Challenges

- **Service Fragmentation**: Multiple services with overlapping responsibilities
- **Data Consistency**: Synchronization issues between services
- **Performance**: Network latency between services
- **Maintenance**: Complex deployment and monitoring
- **Scalability**: Difficult to scale individual components

2. Unified Trading Engine Architecture

Core Design Philosophy

- **Single Responsibility**: One service for all trading operations
- **High Performance**: Sub-millisecond latency for critical operations
- **Scalability**: Horizontal scaling with load balancing
- **Reliability**: 99.9% uptime with fault tolerance
- **Extensibility**: Plugin architecture for new features

System Architecture Overview

Technology Stack

Core Technologies

- **Language**: Python 3.11+ (for performance and ML ecosystem)
- **Framework**: FastAPI (async/await, high performance)
- **Database**: PostgreSQL (transactional) + TimescaleDB (time-series)
- **Cache**: Redis (high-performance caching and pub/sub)
- **Message Queue**: Apache Kafka (event streaming)

- **ML Framework**: MLflow + scikit-learn + TensorFlow

Infrastructure

- **Containerization**: Docker with multi-stage builds
- **Orchestration**: Kubernetes with auto-scaling
- **Service Mesh**: Istio for traffic management
- **Monitoring**: Prometheus + Grafana + Jaeger
- **Logging**: ELK Stack (Elasticsearch, Logstash, Kibana)

3. Detailed Feature Specifications

3.1 Advanced Order Management System

Order Types

- **Market Orders**: Immediate execution at best available price
- **Limit Orders**: Execution at specified price or better
- **Stop Orders**: Trigger market order when price reaches stop level
- **Stop-Limit Orders**: Trigger limit order when price reaches stop level
- **Iceberg Orders**: Large orders split into smaller visible portions
- **TWAP Orders**: Time-weighted average price execution
- **VWAP Orders**: Volume-weighted average price execution
- **Fill-or-Kill (FOK)**: Execute immediately in full or cancel
- **Immediate-or-Cancel (IOC)**: Execute immediately, cancel remainder
- **Good-Till-Canceled (GTC)**: Active until manually canceled
- **Good-Till-Date (GTD)**: Active until specified date
- **Trailing Stop**: Dynamic stop that follows price movement

Order Execution Features

- **Smart Order Routing**: Best execution across multiple venues
- **Order Slicing**: Large orders split for optimal execution
- **Dark Pool Access**: Hidden liquidity for institutional orders
- **Algorithmic Execution**: Pre-built execution algorithms
- **Latency Optimization**: Sub-millisecond order processing
- **Market Making**: Automated bid/ask spread management

3.2 Enhanced Strategy Engine

Strategy Types

- **Trend Following**: Moving average crossovers, momentum
- **Mean Reversion**: Statistical arbitrage, pairs trading
- **Arbitrage**: Cross-exchange, triangular arbitrage
- **Market Making**: Automated liquidity provision
- **Grid Trading**: Dynamic buy/sell levels
- **DCA (Dollar Cost Averaging)**: Systematic periodic investment
- **ML-Based**: Machine learning driven strategies
- **Options Strategies**: Covered calls, protective puts, straddles
- **Multi-Asset**: Portfolio-level strategy execution

Strategy Features

- **Real-time Signal Generation**: Sub-second signal processing
- **Backtesting Engine**: Historical performance analysis
- **Paper Trading**: Risk-free strategy testing
- **Strategy Optimization**: Parameter tuning and A/B testing
- **Portfolio-level Strategies**: Multi-asset, multi-strategy execution
- **Dynamic Risk Management**: Real-time risk adjustment
- **Performance Attribution**: Detailed performance analysis

3.3 Advanced Risk Management

Risk Controls

- **Position Limits**: Maximum position size per asset/strategy
- **Leverage Limits**: Maximum leverage per account/strategy
- **Daily Loss Limits**: Maximum daily drawdown limits
- **Concentration Limits**: Maximum exposure per sector/asset class
- **Correlation Limits**: Maximum correlated position exposure
- **Volatility Limits**: Maximum volatility exposure
- **Liquidity Limits**: Minimum liquidity requirements

Risk Monitoring

- **Real-time Risk Metrics**: VaR, Expected Shortfall, Beta
- **Stress Testing**: Scenario analysis and stress testing
- **Risk Alerts**: Automated risk threshold alerts
- **Compliance Monitoring**: Regulatory compliance tracking
- **Risk Reports**: Comprehensive risk reporting
- **Circuit Breakers**: Automatic trading halts on risk events

3.4 Advanced Market Data Management

Data Sources

- **Real-time Feeds**: Level 1 and Level 2 market data
- **Historical Data**: Long-term historical price data
- **Alternative Data**: News, social media sentiment, economic indicators
- **Fundamental Data**: Financial statements, earnings, ratios
- **Options Data**: Greeks, implied volatility, option chains
- **Crypto Data**: On-chain metrics, DeFi protocols

Data Processing

- **Data Normalization**: Standardized data formats
- **Data Validation**: Quality checks and error handling
- **Data Enrichment**: Additional metadata and calculations
- **Real-time Processing**: Stream processing for live data
- **Data Storage**: Optimized storage for time-series data
- **Data APIs**: RESTful and WebSocket APIs for data access

3.5 Machine Learning Integration

ML Pipeline

- **Feature Engineering**: Automated feature extraction
- **Model Training**: Automated model training and validation
- **Model Deployment**: Real-time model serving
- **Model Monitoring**: Performance monitoring and drift detection
- **A/B Testing**: Model comparison and selection
- **AutoML**: Automated machine learning workflows

ML Models

- **Price Prediction**: Time series forecasting models
- **Sentiment Analysis**: News and social media analysis
- **Risk Modeling**: Credit risk, market risk models
- **Optimization**: Portfolio optimization algorithms
- **Anomaly Detection**: Fraud and market anomaly detection
- **Reinforcement Learning**: Adaptive trading strategies

4. Development Roadmap

Phase 1: Core Infrastructure (Weeks 1-4)

Objective: Establish the foundation architecture and core services

Week 1-2: Service Architecture Setup

- **Service Structure**: Create unified service architecture
- **Database Design**: Design PostgreSQL and TimescaleDB schemas
- **API Framework**: Set up FastAPI with async support
- **Authentication**: Implement JWT-based authentication
- **Configuration Management**: Environment-based configuration
- **Logging & Monitoring**: Basic logging and health checks

Week 3-4: Core Data Models

- **Order Models**: Order, Trade, Position data models
- **Portfolio Models**: Account, Balance, Performance models
- **Market Data Models**: OHLCV, Trade, OrderBook models
- **Strategy Models**: Strategy, Signal, Execution models
- **Risk Models**: RiskMetrics, Limits, Alerts models
- **Database Migration**: Initial schema creation and migration

Phase 2: Basic Trading Operations (Weeks 5-8)

Objective: Implement basic trading functionality

Week 5-6: Order Management

- **Order Creation**: API endpoints for order creation
- **Order Validation**: Business logic validation

- **Order Persistence**: Database storage and retrieval
- **Order Status**: Status tracking and updates
- **Order Cancellation**: Order modification and cancellation
- **Order History**: Historical order tracking

Week 7-8: Trade Execution

- **Execution Engine**: Basic trade execution logic
- **Broker Integration**: Connection to exchange APIs
- **Order Matching**: Simple order matching algorithm
- **Trade Settlement**: Post-trade processing
- **Position Management**: Position tracking and updates
- **Error Handling**: Comprehensive error handling

Phase 3: Advanced Order Types (Weeks 9-12)

Objective: Implement advanced order types and execution algorithms

Week 9-10: Complex Order Types

- **Stop Orders**: Stop-loss and stop-limit orders
- **Conditional Orders**: If-then order logic
- **Time-based Orders**: GTD, GTC order handling
- **Iceberg Orders**: Large order slicing
- **Trailing Orders**: Dynamic stop adjustments
- **Order Combinations**: OCO, Bracket orders

Week 11-12: Execution Algorithms

- **TWAP Algorithm**: Time-weighted average price execution
- **VWAP Algorithm**: Volume-weighted average price execution
- **Implementation Shortfall**: Optimal execution timing
- **Smart Order Routing**: Best execution across venues
- **Market Making**: Automated bid/ask management
- **Latency Optimization**: Performance optimization

Phase 4: Strategy Engine Integration (Weeks 13-16)

Objective: Integrate advanced strategy capabilities

Week 13-14: Strategy Framework

- **Strategy Base Classes**: Abstract strategy framework
- **Strategy Registration**: Dynamic strategy loading
- **Strategy Execution**: Real-time strategy execution
- **Signal Processing**: Strategy signal handling
- **Strategy Monitoring**: Performance and health monitoring
- **Strategy Configuration**: Dynamic parameter management

Week 15-16: Built-in Strategies

- **Trend Following**: Moving average strategies
- **Mean Reversion**: Statistical arbitrage strategies
- **Grid Trading**: Dynamic grid strategies
- **DCA Strategies**: Dollar cost averaging
- **Momentum Strategies**: Technical momentum trading
- **Arbitrage Strategies**: Cross-exchange arbitrage

Phase 5: Risk Management System (Weeks 17-20)

Objective: Implement comprehensive risk management

Week 17-18: Risk Controls

- **Position Limits**: Real-time position monitoring
- **Loss Limits**: Daily and total loss limits
- **Leverage Controls**: Maximum leverage enforcement
- **Concentration Limits**: Sector and asset concentration
- **Risk Calculations**: VaR, Expected Shortfall calculations
- **Risk Alerts**: Automated risk notifications

Week 19-20: Advanced Risk Features

- **Stress Testing**: Scenario analysis framework
- **Correlation Monitoring**: Cross-asset correlation tracking
- **Liquidity Risk**: Liquidity monitoring and controls
- **Compliance**: Regulatory compliance monitoring
- **Risk Reports**: Comprehensive risk reporting
- **Circuit Breakers**: Automatic trading halts

Phase 6: Market Data Enhancement (Weeks 21-24)

Objective: Implement advanced market data capabilities

Week 21-22: Real-time Data

- **WebSocket Connections**: Real-time market data feeds
- **Data Normalization**: Standardized data processing
- **Data Validation**: Quality checks and error handling
- **Data Distribution**: Pub/sub data distribution
- **Data Caching**: High-performance data caching
- **Data APIs**: RESTful and WebSocket APIs

Week 23-24: Historical Data

- **Data Storage**: Optimized time-series storage
- **Data Retrieval**: High-performance data queries
- **Data Aggregation**: OHLCV aggregation
- **Data Backfill**: Historical data population
- **Data Export**: Data export capabilities
- **Data Analytics**: Statistical analysis tools

Phase 7: Machine Learning Integration (Weeks 25-28)

Objective: Integrate ML capabilities for advanced trading

Week 25-26: ML Infrastructure

- **MLflow Integration**: Model lifecycle management
- **Feature Store**: Feature engineering and storage
- **Model Training**: Automated training pipelines
- **Model Serving**: Real-time model serving
- **Model Monitoring**: Performance monitoring
- **A/B Testing**: Model comparison framework

Week 27-28: ML Models

- **Price Prediction**: Time series forecasting
- **Sentiment Analysis**: News and social media analysis
- **Risk Modeling**: Advanced risk models
- **Portfolio Optimization**: ML-based optimization
- **Anomaly Detection**: Market anomaly detection
- **Reinforcement Learning**: Adaptive strategies

Phase 8: Performance & Scalability (Weeks 29-32)

Objective: Optimize performance and scalability

Week 29-30: Performance Optimization

- **Code Optimization**: Critical path optimization
- **Database Optimization**: Query and index optimization
- **Caching Strategy**: Multi-layer caching
- **Connection Pooling**: Database connection optimization
- **Async Processing**: Asynchronous task processing
- **Memory Management**: Memory usage optimization

Week 31-32: Scalability & Reliability

- **Horizontal Scaling**: Multi-instance deployment
- **Load Balancing**: Traffic distribution
- **Fault Tolerance**: Error recovery and resilience
- **Data Replication**: Database replication
- **Monitoring & Alerting**: Comprehensive monitoring
- **Disaster Recovery**: Backup and recovery procedures

Phase 9: Security & Compliance (Weeks 33-36)

Objective: Implement security and compliance features

Week 33-34: Security

- **Authentication**: Multi-factor authentication
- **Authorization**: Role-based access control

- **Encryption**: Data encryption at rest and in transit
- **Security Monitoring**: Security event monitoring
- **Vulnerability Management**: Security scanning and patching
- **Audit Logging**: Comprehensive audit trails

Week 35-36: Compliance

- **Regulatory Compliance**: Financial regulations compliance
- **Trade Reporting**: Automated trade reporting
- **Record Keeping**: Comprehensive record retention
- **Compliance Monitoring**: Real-time compliance checks
- **Regulatory Reports**: Automated regulatory reporting
- **Audit Support**: Audit trail and documentation

Phase 10: Testing & Deployment (Weeks 37-40)

Objective: Comprehensive testing and production deployment

Week 37-38: Testing

- **Unit Testing**: Comprehensive unit test coverage
- **Integration Testing**: End-to-end integration tests
- **Performance Testing**: Load and stress testing
- **Security Testing**: Security vulnerability testing
- **User Acceptance Testing**: Business user testing
- **Regression Testing**: Automated regression testing

Week 39-40: Deployment

- **Production Deployment**: Kubernetes deployment
- **Blue-Green Deployment**: Zero-downtime deployment
- **Monitoring Setup**: Production monitoring
- **Performance Tuning**: Production optimization
- **Documentation**: Comprehensive documentation
- **Training**: User and developer training

5. API Design Specifications

5.1 Trading API Endpoints

Order Management

```
POST   /api/v1/orders                    # Create new order
GET    /api/v1/orders                    # Get orders (with filters)
GET    /api/v1/orders/{order_id}         # Get specific order
PUT    /api/v1/orders/{order_id}         # Modify order
DELETE /api/v1/orders/{order_id}         # Cancel order
POST   /api/v1/orders/{order_id}/cancel  # Cancel order (alternative)
```

Trade Management

```
GET    /api/v1/trades                    # Get trades (with filters)
GET    /api/v1/trades/{trade_id}         # Get specific trade
GET    /api/v1/trades/summary            # Get trade summary
```

Portfolio Management

```
GET    /api/v1/portfolio                 # Get portfolio summary
GET    /api/v1/portfolio/positions       # Get positions
GET    /api/v1/portfolio/balances        # Get balances
GET    /api/v1/portfolio/performance     # Get performance metrics
```

Strategy Management

```
POST   /api/v1/strategies                # Create strategy
GET    /api/v1/strategies                # Get strategies
GET    /api/v1/strategies/{strategy_id}  # Get specific strategy
PUT    /api/v1/strategies/{strategy_id}  # Update strategy
DELETE /api/v1/strategies/{strategy_id}  # Delete strategy
POST   /api/v1/strategies/{strategy_id}/start    # Start strategy
POST   /api/v1/strategies/{strategy_id}/stop     # Stop strategy
GET    /api/v1/strategies/{strategy_id}/signals  # Get strategy signals
```

Market Data

```
GET    /api/v1/market-data/symbols       # Get available symbols
GET    /api/v1/market-data/ticker        # Get ticker data
GET    /api/v1/market-data/ohlcv         # Get OHLCV data
GET    /api/v1/market-data/trades        # Get recent trades
GET    /api/v1/market-data/orderbook     # Get order book
```

Risk Management

```
GET    /api/v1/risk/limits               # Get risk limits
PUT    /api/v1/risk/limits               # Update risk limits
GET    /api/v1/risk/metrics              # Get risk metrics
GET    /api/v1/risk/alerts               # Get risk alerts
```

5.2 WebSocket API Endpoints

Real-time Market Data

```
ws://host/ws/market-data/ticker/{symbol}      # Real-time ticker
ws://host/ws/market-data/trades/{symbol}      # Real-time trades
ws://host/ws/market-data/orderbook/{symbol}   # Real-time order book
```

Real-time Trading Updates

```
ws://host/ws/orders/updates                   # Order status updates
ws://host/ws/trades/updates                   # Trade updates
ws://host/ws/portfolio/updates                # Portfolio updates
```

Real-time Strategy Updates

```
ws://host/ws/strategies/{strategy_id}/signals # Strategy signals
ws://host/ws/strategies/{strategy_id}/status  # Strategy status
```

5.3 Data Models

Order Model

```python
class Order(BaseModel):
    id: str
    user_id: int
    symbol: str
    side: OrderSide  # BUY, SELL
    order_type: OrderType  # MARKET, LIMIT, STOP, etc.
    quantity: Decimal
    price: Optional[Decimal]
    stop_price: Optional[Decimal]
    time_in_force: TimeInForce  # GTC, IOC, FOK, etc.
    status: OrderStatus  # PENDING, FILLED, CANCELED, etc.
    filled_quantity: Decimal
    average_price: Optional[Decimal]
    created_at: datetime
    updated_at: datetime
    expires_at: Optional[datetime]
```

Trade Model

```python
class Trade(BaseModel):
    id: str
    order_id: str
    symbol: str
    side: OrderSide
    quantity: Decimal
    price: Decimal
    fee: Decimal
    timestamp: datetime
    trade_type: TradeType  # MAKER, TAKER
```

Position Model

```python
class Position(BaseModel):
    user_id: int
    symbol: str
    quantity: Decimal
    average_price: Decimal
    market_value: Decimal
    unrealized_pnl: Decimal
    realized_pnl: Decimal
    updated_at: datetime
```

Strategy Model

```python
class Strategy(BaseModel):
    id: int
    user_id: int
    name: str
    strategy_type: StrategyType
    config: Dict[str, Any]
    status: StrategyStatus  # ACTIVE, INACTIVE, PAUSED
    performance: Dict[str, Any]
    created_at: datetime
    updated_at: datetime
```

6. Database Schema Design

6.1 Core Tables

Users and Accounts

```sql
-- Users table
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL,
    first_name VARCHAR(100),
    last_name VARCHAR(100),
    is_active BOOLEAN DEFAULT true,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);

-- Trading accounts
CREATE TABLE trading_accounts (
    id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES users(id),
    account_type VARCHAR(20) NOT NULL, -- spot, margin, futures
    exchange VARCHAR(50) NOT NULL,
    api_key_encrypted TEXT,
    api_secret_encrypted TEXT,
    is_active BOOLEAN DEFAULT true,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);
```

Orders and Trades

```sql
-- Orders table
CREATE TABLE orders (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id INTEGER REFERENCES users(id),
    account_id INTEGER REFERENCES trading_accounts(id),
```

```
    symbol VARCHAR(20) NOT NULL,
    side VARCHAR(10) NOT NULL, -- BUY, SELL
    order_type VARCHAR(20) NOT NULL, -- MARKET, LIMIT, STOP, etc.
    quantity DECIMAL(20,8) NOT NULL,
    price DECIMAL(20,8),
    stop_price DECIMAL(20,8),
    time_in_force VARCHAR(10) DEFAULT 'GTC',
    status VARCHAR(20) DEFAULT 'PENDING',
    filled_quantity DECIMAL(20,8) DEFAULT 0,
    average_price DECIMAL(20,8),
    fee DECIMAL(20,8) DEFAULT 0,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    expires_at TIMESTAMP WITH TIME ZONE
);

-- Trades table
CREATE TABLE trades (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    order_id UUID REFERENCES orders(id),
    symbol VARCHAR(20) NOT NULL,
    side VARCHAR(10) NOT NULL,
    quantity DECIMAL(20,8) NOT NULL,
    price DECIMAL(20,8) NOT NULL,
    fee DECIMAL(20,8) NOT NULL,
    timestamp TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    trade_type VARCHAR(10) -- MAKER, TAKER
);
```

Portfolio and Positions

```
-- Positions table
CREATE TABLE positions (
    id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES users(id),
    account_id INTEGER REFERENCES trading_accounts(id),
    symbol VARCHAR(20) NOT NULL,
    quantity DECIMAL(20,8) NOT NULL,
    average_price DECIMAL(20,8) NOT NULL,
    market_value DECIMAL(20,8),
    unrealized_pnl DECIMAL(20,8),
    realized_pnl DECIMAL(20,8),
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    UNIQUE(user_id, account_id, symbol)
);

-- Account balances
CREATE TABLE account_balances (
    id SERIAL PRIMARY KEY,
```

```
    account_id INTEGER REFERENCES trading_accounts(id),
    asset VARCHAR(20) NOT NULL,
    total_balance DECIMAL(20,8) NOT NULL,
    available_balance DECIMAL(20,8) NOT NULL,
    locked_balance DECIMAL(20,8) NOT NULL,
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    UNIQUE(account_id, asset)
);
```

Strategies

```
-- Trading strategies
CREATE TABLE trading_strategies (
    id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES users(id),
    name VARCHAR(255) NOT NULL,
    description TEXT,
    strategy_type VARCHAR(50) NOT NULL,
    config JSONB NOT NULL,
    status VARCHAR(20) DEFAULT 'INACTIVE',
    performance JSONB,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);

-- Strategy executions
CREATE TABLE strategy_executions (
    id SERIAL PRIMARY KEY,
    strategy_id INTEGER REFERENCES trading_strategies(id),
    execution_type VARCHAR(20) NOT NULL, -- backtest, paper, live
    status VARCHAR(20) DEFAULT 'PENDING',
    config JSONB,
    results JSONB,
    started_at TIMESTAMP WITH TIME ZONE,
    completed_at TIMESTAMP WITH TIME ZONE,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);
```

6.2 Time-Series Tables (TimescaleDB)

Market Data

```
-- Market data (OHLCV)
CREATE TABLE market_data (
    time TIMESTAMP WITH TIME ZONE NOT NULL,
    symbol VARCHAR(20) NOT NULL,
    exchange VARCHAR(50) NOT NULL,
    timeframe VARCHAR(10) NOT NULL,
    open_price DECIMAL(20,8) NOT NULL,
    high_price DECIMAL(20,8) NOT NULL,
```

```
    low_price DECIMAL(20,8) NOT NULL,
    close_price DECIMAL(20,8) NOT NULL,
    volume DECIMAL(20,8) NOT NULL,
    quote_volume DECIMAL(20,8),
    trades_count INTEGER,
    PRIMARY KEY (time, symbol, exchange, timeframe)
);

-- Convert to hypertable
SELECT create_hypertable('market_data', 'time');

-- Trading signals
CREATE TABLE trading_signals (
    time TIMESTAMP WITH TIME ZONE NOT NULL,
    strategy_id INTEGER NOT NULL,
    symbol VARCHAR(20) NOT NULL,
    signal_type VARCHAR(20) NOT NULL,
    strength DECIMAL(5,4) NOT NULL,
    confidence DECIMAL(5,4) NOT NULL,
    price DECIMAL(20,8),
    quantity DECIMAL(20,8),
    metadata JSONB,
    processed BOOLEAN DEFAULT false
);

SELECT create_hypertable('trading_signals', 'time');
```

Performance Metrics

```
-- Portfolio performance metrics
CREATE TABLE portfolio_metrics (
    time TIMESTAMP WITH TIME ZONE NOT NULL,
    user_id INTEGER NOT NULL,
    account_id INTEGER,
    total_value DECIMAL(20,8) NOT NULL,
    unrealized_pnl DECIMAL(20,8),
    realized_pnl DECIMAL(20,8),
    daily_return DECIMAL(10,6),
    cumulative_return DECIMAL(10,6),
    volatility DECIMAL(10,6),
    sharpe_ratio DECIMAL(10,6),
    max_drawdown DECIMAL(10,6),
    PRIMARY KEY (time, user_id, account_id)
);

SELECT create_hypertable('portfolio_metrics', 'time');
```

6.3 Indexes and Constraints

Performance Indexes

```sql
-- Orders indexes
CREATE INDEX idx_orders_user_id ON orders(user_id);
CREATE INDEX idx_orders_symbol ON orders(symbol);
CREATE INDEX idx_orders_status ON orders(status);
CREATE INDEX idx_orders_created_at ON orders(created_at);

-- Trades indexes
CREATE INDEX idx_trades_order_id ON trades(order_id);
CREATE INDEX idx_trades_symbol ON trades(symbol);
CREATE INDEX idx_trades_timestamp ON trades(timestamp);

-- Market data indexes
CREATE INDEX idx_market_data_symbol ON market_data(symbol, time DESC);
CREATE INDEX idx_market_data_exchange ON market_data(exchange, time DESC);

-- Trading signals indexes
CREATE INDEX idx_trading_signals_strategy ON trading_signals(strategy_id, time DESC);
CREATE INDEX idx_trading_signals_symbol ON trading_signals(symbol, time DESC);
```

7. Deployment Strategy

7.1 Containerization

Multi-stage Dockerfile

```dockerfile
# Build stage
FROM python:3.11-slim as builder

WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Production stage
FROM python:3.11-slim

WORKDIR /app
COPY --from=builder /usr/local/lib/python3.11/site-packages /usr/local/lib/python3.11/site-packag
COPY --from=builder /usr/local/bin /usr/local/bin

COPY . .

EXPOSE 8080
CMD ["python", "main.py"]
```

Kubernetes Deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: trading-engine
```

```yaml
  namespace: alphintra
spec:
  replicas: 3
  selector:
    matchLabels:
      app: trading-engine
  template:
    metadata:
      labels:
        app: trading-engine
    spec:
      containers:
      - name: trading-engine
        image: alphintra/trading-engine:latest
        ports:
        - containerPort: 8080
        env:
        - name: DATABASE_URL
          valueFrom:
            secretKeyRef:
              name: database-secret
              key: url
        - name: REDIS_URL
          valueFrom:
            secretKeyRef:
              name: redis-secret
              key: url
        resources:
          requests:
            memory: "1Gi"
            cpu: "500m"
          limits:
            memory: "2Gi"
            cpu: "1000m"
        livenessProbe:
          httpGet:
            path: /health
            port: 8080
          initialDelaySeconds: 30
          periodSeconds: 10
        readinessProbe:
          httpGet:
            path: /ready
            port: 8080
          initialDelaySeconds: 5
          periodSeconds: 5
```

7.2 Scaling Strategy

Horizontal Pod Autoscaler

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: trading-engine-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: trading-engine
  minReplicas: 3
  maxReplicas: 20
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
  - type: Resource
    resource:
      name: memory
      target:
        type: Utilization
        averageUtilization: 80
```

7.3 Monitoring and Observability

Prometheus Metrics

```python
from prometheus_client import Counter, Histogram, Gauge

# Metrics
orders_total = Counter('orders_total', 'Total orders processed', ['status'])
order_processing_time = Histogram('order_processing_seconds', 'Order processing time')
active_strategies = Gauge('active_strategies', 'Number of active strategies')
portfolio_value = Gauge('portfolio_value', 'Portfolio value', ['user_id'])
```

Grafana Dashboard Configuration

```json
{
  "dashboard": {
    "title": "Trading Engine Dashboard",
    "panels": [
      {
        "title": "Orders per Second",
        "targets": [
          {
```

```
            "expr": "rate(orders_total[5m])",
            "legendFormat": "Orders/sec"
          }
        ]
      },
      {
        "title": "Order Processing Time",
        "targets": [
          {
            "expr": "histogram_quantile(0.95, order_processing_seconds_bucket)",
            "legendFormat": "95th percentile"
          }
        ]
      }
    ]
  }
}
```

8. Testing Strategy

8.1 Unit Testing

```python
import pytest
from decimal import Decimal
from trading_engine.models import Order, OrderSide, OrderType

@pytest.fixture
def sample_order():
    return Order(
        user_id=1,
        symbol="BTCUSDT",
        side=OrderSide.BUY,
        order_type=OrderType.LIMIT,
        quantity=Decimal("1.0"),
        price=Decimal("50000.0")
    )

def test_order_creation(sample_order):
    assert sample_order.symbol == "BTCUSDT"
    assert sample_order.side == OrderSide.BUY
    assert sample_order.quantity == Decimal("1.0")

def test_order_validation(sample_order):
    assert sample_order.is_valid()

    # Test invalid quantity
    sample_order.quantity = Decimal("0")
    assert not sample_order.is_valid()
```

8.2 Integration Testing

```python
import pytest
from httpx import AsyncClient
from trading_engine.main import app

@pytest.mark.asyncio
async def test_create_order():
    async with AsyncClient(app=app, base_url="http://test") as client:
        order_data = {
            "symbol": "BTCUSDT",
            "side": "BUY",
            "order_type": "LIMIT",
            "quantity": "1.0",
            "price": "50000.0"
        }

        response = await client.post("/api/v1/orders", json=order_data)
        assert response.status_code == 201
        assert response.json()["symbol"] == "BTCUSDT"
```

8.3 Performance Testing

```python
import asyncio
import aiohttp
import time

async def create_order_request(session, order_data):
    async with session.post('/api/v1/orders', json=order_data) as response:
        return await response.json()

async def performance_test():
    order_data = {
        "symbol": "BTCUSDT",
        "side": "BUY",
        "order_type": "MARKET",
        "quantity": "0.001"
    }

    async with aiohttp.ClientSession() as session:
        start_time = time.time()

        # Create 1000 concurrent orders
        tasks = [create_order_request(session, order_data) for _ in range(1000)]
        results = await asyncio.gather(*tasks)

        end_time = time.time()

        print(f"Processed {len(results)} orders in {end_time - start_time:.2f} seconds")
        print(f"Orders per second: {len(results) / (end_time - start_time):.2f}")
```

9. Security Considerations

9.1 Authentication and Authorization

- **JWT Tokens**: Secure authentication with configurable expiration
- **Role-based Access Control**: Different permissions for different user roles
- **API Rate Limiting**: Prevent abuse and ensure fair usage
- **Multi-factor Authentication**: Additional security for sensitive operations

9.2 Data Protection

- **Encryption at Rest**: Database encryption for sensitive data
- **Encryption in Transit**: TLS/SSL for all communication
- **API Key Management**: Secure storage and rotation of exchange API keys
- **Audit Logging**: Comprehensive logging of all trading activities

9.3 Trading Security

- **Order Validation**: Comprehensive validation to prevent invalid orders
- **Position Limits**: Prevent excessive position sizes
- **Fraud Detection**: Anomaly detection for suspicious trading patterns
- **Compliance Monitoring**: Automated compliance checks

10. Performance Optimization

10.1 Database Optimization

- **Connection Pooling**: Efficient database connection management
- **Query Optimization**: Optimized queries with proper indexing
- **Read Replicas**: Separate read and write operations
- **Caching Strategy**: Multi-layer caching for frequently accessed data

10.2 Application Optimization

- **Async Processing**: Non-blocking I/O operations
- **Memory Management**: Efficient memory usage and garbage collection
- **Connection Reuse**: Reuse HTTP connections for external APIs
- **Batch Processing**: Batch operations for improved throughput

10.3 Network Optimization

- **WebSocket Connections**: Persistent connections for real-time data
- **Data Compression**: Compress data transmission
- **CDN Integration**: Content delivery network for static assets
- **Load Balancing**: Distribute traffic across multiple instances

11. Monitoring and Alerting

11.1 System Metrics

- **CPU and Memory Usage**: Resource utilization monitoring
- **Database Performance**: Query performance and connection metrics

- **Network Latency**: End-to-end latency monitoring
- **Error Rates**: Error tracking and alerting

11.2 Business Metrics

- **Trading Volume**: Monitor trading activity
- **Order Fill Rates**: Track order execution success
- **Strategy Performance**: Monitor strategy profitability
- **Risk Metrics**: Real-time risk monitoring

11.3 Alerting

- **Threshold Alerts**: Automated alerts for metric thresholds
- **Anomaly Detection**: Machine learning-based anomaly detection
- **Escalation Procedures**: Automated escalation for critical issues
- **Integration**: Slack, PagerDuty, and email notifications

12. Maintenance and Operations

12.1 Deployment Procedures

- **Blue-Green Deployment**: Zero-downtime deployments
- **Canary Releases**: Gradual rollout of new features
- **Rollback Procedures**: Quick rollback in case of issues
- **Database Migrations**: Safe database schema changes

12.2 Backup and Recovery

- **Database Backups**: Regular automated backups
- **Point-in-Time Recovery**: Ability to restore to specific timestamps
- **Disaster Recovery**: Cross-region backup and recovery
- **Testing**: Regular testing of backup and recovery procedures

12.3 Capacity Planning

- **Growth Projections**: Predict future resource needs
- **Resource Scaling**: Automated scaling based on demand
- **Performance Baselines**: Establish performance benchmarks
- **Cost Optimization**: Optimize resource usage and costs

13. Documentation and Training

13.1 Technical Documentation

- **API Documentation**: Comprehensive API documentation with examples
- **Architecture Documentation**: System design and architecture
- **Deployment Documentation**: Deployment and operations procedures
- **Troubleshooting Guide**: Common issues and solutions

13.2 User Documentation

- **User Guide**: End-user documentation for traders
- **Strategy Guide**: Documentation for strategy development
- **Risk Management Guide**: Risk management best practices
- **FAQ**: Frequently asked questions and answers

13.3 Training Program

- **Developer Training**: Technical training for development team
- **Operations Training**: Training for operations and support team
- **User Training**: Training for end users and traders
- **Regular Updates**: Ongoing training for new features and updates

14. Success Metrics and KPIs

14.1 Performance Metrics

- **Order Latency**: $<$ 10ms for order processing
- **System Uptime**: 99.9% availability
- **Throughput**: 10,000+ orders per second
- **Error Rate**: $<$ 0.1% error rate

14.2 Business Metrics

- **User Adoption**: Number of active traders
- **Trading Volume**: Monthly trading volume
- **Strategy Performance**: Average strategy returns
- **Customer Satisfaction**: User satisfaction scores

14.3 Operational Metrics

- **Deployment Frequency**: Time between deployments
- **Mean Time to Recovery**: Time to resolve issues
- **Code Quality**: Test coverage and code quality metrics
- **Team Velocity**: Development team productivity

15. Conclusion

This comprehensive development plan provides a roadmap for creating a world-class trading engine backend service that consolidates and enhances the existing trading infrastructure. The unified architecture will provide better performance, scalability, and maintainability while offering advanced trading features that can compete with institutional-grade trading platforms.

The 40-week development timeline is structured to deliver incremental value while building towards a complete, production-ready trading engine. Each phase builds upon the previous one, ensuring a solid foundation while progressively adding advanced features.

Key success factors include: - **Strong technical foundation** with proper architecture and infrastructure - **Comprehensive testing** at all levels (unit, integration, performance) - **Robust security** and risk management capabilities - **Scalable design** that can grow with the

business - **Thorough documentation** and training programs - **Continuous monitoring** and improvement processes

This trading engine will position Alphintra as a leader in the algorithmic trading space, providing users with institutional-quality trading capabilities in a user-friendly platform.